INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.

University Microfilms International
300 N. ZEEB ROAD, ANN ARBOR, MI 48106
18 BEDFORD ROW, LONDON WC1R 4EJ, ENGLAND
THE RECOGNITION OF STRAIGHT LINE PATTERNS BY
BUS AUTOMATONS USING PARALLEL PROCESSING

DISSERTATION

Presented in Partial Fulfillment of the
Requirements for the Degree Doctor of
Philosophy in the Graduate School of
The Ohio State University

by
John Rolf Mellby, B.A.

* * * * *

The Ohio State University
1980

Reading Committee:
Professor Jerome Rothstein
Professor S. H. Zweben
Professor B. Chandrasekaran

Approved By
Professor Jerome Rothstein
Department of Computer & Information Science
Acknowledgments

It can be said without a doubt that Professor Rothstein put at least as much effort into this work as I did. He patiently endured my foibles and delays, and guided and encouraged me to the final production of this report.
VITA

PERMANENT ADDRESS

1451 3rd St. S.E.
Apt. 111B
St. Cloud, Mn 56301
612/255-0071

FIELDS OF INTEREST

Parallel Processing  Pattern Recognition
Programming Languages  Statistics
Automata

EDUCATION

Ph.D. from the Ohio State University in June, 1980.
Majored in Computer and Information Sciences.
Earned a 3.82 grade point average (on a 4.0 scale).
Dissertation: The Recognition of Straight Line
Patterns by Bus Automatons using
Parallel Processing
Adviser: Jerome Rothstein

B.A. from St. Olaf College in May, 1973. Double
major in Mathematics and Physics. Earned a 3.6
grade point average (on a 4.0 scale).

DISTINCTIONS

Recipient of University Fellowship from Ohio State

Graduated Magna Cum Laude with honors in math
from St. Olaf College in three years.

Recipient of St. Olaf College award for achievement
on the Putnam Mathematics Examination.

Recipient of NSF Honorary Scholarship in 1970.
EXPERIENCE

9/79-6/80 Assistant Professor of Math & Computer Science at St. Cloud State University. Responsibilities included teaching and Curriculum development in Computer Science.

1/79-8/79 & 9/74-6/76 Teaching Assistant for the Department of Computer and Information Science at the Ohio State University. Responsibilities involved course preparation and teaching duties.

9/70-5/73 Assistant Systems Programmer for the computer center at St. Olaf College. Duties included maintaining and developing systems programs.

Summer 1972 Student Researcher in the NSF Undergraduate Research Program at St. Olaf College. Research investigated possible applications of algebra.

Personal

Date of Birth: April 2, 1952
Marital Status: Married to Priscilla Jane Graham
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>VITA</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. BUS AUTOMATA AND STRAIGHT LINES</td>
<td>8</td>
</tr>
<tr>
<td>Rothstein's Line Code</td>
<td>8</td>
</tr>
<tr>
<td>Bus Automaton Definition</td>
<td>12</td>
</tr>
<tr>
<td>Immediate Routines</td>
<td>13</td>
</tr>
<tr>
<td>3. DEFINITIONS AND CONVENTIONS</td>
<td>15</td>
</tr>
<tr>
<td>Geometric Terms</td>
<td>15</td>
</tr>
<tr>
<td>Bus Elements</td>
<td>16</td>
</tr>
<tr>
<td>Trace Diagrams</td>
<td>18</td>
</tr>
<tr>
<td>4. STATISTICS</td>
<td>21</td>
</tr>
<tr>
<td>Model</td>
<td>21</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>22</td>
</tr>
<tr>
<td>Non-Parametric Method</td>
<td>24</td>
</tr>
<tr>
<td>Orthogonal Regression</td>
<td>25</td>
</tr>
<tr>
<td>Best Line</td>
<td>26</td>
</tr>
<tr>
<td>5. BUS CONSTRUCTION AND DATA FLOW</td>
<td>28</td>
</tr>
<tr>
<td>Command Busses and Bus Pattern</td>
<td>28</td>
</tr>
<tr>
<td>Construction</td>
<td>28</td>
</tr>
<tr>
<td>Compression Routines</td>
<td>36</td>
</tr>
<tr>
<td>Line Compression</td>
<td>38</td>
</tr>
<tr>
<td>Line Expansion</td>
<td>39</td>
</tr>
<tr>
<td>Area Compression</td>
<td>41</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Sum of Carry and Intermediate</td>
<td>69</td>
</tr>
<tr>
<td>2. Multiplying 110101&lt;sub&gt;two&lt;/sub&gt; by 11101&lt;sub&gt;two&lt;/sub&gt;</td>
<td>73</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Standard 2-dimensional Neighborhood</td>
<td>4</td>
</tr>
<tr>
<td>2. Types of cells in a Line Pattern</td>
<td>9</td>
</tr>
<tr>
<td>3. Code Segments</td>
<td>9</td>
</tr>
<tr>
<td>4. Bus Automata Cell</td>
<td>14</td>
</tr>
<tr>
<td>5. Geometric Terms</td>
<td>17</td>
</tr>
<tr>
<td>6. Bus Directions</td>
<td>17</td>
</tr>
<tr>
<td>7. Trace Diagram</td>
<td>19</td>
</tr>
<tr>
<td>8. Desired Configuration</td>
<td>30</td>
</tr>
<tr>
<td>9. Initial Cells</td>
<td>30</td>
</tr>
<tr>
<td>10. Step 1</td>
<td>32</td>
</tr>
<tr>
<td>11. Step 2, Fourth Corner</td>
<td>32</td>
</tr>
<tr>
<td>12. Step 3, Limit Boundaries</td>
<td>32</td>
</tr>
<tr>
<td>13. Step 4</td>
<td>32</td>
</tr>
<tr>
<td>14. Infinite Rectangle</td>
<td>32</td>
</tr>
<tr>
<td>15. Infinite Traingle</td>
<td>32</td>
</tr>
<tr>
<td>16. Initial Configuration, Busses from a Pattern</td>
<td>37</td>
</tr>
<tr>
<td>17. Creation of Busses from a Pattern</td>
<td>37</td>
</tr>
<tr>
<td>18. Data set transferring past Data Cell</td>
<td>37</td>
</tr>
<tr>
<td>19. Adding new Data Element</td>
<td>37</td>
</tr>
<tr>
<td>20. Compressing Data</td>
<td>39</td>
</tr>
<tr>
<td>21. Data Point Busses</td>
<td>39</td>
</tr>
<tr>
<td>22. Non-Data Point Busses</td>
<td>39</td>
</tr>
<tr>
<td>23. Expansion Busses</td>
<td>39</td>
</tr>
<tr>
<td>24. Initial Area, Compression Configuration</td>
<td>39</td>
</tr>
<tr>
<td>25. Order of Data Transfer</td>
<td>39</td>
</tr>
<tr>
<td>26. First Row Transfer</td>
<td>39</td>
</tr>
<tr>
<td>27. Goal Plane Busses</td>
<td>42</td>
</tr>
<tr>
<td>28. Ith Plane</td>
<td>42</td>
</tr>
<tr>
<td>29. Transfer to Lower Plane</td>
<td>42</td>
</tr>
<tr>
<td>30. Complete Ith plane Busses</td>
<td>42</td>
</tr>
<tr>
<td>31. Goal Line</td>
<td>42</td>
</tr>
<tr>
<td>32. Cells above Goal Line</td>
<td>42</td>
</tr>
<tr>
<td>33. Boundary opposite Goal Line</td>
<td>42</td>
</tr>
<tr>
<td>34. Shift cells (Diagonal Field)</td>
<td>42</td>
</tr>
<tr>
<td>35. Addition, Initial Configuration</td>
<td>45</td>
</tr>
<tr>
<td>36. Addition Process</td>
<td>45</td>
</tr>
<tr>
<td>37. Addition Busses</td>
<td>45</td>
</tr>
<tr>
<td>38. Diagonal Bus</td>
<td>45</td>
</tr>
<tr>
<td>39. Interval Marking Busses</td>
<td>45</td>
</tr>
</tbody>
</table>
88. Basic Structure 71
89. Addition Busses 71
90. Division by 2 Busses 71
91. Carry Busses for Column k 74
92. Shifted Multiplicand 74
93. Shifted Copies of Multiplicand 74
94. Busses for Multiplicand Shift 74
95. Busses for Multiplier 74
96. Line Code Determination 78
97. '1' Row Adds a '1' Bus and Absorbs a '0' Bus 78
98. Code Compression 78
99. Generated Line 78
100. Run Cell 78
101. Rise Cell 78
102. Types of Cells in a Pattern 78
103. Determining Code of Pattern 78
104. Data Representing a Line 83
105. y-z Plane containing 3rd x-row 83
106. Compression, the number of points in the Row 83
107. Summing: \( x_i \) 83
108. Data in x-z Plane 83
109. Multiplication Busses for \( x=3 \) 83
110. \( x^2 \) 83
111. Sending \( x^2 \) 86
112. Busses for Multiplication by \( x^2 \) 86
113. Plane receives its x value 86
114. Multiplication busses for \( x=2 \) 86
115. \( x^2 \) 86
116. Duplicate \( x^2 \) 86
117. Busses for Multiplication and Compression 86
118. Multiply data by \( x^2 \) 86
119. Sawtooth Producing \( i \mod N \) 88
120. Stroke Addition 102
121. Numbers to be Sorted 109
122. Compacting to Sort 109
123. Sorting Digits 109
Chapter 1: Introduction

Lately there has been much interest in parallel processing, i.e. applying many computers simultaneously to a complex task. We present some results in which a parallel combination of machines performs complex computations and recognizes patterns very rapidly.

Some of the approaches to the parallel combination of computing elements achieve parallelism by producing and combining elements specifically designed for parallelism (Batcher, 1968; Chen, 1975), however a more common approach is to combine conventional computers. Although we do not, in this work use conventional computers, insights into parallel problems may be obtained by observing several aspects of combining conventional computers in parallel.

Two subjects must be considered when combining conventional computers, the number of computers in the parallel system, and the process of communication between computers. Most authors, if they consider the number of computers at all, limit the systems to a very small number of computers (because of their cost). The Illiac IV is a good example of this type of approach. Communication between computers is either direct CPU-CPU
link (Agerwala and Lint, 1978), or by a shared memory (Foster, 1976). Our work sidesteps both problems by using a cellular automaton. Each computing unit (cell) is very inexpensive, so there may be any number of cells available, and each cell can communicate instantly (negligible time) with its neighbors.

Automatons have been used in parallel processing before (Chen, 1975; Smith, 1972; Smith, Lin, and Shen, 1975), however communication between distant cells was a handicap. Rothstein (1976, 1978) developed a fast communications line (Busses) to overcome this handicap. This approach is taken here.

Two problems which can be directly compared to our work is that of parallel evaluation of arithmetic expressions, and parallel sorting. It has been shown an arithmetic expression can be evaluated in \( O(\log n) \) time (Beaty, 1974; Brent, 1974; Brent, Kuck, and Maruyama, 1973), if it is properly parsed (Towle and Brent, 1976). A list of keys may be sorted in \( O(\log n) \) time by a parallel combination of machines (Batcher, 1968; Lee and Fend, 1975; Preparata, 1977). These two types of parallel algorithms will later be compared to our results; we achieve "immediacy", i.e. \( O(1) \) time.

A Cellular automaton is an array of identical finite state automata. It is probably the simplest collection of automata where theoretical discussion is essentially
unchanged as the number of automata (cells) becomes indefinitely large. It is therefore a good starting point to begin a study of the capabilities of parallel operation. In every cycle of its operation each cell receives inputs and computes its next state. As such, the cellular automaton, or CA, is a parallel computer. It has been shown that the CA is capable of anything any other computer is capable of, but CA algorithms are slow. The reason for investigating parallel computers is not to discover whether or not they can solve problems, but to see how much faster they can solve them. In the case of CA's their lack of speed is a handicap.

The slowness of CA's results from limited direct intercellular communication. Each cell can receive inputs only from a finite set of cells called its neighbors. A "standard" neighborhood for a cell (in a 2-dimensional CA) is the eight cells immediately adjacent to that cell (figure 1). The "radius of this neighborhood is one cell. If this cell needs to send a signal to another k cells away, it will take at least k cycles. In effect, the fixed finite neighborhoods restrict each cell's rapid access to information.

Rothstein alleviated this communication problem by changing the CA into a fast parallel computer which he called a Bus Automaton (BA). The BA has the ability to construct rapid communication channels (busses)
1 Standard 2-dimensional Neighborhood
through every cell. On these channels signals can be sent from any cell of the BA to any other cell. With this ability the BA becomes an efficient parallel computer.

While most parallel systems assume a fixed maximum number of computers available to each problem, our system assumes (since each cell is a small and hence cheap finite state automaton) that the BA can involve any number of cells. In actuality this number is limited by communication delay, but in this research we assume that this lag can be neglected.

This work originally intended to investigate pattern recognition capabilities of BA, however as the work progressed, we were drawn more into studying its computational ability. Many results are useful in wider contexts than pattern recognition. We have carried them beyond the needs of the original problem because of their intrinsic importance and interest.

The initial problem was to recognize the straight line best fitting a set of data points in a plane. In the course of solving this problem we had to work out numerous computational and statistical routines. All have been made "immediate", so that straight lines can be fitted statistically to given data, by parallel computation, in the minimum time needed to perform the simplest kind of computation.
Chapter two reviews the relevant early work on BA's and on straight line recognition by CA's. BA's are defined and the basis for comparing speeds of certain routines is developed, including the concept of immediacy. The straight line code is reviewed.

Chapter three begins with the definitions and conventions which will be used in our presentation. The method by which formal BA algorithms are presented (called trace diagrams) is explained.

Chapter four defines the statistical straight line problem in detail. The statistical model representing the data is presented, and various statistical methods of solution are examined. Orthogonal regression is the final method selected and justification is given for this choice.

Chapter five presents certain basic BA routines. The underlying bus structure is defined, then using this we show how specific bus configurations are constructed. Virtually all parallel routines require construction of specific configurations of busses. We develop several basic routines to move or transform data to different locations and to modify its configuration in various ways.

The first form of computation on BA's is arithmetic in stroke notation. In chapter six arithmetic routines for stroke numbers are demonstrated.
To show BA's flexibility, chapter seven shows BA's ability to work with binary numbers also. The BA can convert immediately from stroke to binary (and back). Addition and multiplication also prove to be immediate.

Chapter eight returns to routines needed to handle the straight line. Using previous routines general problems involving the recognition and construction of straight lines are solved. The remaining routines necessary for the complete solution of our original line recognition problem are produced.

Chapter nine demonstrates the process of "joining" several bus routines. In this way the original straight line problem is solved immediately, i.e. Faster than any sequential computer.

Chapter ten reviews the results obtained in this dissertation. The speed and flexibility of the BA for general problems are summarized and compared to sequential computers. The BA is shown to be at least equal to, and generally superior to any sequential computer or conventional parallel combination of computers. We examine some limitations of BA performance arising from finite signal propagation velocity. They turn out to have sequential analogues in timing and overflow problems.
Chapter 2: Bus Automatons and Straight Lines

Before we introduce the BA itself we review the concepts to be used in dealing with straight lines. These were originally introduced by Rothstein (Rothstein and Weiman, 1972).

2.1 Rothstein's Line Code

One of the more powerful tools of the previous research concerning lines is Rothstein's code. We begin with a line drawn on a NxN grid. Rothstein's code produces a N-digit binary number which specifies this line (to the limit of the grid's sensitivity).

We can assume without loss of generality that the line's slope is in the first quadrant, and moreover that the slope is between 0° and 45°, so it is in the first octant. (All other octants may be handled similarly.)

If we examine those active cells, i.e. those cells through which the line passes, each cell can be in one of three states called "run", "lower rise", and "upper rise". The run state exists when the line passes through both vertical sides of the cell, i.e. the line "runs"
2. Types of Cells in a Line Pattern

A. Run

B. Lower Rise

C. Upper Rise

3. Code Segments
straight (figure 2). If the line passes through a horizontal cell side, it will cause "rise" states. These states occur in pairs, as the line rises from the "lower rise" cell up to the "upper rise" cell (figures 2.B, 2.C).

If we proceed along the grid in the x direction, observing the cells along the line, each column of the grid contains either a run cell, or a pair of rise cells. (The rise pair is considered as one unit.) Then there is only one activated unit on each vertical column of the grid. Since run indicates no vertical change in that cell, it is called "0". The rise indicates a change of one line of cells, and is called "1". This set of 0-1 digits is an N-digit representation for this line and is the Rothstein Code of that line.

Not all N-digit binary strings are the codes of straight lines. It is possible to break the code down into a set of separate segments of digits defined by O*1. The segments which compose the pattern of a line are interrelated. For a specific line, the lengths of its segments cannot differ by more than one. The proof of this, (p. 109, Rothstein and Weiman, 1976) is as follows: A segment of length k, has k-1 runs terminated at both ends by rises. The line travels vertically one unit in this segment. The minimum horizontal distance the line could have traveled is k-1, and the maximum, k+1. The slope, m, is then constrained by
If another segment of that line is of length $k+h$ then by the same argument,

\[
\frac{1}{k+1} \leq m \leq \frac{1}{k-1}.
\]

These constraints together imply that if $h$ is not equal to 0, it can only be either 1 or -1. Then one line will have segments of length $k$ and $k+1$ only.

These segments of two different lengths are organized in patterns just as the digits are. If we substitute for each short segment a 0 and long segment a 1, we again have segments of 0's followed by a 1. The length of these supra-segments varies by only one just as the original segments. If we again substitute 0 and 1 for segments we can continue forming segments of segments. If the code represents a line, it will eventually reduce to a single digit. See figure 3.

This is one of the line recognition algorithms described in Rothstein and Weiman (1976). If at any point the segment lengths differ by more than one, the original pattern is not a straight line. (Another of Rothstein and Weiman's (1976) straight line algorithms will be used to recognize lines; see chapter 8.)

This algorithm to recognize lines used a primitive form of busses. Busses were clearly formalized in Rothstein (1976) and Moschell and Rothstein (1979).
2.2 Definition of Bus Automaton

A CA consists of an array of cells, where each cell is a FSA. During a clock cycle each cell receives input from its neighbors, then from this and its own state the cell computes its next state.

In a BA each cell contains a FSA but it also contains bus connections (see 3.2). (Cell State = FSA State + Bus Connections.) Each cell can send signals on any output bus to which it is connected. These signals propagate to each cell connected to this bus as an input bus. We assume signal propagation to be fast enough that all signals propagate before the next step.

Each cell receives inputs from its neighbors and its input busses. Then from these inputs and its state, the cell computes its next state. Finally the cell can change its bus connections on each state change, after which the cycle repeats (or the process halts). A cell state change is often conveniently viewed as a doublet, i.e. a FSA state change plus a change in bus connections.

In detail, in the first part of a cycle, each cell sends signals onto its output busses. The time allowed to propagate these signals, $T_p$, is here assumed for convenience in design, to be much less than one clock pulse, $T_s$. The signals may then propagate to any cell before step 2 of the cycle. In step 2 the FSA receives
inputs from its neighbors and from its input busses. Step 3 consists of the FSA computing from its inputs the necessary change to the FSA state including bus connections. Finally, in step 4, the FSA brings about the new bus connections and alters its own state. See figure 4.

Actually, no matter how short the propagation time $T_p$ is, if a bus is long enough, the signal will not reach all the cells in one cycle. However since we intend the signal to propagate at (or close to) the speed of light, the propagation time will only be bothersome for very long busses. This problem will be considered further in chapter 10.

2.3 Immediate Routines

Rothstein (1976) and Moshell and Rothstein (1979) define the concept of immediate operations and languages. An immediate operation is one which can be done in a fixed finite number of cycles no matter how long the input, $O(1)$. For example, an immediate language is one where each word can be recognized immediately, no matter how long the word. Rothstein and Moshell's work investigated immediate languages, relating a hierarchy of these languages to standard hierarchies of formal languages. The language of all straight line codes in context sensitive (Rothstein and Weiman, 1975) and is later shown to be immediate.
4. Bus Automata Cell
Chapter 3: Definitions and Conventions

In this chapter we set forth a number of definitions and conventions useful in specifying the particular BA's used to solve the problems with which we are concerned. (Many alternative definitions, conventions and BA designs are, of course, possible.) First we adopt several conventions concerning the geometry of the BA and the diagrams describing it. We standardize our bus description terminology. Lastly we construct a time trace diagram depicting the execution of BA algorithms. These are useful for BA design and for checking algorithm correctness.

3.1 Geometric Terms

As many of our routines require a three-dimensional BA, a single two-dimensional diagram does not always clearly describe the routine. Hence we will at times be talking about parts of a BA, either planar sections of the three-dimensional BA or one-dimensional sections. When we refer to lines of cells, we will call these by different names depending on their orientation. A row will be a section whose x-coordinate varies, a rank or
line will indicate a varying y-coordinate, and column will mean a varying z-coordinate (figure 5).

To designate specific planes we will usually describe them in the natural way, by those coordinates which vary in that plane; e.g. the x-y plane in figure 5.

3.2 Bus Elements

Depending on how broad a picture we are observing we may use three terms for busses. First, a bus connection or a channel is a communication connection between an input link and an output link of one cell. A bus is an alternating concatenation of links and channels from a cell serving as a message source to a set of cells serving as message destinations. A single (directional) link always joins a pair of nearest neighbors.

In this work these communication nearest neighbors are also the geometric nearest neighbor cells in a lattice (cf. the standard neighborhood of figure 1 and the 3-dimensional case in figure 6 below).

A bus configuration is a specific set of busses constructed to perform a specific task.

When we wish to define a particular channel in a cell we must specify its input and output links. Accordingly we number the neighboring cells in two-space 1-8 (figure 1). In three-space the neighboring planes are the "+" and "-" planes and the cells in those
5. Geometric Terms

6. Bus Directions
planes are $\pm 0-8$ (figure 6).

Then the channel designation consists of three parts. First, a letter to indicate incoming, R (receive), or outgoing, T (transmit). Second, the above code is used to indicate the direction of the link involved. Finally, if there are several busses in one direction, we may add a digit to indicate which particular bus in that direction is used. Then R.+0 T.3.2 is the channel which receives a signal from the cell directly above, and transmits it over the second link in the positive x direction.

3.3 Trace Diagrams

To help describe the execution of some of these operations we use what we call trace diagrams. They are essentially timing diagrams consisting of labeled boxes and arrows. Each box represents a cell, or a set of cells which undergo identical transformations. The box label describes the state of the cell.

Single line arrows from one box to another indicate a state change. The line may have two labels. The label below the arrow (sublabel) describes the signals which cause the state change. No sublabel indicates that the change occurs automatically. The label above the arrow (superlabel) tells whether command busses (see chapter 5) are interrupted by this state change.
7. Trace Diagrams
Double line arrows indicate signals being sent. The arrow goes from the cell which sends the signal to the cell which receives it. The label is the name of the signal. Figure 7 shows a trace diagram for the Rectangular Bus Configuration Constructor of section 5.1.

The states at the left of the diagram are the beginning states of the operation. The states at the right are the final states. As we go from left to right the diagram specifies what states each cell or set of cells enters. Also, as we progress towards the right, activated cells, previously quiescent (state B.0), appear in the appropriate step of the diagram.
Chapter 4: Statistics

The purpose of this chapter is to determine what process produces the "best" line for a set of data. We first explain the statistical model which we assume produces the data points. Next we examine three statistical processes which determine lines, linear regression, a non-parametric method, and orthogonal regression. Finally we select orthogonal regression as producing the "best" line in the context of our model and give justification for this choice.

4.1 Model

Assume a target exists which has a line on it. A point is picked on the line (randomly) and a dart is thrown at it. This generates one data point.

The distribution of the actual data points around the point aimed at is a normal distribution. The distance between the two points is normally distributed, and the direction from the aiming point to the data point is uniform over $360^\circ$. The aiming points are uniformly distributed along the line, so the perpendicular distance from the line to each data point is normally distributed.
The line is represented by

(1) \( y = \alpha + \beta x, \)

where \( \alpha \) is the y-intercept and \( \beta \) the slope. The set of \( (N) \) data points exists \((x_i, y_i)\) such that they are distributed around (1) with the distance to the line normal. The line will be completely determined by one point on the line and the slope of the line.

All approaches indicate that the most likely point to be on the line is the centroid of the data \((\bar{x}, \bar{y})\), Cramer (pp. 270-285, 1946). Thus, the centroid is chosen as the estimate for the point on the line, and only the slope remains to be determined.

We will consider three basic approaches for the determination of the line's slope. They are: Linear regression, non-parametric regression, and orthogonal regression.

4.2 Linear Regression

The goal of linear regression is to determine the parameters in (1), such that the probability of producing the data points is maximized. The regression of \( y \) on \( x \) first assumes each \( x_i \) to be fixed and all the error is in the \( y_i \) measurement. Then the distance to the line from the data point is

(2) \( \Delta y = y_i - (\bar{y} + \beta x_i). \)

Since \( \Delta y \) is normally distributed, the probability
density of this is

\[ P(x_1, y_1) = \frac{1}{\sqrt{2\pi} \sigma} e^{-\frac{1}{2} \left( \frac{y_1 - (\alpha + \beta x_1)^2}{\sigma^2} \right)} \]

Then the likelihood, or probability of the set of data points (given \( \alpha, \beta \)) is

\[ L(\alpha, \beta) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi} \sigma} e^{-\frac{1}{2} \left[ \frac{(y_i - (\alpha + \beta x_i))^2}{\sigma^2} \right]} = \left( \frac{1}{2\pi \sigma^2} \right) e^{\frac{1}{2} \sum_{i=1}^{N} \left( \frac{y_i - (\alpha + \beta x_i)^2}{\sigma^2} \right)} \]

We want to find the estimates of \( \alpha, \beta \) which maximize \( L \), (which is the same as maximizing \( \log(L) \)). To maximize \( L \) we must minimize the exponent. This is minimizing

\[ \sum_{i=1}^{N} \left( \frac{y_i - (\alpha + \beta x_i)^2}{\sigma^2} \right) \]

which is the principle of least squares.

Minimizing (5) with respect to \( \alpha, \beta \) we obtain the Maximum Likelihood Estimates:

\[ \hat{\beta} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2}, \quad \hat{\alpha} = \bar{y} - \hat{\beta} \bar{x}. \]

This is the regression line of \( y \) on \( x \).

One difficulty of this is that we are assuming the \( x_i \)'s are fixed and then corresponding \( y_i \)'s determined experimentally. It is possible to assume some error in the \( x_i \) points, but this does not alter the basic form of the results, it merely adds another term into the estimate for \( \beta \), (Bartlett, 1949),

\[ \hat{\beta}' = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2 - \sum \sigma_i^2} \]

There is another more serious problem with the
regression line of $y$ on $x$. For the same data we have another regression line, the regression of $x$ on $y$. In this case instead of minimizing the vertical distances to the line, we want to minimize the horizontal distances to the line. The $y_i$'s are assumed to be fixed and the $x_i$'s determined experimentally. The resulting line is determined by:

$$ (8) \hat{\beta}_{x \text{ on } y} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} , \hat{\alpha}_{x \text{ on } y} = \bar{y} - \hat{\beta}_{x \text{ on } y} \bar{x}. $$

Now we have two different regression lines, and we are faced with the problem of choosing between them as the "best" line (see Rothstein, p. 1, 1977). We will postpone this decision for a moment until we consider yet another regression line.

### 4.3 Non-Parametric Method

Part of our model is that the distance to the line is normal. While many real-world situations are normal, and many other distributions become normal in the limit, in some cases this may be too big a restriction.

Here is a distribution-free test for estimating the slope of a line (Thiel's method from Non-Parametric Method)

The idea behind this test is to examine all possible slopes between two data points taking the median of these slopes to represent the true slope of the line.

1. Form the $M = \left\lfloor \frac{N}{2} \right\rfloor$ sample slopes
2. Order the sample slopes $S^a$: $S^1 \leq S^2 \leq \ldots \leq S^m$.

3. Take the median sample slope as our estimate of $\beta$,

$$
\beta = \frac{S^{(M/2+1)}}{S^{(M/2)}} \text{ if } M \text{ odd}, \\
\frac{S^{(M+1)/2}}{S^{(M/2)}} \text{ if } M \text{ even}.
$$

The difficulty of this approach is that it doesn't take all the information of our model into account. It does not consider that the data was generated under a normal distribution. Also the estimate is not a maximum likelihood estimate. Although the non-parametric test generates an estimate of the slope, (and there may be other estimates of the slope) the maximum likelihood estimate, by its definition, is an estimate which is superior to any estimate which is not a maximum likelihood estimate. Because of this, we will not consider the non-parametric test further.

4.4 Orthogonal Regression

This regression again finds the maximum likelihood estimate of $\gamma, \beta$, but rather than minimizing the horizontal or vertical distance to the line, we now minimize the orthogonal distance to the line. The minimization is not nearly as simple, but the result (Goldberger, pp. 23-24, 1968; Cramer, pp. 270-285, 1946; Rothstein, p. 2, 1977) is
Now we have three regression lines. Since the orthogonal regression works with the minimum distance to the line it would seem to be the most accurate, however there is a reason why this has not come into general use as have linear regressions. In linear regression, if we change the scale, the line is invariant. That is, if we multiply all y values by a factor k, the slope of the line will also increase by k. In another sense, if the data and linear regression line are plotted, the scale by which we measure the data can be changed and the line will remain unchanged.

The orthogonal line is not invariant to scale changes. The slope of the orthogonal line is between that for the two regressions, but as the scale changes, the orthogonal line will also change.

The orthogonal line does remain invariant under rotation, while the linear regressions, being defined by the axis, must change under rotation.

4.5 Best Line

Since the non-parametric estimate is not a maximum likelihood estimate we have decided to reject that estimate, and will choose between the linear and
orthogonal regression.

Our model assumes the perpendicular distance to the line to be normal. In effect it is a model of visual data. Linear regression minimizes vertical (or horizontal) error. This does not correspond to our model, while orthogonal regression minimizes the perpendicular distance, which exactly corresponds to our model's error. Also, one can argue that in visual patterns, rotational invariance, rather than scale invariance is important, thus implying that orthogonal regression is appropriate rather than linear regression. This point is discussed at length, in a more general context, by Rothstein (1977).

In short, although linear regression is used more frequently, in our model orthogonal regression is more appropriate. Thus to find the best line we minimize the squared orthogonal distances, of the data, to the line.
5.1 Command Busses and Bus Pattern Construction

Much of the processing to be done by our BA consists of restructuring or moving large amounts of data. The ability to do this is what gives the BA its great speed. To do this we use a large set of special purpose busses. In general they are not already active but must be connected up in configurations relevant to the particular problem. To construct these connection patterns (special purpose busses) rapidly we need fast access to many cells. We provide command busses to enable any cell to help create special purpose busses in any region of the automaton.

The command busses are straight busses running in the basic directions through every cell. In the three-dimensional BA there are thus 26 command busses through each cell, one in each direction (positive and negative along the three axes, and along the ten diagonals). Each cell can send (or receive) command signals on links with each neighboring cell (figure 6).

Each cell can normally receive signals via its input links. To send signals on the command busses,
a cell connects with its output links.

We may also want to send signals to a limited area only, so we provide any cell with the capability of interrupting a command bus. This is, if a cell receives a signal to do so it can disconnect a command channel temporarily, so that the command signals propagate as far as this cell but not beyond it along that bus. This interruption will generally though not necessarily last one cycle.

For convenience, we generally use the same label for a signal and for the bus along which it propagates. A "2" signal along bus 3 is thus signal 3.2.

Each cell is distinguished by its FSA state, the conditions, interrupted or normal, of its command busses, and its other bus connections.

Example: Rectangular and related Bus Configurations

As a specific example, (used later on), suppose we need to construct a set of busses within a rectangular area, the interior of which consists of cells in the same state, with boundary cells in (possibly) different states (figure 8).

The case considered will require four steps. The first three steps establish a command structure in boundary cells while the fourth creates the special purpose busses over the rectangle. We give a motivational discussion first with more details later.
8. Desired Configuration

- B.3
- B.5
- B.1
- B.4
- B.2

9. Initial Cells

- Desired Rectangle
  - 2
  - 4

- Active
  - 1
  - 3
To specify the boundary we begin with three activated corner cells (figure 9). The lower left cell (1) sends a pulse down each side of the rectangle activating the other two cells (2, 3) (figure 10). They (step 2) then send pulses along the other sides, the combination activating the fourth corner (4). With all corners activated, the bottom cells (1, 3) send pulses up along two sides (figure 11), putting their cells in a "boundary" state (figure 12). Lastly, to construct the busses the left column of boundary cells sends "bus construction" pulses through the rectangle (figure 13).

In more detail, we want to construct a bus configuration of a type we call "B", as indicated in figure 8. Here the general interior central cells are in state B.1, while the boundary cells are in states B.2-B.5 as shown. The four steps of the construction will use a total of 16 active cell states.

Initially cell 1 is in state B.6, and cells 2, 3 are in B.7. All other cells are in the quiescent state, B.0 (figure 10).

For brevity, we will often name a cell by its state where no confusion is possible between cells in the same state. One cell may thus take on a sequence of names.

As we wish to affect only cells inside the rectangle, we interrupt the command busses at the rectangle borders.
10. Step 1

11. Step 2, Fourth Corner

12. Step 3, Limit Boundaries

13. Step 4

14. Infinite Rectangle

15. Infinite Triangle
We do this by activating the cell in the fourth corner to block propagation beyond it (step 2), we then send a signal to the edge cells interrupting their command busses (step 3).

In step 1 (figure 16), B.6 sends signals on the 1.1 and 3.1 busses to the B.7 cells, changing them to B.8 (1.1 signal), and B.9 (3.1 signal), and interrupting the respective command busses. Cell B.6 then enters state B.10 (figure 10). The B.8 and B.9 cells (step 2) send signals 3.1 and 1.1; this combination is received by the cell in the fourth corner, putting it into state B.7 and interrupting its command busses 1 and 3. At the same time, B.10 now enters B.11, B.8 returns to B.7, and B.9 enters B.12 (figure 11). In step three two things happen. First, B.12 sends a 1.3 signal, putting boundary cells B.0 and B.7 into state B.13 and interrupting the "3" busses through those cells. This thus limits later signals to the rectangle. Second, B.11 emits an activating 1.2 signal (figure 12). The B.0 cells then enter state B.15, the B.7 cell enters state B.16, while B.11 enters state B.14. Cells B.14–B.16 have the job of activating cells in the rectangle. In the final step (4) this row of cells (B.14–B.16) emits three signals, 3.2, 3.3, and 3.4, changing the state of all cells within the rectangle. The 3.2 signal changes lower edge cells to B.2. The 3.3 signal changes
B.0 cells to B.1, and B.13 cells to B.4. The 3.4 signal changes upper edge cells to B.3. The B.15 cells change to B.5. The result is the desired configuration shown in figure 8.

This process is described by the trace diagram explained in section 3.3.

We will also wish to construct bus configurations which are not limited by a rectangular shape. A common alternate shape is an "infinite" rectangle, that is, a rectangle with one or both dimensions extending to the (indefinite) limit of the automaton (figure 14). It can be seen by inspection that such a configuration can be constructed by putting either or both the corner cells (B.7) in state B.0.

Another useful configuration is that of a triangle (figure 15). This can be obtained easily by having the apex cell send signals diagonally along bus 2 rather than bus 1.

It is also straightforward to extend these routines to construct three-dimensional bus configuration.

Varied Rectangular Configuration Construction

Some of the configurations needed will have a variety of interior cell states. The bus configuration in the interior can depend on previous input data. In the present case there is a line of initial cells, I.1, ..., I.k each of which will determine a row of cell
states (figure 16). The construction routine is the same as the foregoing until the fourth step. The cells which were previously B.15 are now affected by 1.1, ..., I.k; their states are now designated by B.I.1, ..., B.I.k. They now emit signals depending on their states, generating states B.1.I.1, ..., B.1.I.k. Each row of interior cells may thus be in a different state.

We illustrate a particular case in figure 17. The "x" and "o" cells cause their row cells to form different channels. The operation is the same as the previous transformation, except that a complicated family of wiggly (parallel) busses is formed. This strategy for bus family construction by parallel processes is fundamental in most later routines.

5.2 Compression Routines

To work with numbers we must represent them in a consistent fashion. We will use stroke notation for now. A number represented by the same number of cells in a specific state in a contiguous straight line.

We have a set of routines whose purpose is to take some pattern of data and reduce it to a number in stroke notation. We call them Compression routines. The first two of these consider patterns where each cell can indicate at most one data point. In later chapters we will introduce patterns where the position of a
16. Initial Configuration
Busses from a Pattern

17. Creation of Busses from a Pattern

result data transfer busses

18. Data set
Transferring Old Data
Past Data Cell

19. Adding
New Data
Element
data point may denote a numeric value greater than one.

5.2.1 Line Compression

The basic routine starts with a row of cells containing data or nothing. We want to eliminate from this row all the "empty cells", and end with the stroke notation for the number of data cells. The same procedure can compact non-numeric data, ending with contiguous data cells.

We wish to transform the row of data cells to another (compacted) form and transfer it to a different location. Since a bus can transfer one atom of data, we can simultaneously transfer a data set of n items with n parallel busses. To see how we transform our data, observe how one of our data cells affects a data transfer of n items.

Above the row containing the data cell we have (n) rows containing the data transfer busses. The relevant element in the transformation of the data set is the line of cells (perpendicular to the data rows) which contains the data cell (figure 18). If the cell is empty we will not change the data set, so the busses run through this line of cells unimpeded (18). If the cell contains information this data must be added to the data set. To do this, the line of cells shifts
the data set up one row, and the new data enters from the bottom (figure 19).

We repeat this process in the lines of cells above each data point. The resulting configuration transfers the data to a line of cells perpendicular to the original row of data cells, and transforms the data so that all "empty" cells are eliminated from the data. An example of this is in figure 20. (Note that this also preserves the order of the data.)

We can use the varied rectangular construction routine to construct the line compression configuration. The actual busses are detailed here. The cells containing a data point construct a line of cells which shift the data up. The busses are R.8 T.3; R.6 T.1 (figure 21). The non-data-point cells pass the data straight so the bus is R.6 T.3 (figure 22).

5.2.2 **Line Expansion**

As a complement to compression, there are times when we wish to expand a data set. That is, we wish to take a data set of \( k \) elements and transform it to a line of \( k+n \) cells so that there are extra (now empty) cells between certain data cells.

To do this we need a pattern detailing how we want to expand the data. The pattern will be a line of cells which contain "0" or "1". "1" means that cell
20. Compressing Data

21. Data Point Busses

22. Non-Data Point Busses

23. Expansion Busses

24. Initial Area Compression Configuration

25. Order of Data Transfer

26. First Row Transfer
can contain a data element and "0" cells will be empty. This pattern may be any configuration, but usually is a regular repeating pattern, for example, every other cell a "1" (01010101...).

Now we create busses to expand (into this pattern) exactly as we did for compression, except that the direction of the busses is reversed. Thus, a configuration to expand data into a pattern where every other cell is a one will be busses as in figure 23.

5.2.3 Area Compression

This routine will compress data from a (NxM) rectangular area to stroke notation. If we first transform this area to a line we can then compress the line with the previous routine. We will construct a bus configuration which will transform an area to a line perpendicular to the data area containing the same number of cells loaded with the same data (figure 24). Note that we are now working with a three-dimensional BA.

We will transfer the data to the goal line in such fashion that the first elements on the goal line are elements from the first row of the data. The next elements on the goal line come from the second row of the data, and so on (figure 25).

The goal line is in the same plane as the first
row of the data area, thus a set of diagonal busses will transfer the first row to the goal line (figure 25). We call this set of busses the diagonal field.

For the higher rows we must not only transfer them diagonally, but we must also transfer them to lower planes. To do this note that in the goal plane (figure 26) only the first diagonal busses are connected to the first row. We will connect all other busses to the next higher plane (figure 27). Now in each higher plane the configuration will be similar. The busses in the higher planes will be described in two parts, the busses which transfer towards the goal line and the busses which transfer down one plane.

In the Ith plane the busses which transfer towards the goal line are the same as in the goal plane. There are diagonal busses which either start in the Ith row or continue busses from the I+1th plane. Now, however these busses do not end in the goal line, but reach a line of cells above the goal line (figure 28).

These busses must then be transferred down one plane. The transfer down is done on the other side of the diagonal field, so the busses travel across the diagonal field, and then down to the next plane (figure 29).

The combination of these busses looks like figure 30.

In effect, the area is transfered line by line
27. Goal Plane Busses

28. I\textsuperscript{th} Plane

29. Transfer to lower plane

30. Complete I\textsuperscript{th} Plane Busses

31. Goal line

32. Cells above goal line

33. Boundary Opposite Goal Line

34. Shift Cells (Diagonal Field)
to the goal row. In the goal plane, while each bus transfers a stroke to the goal row another signal is bussed in from the plane above. Eventually all data in the upper planes is shifted to the goal plane, and then the goal row. In the goal row we use line compression to reduce the data to stroke notation.

These busses can be constructed with a three-dimensional version of the rectangular construction routine. There are four different types of bus cells in this resulting configuration. They are the goal line, the cells in lines above the goal line, the boundary on the opposite side of the diagonal field, and the diagonal field cells.

The goal line only receives signals so its busses are R.1 (figure 31). The cells in lines above the goal line receives a signal along busses from the diagonal field, then sends it towards the opposite boundary, R.1.1 T.1.2 (figure 32). The opposite boundary receives a signal along this bus and sends it down, R.8.2 T.-O. It also receives signals from above, sending them to the diagonal field: R.+O T.8.1 (figure 33). The diagonal field cell actions are shown in figure 34. They shift signals via R.6 T.8.1, R.1.1 T.3, and extend busses R.1.1 T.1.2 (figure 32) to the opposite boundary via R.8.2 T.1.2 (figure 34).
Chapter 6: Arithmetic in Stroke Notation

Other functions which the BA can do immediately (and which we will use later) are the basic arithmetic functions. We demonstrate here the routines which immediately do addition, multiplication, division, and square root.

6.1 Addition

The first function and the most simple to accomplish is the addition function. Beginning with two numbers in stroke notation we wish to end with their sum in stroke notation.

Initially we assume the two stroke numbers we want to add are in a plane, perpendicular to each other (figure 35). To add numbers we concatenate the second set of stroke cells onto the end of the first. Basically we will bus the entire second set of stroke cells in the x direction until it reaches the end of the first stroke cells. Then the busses are sent along diagonals to the stroke row (figure 36). The bus pattern to implement this is in figure 37.

This is of course, the simplest type of addition, where both numbers are in stroke notation. It is
stroke 2
stroke 1

35. Addition Initial Configuration

36. Addition Process

37. Addition Busses

38. Diagonal Bus

39. Interval Marking Busses

R.7 T.2 R.7 T.8 R.1 T.2
R.1 T.8
Middle kth row Goal row Cell

40. Cells for Interval Marking

41. Multiplication, Step 1, Marking the goal Line

quotient
remainder

42. Busses for Multiplication

43. Division of 10 by 3 by 3
possible, if the numbers are in lines of cells but have
gaps in the lines, to do addition as above and then
compact it to stroke notation with the compaction
routine.

6.2 Interval Marking

For both the multiplication and division routines
we have to be able to operate in terms of k cells at
a time (for some k). To do this we need to be able to
select cells on a row at intervals of k, i.e. k, 2k,
3k, ..., nk, ...

The same procedure, with only a change in the
initial cell, marks all cells congruent to b mod k,
b < k.

Observe what happens when we send a bus diagonally
for k cells. The bus will travel k cells in the x-
direction and k cells in the y-direction. Then a
projection of this diagonal bus can indicate the kth
cell in the x-direction (figure 38). If we do this
repeatedly we can "mark" a line at intervals of k cells.

A signal will be sent along a bus diagonally until
it has gone k cells in the y-direction. The signal
then returns directly to the goal row and marks a cell,
i.e. causes this cell to change state. This cell will
be the kth cell. The bus repeats its diagonal path
and then back to the goal row, marking the 2kth cell.
If we continue this process, we mark the goal row every k cells. By hooking each cell in the goal row into the diagonal, the signal needn't stop in each kth cell but continues to mark all the kth cells immediately.

The configuration of busses (figure 39) consists of the middle cells, traversed by diagonal busses and vertical busses, and the cells in the kth row and the goal row which connect the diagonal to the vertical busses. The specific busses which do this are shown in figure 40.

After this configuration is set up, a signal is sent from the first cell of the row. It propagates along the bus and marks the goal row as shown in figure 39.

Once this line is marked at intervals we can use this configuration to create other configurations of busses including the multiplication and division routines. Indeed this routine needs only compaction to achieve division by k.

6.3 Multiplication

The multiplication of p times q is equivalent to adding p to itself q times. We will use this process by marking off a row in intervals of p cells, then sending signals to the first q of these marked cells. The last cell signaled will be the p·qth cell, and
we have the product of $p$ times $q$.

Initially we begin with the two numbers, $p$ and $q$, in adjacent lines as in figure 41. The number $p$ creates the busses for the interval marking, and marks the goal row every $p$th cell (figure 41). This marking is used to create a configuration of busses which does expansion (figure 42). These busses will send signals from the $q$ cells (representing the factor $q$) in the $x$ direction, each "$q$" cell marking a set of cells which is $p$ in length. As the "$q$" cells pass each marked cell, the lowest of the $q$ cells shifts to the goal row, and the rest of the cells shift down one row. After the first marked cell there are $q-1$ cells left. Then after the $k$th marked cell there are $q-k$ (multiplicand) cells left, and the $q$th cell will finally enter the $q$th marked cell, figure 42. Since the marked cells are marked each $p$th cell, the $q$th marked cell is the $p\cdot q$th cell in the line and we have generated the product of $q$ times $p$.

This configuration is the expansion routine, expanding the $q$ data cells by inserting $p-1$ cells before each of the $q$ cells.

6.4 Division

The division algorithm is the reverse of the one above and is very simple. To divide $p$ by $q$ we mark the row containing the $p$ stroke cells every $q$ cells.
44. Least Common Multiplier
Initial Configuration

B.5
B.4
B.3
B.2
B.1

45. Busses for LCM

R.7.1 T.2.1
R.6 T.3 R.7.2 T.2.2 R.1.2 T.8.2
R.1.1 T.2.1 R.1.1 T.8.1 R.7.1 T.8.1
R.1.2 T.2.2 R.1.2 T.8.2 R.7.2 T.2.2

R.7.2 T.8.2

B.1 B.2 B.3 B.4 B.5

46. Individual Buss Cells for LCM

Step 1: A. Sends Pulses
B. Stops 3 bus in p/q cells

Step 2: Signal Command Bus, Turns on Stroke Cells

47. LCM Process
Then all marked cells in the p row are compacted to form the result. We are selecting every qth element of p. The cells remaining to the right of the last marked cell are the remainder (figure 43).

6.5 Least Common Multiple

One interesting number theoretic operation which can be performed immediately is to find the least common multiple of two integers. On sequential computers this would take a varying amount of time depending upon the numbers used, but in the BA, with the use of the interval marking algorithm, the LCM can be found immediately.

The LCM of two numbers, p and q, is the smallest number divisible by both p and q. An interval marking algorithm generates the multiples of a number. That is, it generates those numbers which are divisible by the original number. Thus, the interval marking algorithm can by used to find those numbers divisible by p and those divisible by q. These numbers can be compared to find the LCM.

We first set up busses to do interval marking for p and q simultaneously as follows. Start with p and q in stroke notation perpendicular to the row to be marked (figure 44). (Suppose that p=4 and q=6.) Then the first p rows of cells in the working area contain the busses to mark off the p intervals, and
it also contains part of the busses to mark off $q$ intervals. The remaining $q-p$ lines contain the remainder of the busses to mark $q$ intervals (figure 45). The specific busses to do this are shown in figure 46.

With these busses built, we send two pulses along them, one on the $q$ bus and one on the $p$ bus. As in the interval marking, these pulses will indicate cells on the goal line. Somewhere along that line there will be cells which receive both the $p$ pulse and the $q$ pulse. These cells indicate numbers which are divisible by both $p$ and $q$. Call these $p/q$ cells. To find the LCM we need only locate the first such cell. As the $p/q$ cells are singled out, the command bus in the $3$ direction is turned off. The next step, a signal is sent along the marked line (the $3$ bus). It will propagate until it reaches the first $p/q$ cell where the $3$ bus terminates. As it travels, it turns on all cells up to and including the $p/q$ cell. This number in stroke notation is divisible by $p$ and $q$, and is the least (positive) number with that property, thus it is the least common multiple.

A diagram of the complete process is in figure 47.

6.6 Square Root

To see how we find the root of a number we first examine how to find the square of a number. If we take
a number in stroke notation, let us say \( k \), and construct a square where the sides are \( k \) in length, then there are \( k^2 \) cells in that square. Reversing this, if we take a stroke number and fit the cells into a square, then the length of the side of the square is the square root of the number (after the fraction is dropped). So to find the square root we merely have to have a transformation which will take the stroke cells into a square pattern.

We want to find a configuration which will cover a square. Since we do not know initially how big a square we need, ideally the configuration should start with smaller squares and work up to larger ones.

If the stroke number is one, the square is just one cell. From there we work up to the next larger square by next contacting the perimeter of the one cell. (As it is useful to retain a constant cell as an edge we will expand on two rather than four sides of the perimeter, i.e. in a quadrant rather than the plane.) To expand to a square of side two we send the configuration around the semiperimeter (figure 48). Then we can expand to a square of side three by reaching the next semiperimeter (figure 49). We then continue in this fashion (figure 50).

As we will want to send busses along this configuration, we cannot have a set of disconnected segments, so we will connect these segments. The simplest way to
48. Perimeter of one cell

50. Square covering configuration

49. Perimeter of 2x2 square

51. Path to cover squares

52. Connect with last perimeter

53. Cover perimeter
connect them would be to travel in a clockwise direction along one perimeter, then return along the next in a counter-clockwise direction. This produces figure 51. Unfortunately this configuration, since the perimeters have busses going in different directions (i.e. there is no one general description of the busses on a perimeter), is a little more difficult to construct than previous patterns. So instead we will use a configuration which is simpler to construct, although the configuration itself is more complex. Now we choose to connect all perimeters on the right end, so in each perimeter, our configuration must both return from connecting with the last perimeter (figure 52), and travel over the perimeter (figure 53).

Then the configuration we will use in covering a square is in figure 54. Note that in this configuration the busses on all perimeters are consistent, i.e. they do not change from perimeter to perimeter (other than getting larger). Also this configuration passes through (most) perimeter cells twice, once in covering the perimeter (the clockwise direction), and once in returning from the last perimeter (the counter-clockwise direction).

Now we want to send a set of stroke cells along this configuration with the stroke cells transferring to the perimeter cells. The first stroke cell should go into the first cell on the perimeter, and the nth
stroke cell into the nth perimeter cell. We will essentially send a column of cells traveling along the configuration, and whenever the column passes over a new perimeter cell, the bottom cell "leaves" the column and enters the perimeter cell. This of course means we will use a three-dimensional BA. Also since the configuration passes through some cells twice, we will only "drop" stroke cells into perimeter cells on the clockwise pass.

To determine exactly what the busses are, note there are two distinct phases in the configuration. There are the busses which fill in the new perimeter (figure 53), and the busses which go from the end of one perimeter to the beginning of the next (figure 52). The perimeter-filling busses shift stroke cells down. The return busses do not.

Look at one general perimeter row (figure 54). In it there are five types of bus cells. Two at the ends, two for the general cells on each leg of the perimeter, and the corner cell. There are also two other general cells in the pattern, the cell for the initial corner, and the cells in the bottom plane.

The cells in the bottom plane are identical. Their purpose is to receive a signal from the upper planes and activate their cell (figure 55). The corner cell is a special case. It receives the signal
54. General Perimeter

55. Square cell

56. Initial Corner Cell

57. Type 3

58. Type 4

59. Type 5

60. Type 6

61. Type 7
from the initial stroke cells (R.5), and passes the signal down (T.-O). Then it receives a similar signal from above (R.+O), and finally this signal is passed on (T.3, figure 56). Type 3 cells begin a perimeter row. They receive a signal and pass it to the cell beneath them (r.6 T.-O); then receive a signal from the cell above them and pass that signal on (R.+O T.3, figure 57). The remaining cells all have the same type of busses, with only the direction varying. The next cells on the perimeter (called 4 in figure 54) pass the return bus on (R.3 T.6), transmit the stroke bus down (R.6 T.-O), and pass on the stroke bus from the cell above them (R.+O T.3, figure 58). The rest of the cells are seen in figures 59-61.

We have BA algorithms which will produce the square root to any desired accuracy. These algorithms are linear on the number of digits in the desired result.
Chapter 7: Binary Arithmetic

Until now, we have been using stroke notation almost exclusively to represent numbers because of its simplicity. For example, when adding two numbers in stroke notation, their sum can be obtained merely by concatenating the two numbers. This is not only immediate, but the sum of k numbers at once can be immediate.

There are several serious disadvantages to stroke notation. The strings become very long, even for moderate numbers, with corresponding lengthening of the busses. If the numbers get too large, the bus automaton will not be able to propagate signals for enough in one clock pulse to handle the number (see chapter 10). If numbers are output in stroke notation they are both clumsier and more difficult to read than binary. Also excessive time to communicate a number to a sequential computer would be required. We therefore seek means to use binary numbers in the BA.

7.1 Powers of Two

Since binary is based on powers of two, the first thing we want to produce is the powers of two in stroke
notation.

First note that to double a number, say $k$, we plan to send a signal $k$ cells vertically, then back diagonally. The diagonal return will cross the line containing the original $k$ cells and mark off an additional $k$ along it. The combination of vertical and diagonal turns out to admit iteration which can be "folded" into an immediate process later on.

To do this for any number, note what happens when we begin with a row of cells, and create an ascending diagonal line of cells above this row (figure 62). Then the height of the diagonal above any cell corresponds to the number of that cell. Thus, the diagonal cell is exactly $k$ cells above the $k$th cell. This diagonal begins at the origin and travels away from the result line. There are other diagonals perpendicular to the ascending diagonal which go towards the result line (figure 63). Call them the descending diagonals.

Now to double a number, say 5, start at the 5th cell of the row. If we send a signal vertically to the corresponding ascending diagonal cell, then along the descending diagonal from that cell, it will reach the 10th cell on the result row (figure 64).

To mark a line at intervals of $2^N$, we iterate this doubling process. Call the diagonal the cut-off line. Then there are two sets of busses used in our
62. Diagonal for Doubling
63. Perpendicular to diagonal
64. Doubling 5
65. Busses for Powers of Two
66. Marking a line at $2^n$
67. Creating numbers $2^n$ in stroke notation
process. First there are vertical busses running directly from the result row to the cut-off line. Second, there are busses running along the descending diagonals from the cut-off line to the result line. Each cell of the cut-off line connects the vertical and diagonal busses, and the same in the result line (figure 65).

Now a signal is sent from the first cell. The signal returns to the result line, reaching the second cell. Continuing on, the signal doubles its distance each time it returns to the result line, returning on the 4th, 8th, 16th cell, and so on. In this way the line is marked in the cells $2^N$ (figure 66).

A similar process is one designed to create a set of numbers ($2^N$, $N=1, 2, \ldots, k$) in stroke notation. We execute a process on busses as in figure 65, for the $2^N$ marking operation. In this case however, a different signal is sent such that the original line is not marked off, but rather each vertical bus segment which propagates a signal is marked. Thus the cells above the first cell are marked, above the second cell, the fourth cell, and so on (figure 67). This generates the powers of 2 in stroke notation, and a compaction routine can be used to put the numbers adjacent (figure 68). Using these simple routines, we can generate the powers of two, either in stroke notation or as markings on a line.
7.2 **Stroke to Binary Conversion**

To convert a number in stroke notation to binary we need to find which powers of two will fit into the stroke number.

The process will fit the largest power of two into the stroke number (figure 69). We then have a remainder left into which we try to fit the next smaller power of two. This will either leave a remainder, or it will not fit. The "fits" correspond to ones, the "non-fits" to zeros in the binary expansion sought (figures 70-72).

In the example (figures 69-72) we convert 11 to binary. $2^3$ fits 11 with a remainder of 3. $2^2$ does not fit. $2^1$ fits with a remainder of 1. $2^0$ fits the last remainder exactly. Then the number equals $2^3+2^1+2^0$, or in binary,

$$11_{\text{ten}} = \text{1111111111 stroke = 1011 two.}$$

We want to accomplish this operation immediately, regardless of the length of the number. Initially we don't know which powers of two will fit into the number. In the line which contains the stroke number mark each cell by the greatest power of two (as a stroke number above it) which is a factor of that number (figure 73). That is, if the kth cell is evenly divisible by $2^n$ and not evenly divisible by $2^{n+1}$, then the kth cell's marking is $2^n$. Then all odd cells are marked
68. Powers of 2

69. Fit $2^3 = 8$ into 11

70. Fit $2^2$ into 11-2^3

71. Fit $2^1$ into 11-2^3

72. Fit $2^0$ into 11-2^3-2^1

73. Powers of 2

74. Greatest Power of 2 within 11

75. Remainder after $2^3$
by 1, all even cells not divisible by 4 are marked by 2, and so on. Marking cells above each base cell is to be called a mark.

These marking cells can be constructed simply. An interval marking routine can be used to mark at intervals of $2^k$. Each separate power of 2 is handled in a separate plane, then they are merged into the plane where they are to be used, with a higher power of 2 taking precedence over the lower.

To convert the stroke number to binary, we put the strokes into the base line of cells. The first thing we want to select is the greatest power of two which will fit into the number. Since each cell is marked by a power of two, we select the greatest power of two within that number, say $2^k$ (figure 74). This is the high order digit of the binary number. (In figure 74 we use the number 11 and the power of 2 is $2^3$.) Then we want to find the remaining digits so we consider everything beyond that greatest power of 2. In effect we are subtracting that power of 2 (in our example $2^3$, figure 75). We can then repeat the process on the remaining stroke cells, finding the greatest power of two in the remainder. If we continue the process of subtracting the power of two and finding the greatest power of two in the remainder, we will find those powers of two which make up our number (figure 76).
76. Powers of two within 11

77. Stroke to Binary Busses

78. Stroke to Binary Operation

79. Signaling Binary digits

80. Binary digit bus Connection

81. Binary digits
While this is logically what we are doing, we want to accomplish it immediately. We can do this with the structure we constructed above (figure 73). The top cell of each mark sends a signal right (figure 79). When this signal contacts a mark cell it creates a connection into a bus leading up the mark (figure 80). The next step, the last stroke cell sends a signal along that mark. When this signal reaches the top of a mark, it indicates this is one of the digits in the stroke number (figure 81). These marks are then compressed to form the binary number.

7.3 Binary to Stroke

Once the powers of two have been created, the binary to stroke operation is very simple. First, the powers of two in stroke notation are created, then the binary number is matched to them. If the digit for a particular power is "0", that number is eliminated. Then the powers which remain can be added (using Ch. 6's routine) and the stroke number is the result.

7.4 Binary Addition

In sequential machines (i.e. normal computers) the carry-lookahead adder (CLA) is generally used for addition. Were this not used, addition would take a period of time whose order of magnitude would depend
linearly on the length of the numbers to be added. It is the carry which causes the problem in binary addition. The carry must be generated and propagated for every digit. Fortunately, in a BA the carry can be propagated in two steps independent of the length of the number, i.e. addition is immediate.

In a fashion similar to the CLA we consider the two digits in the kth place of the two numbers. If these digits alone are added, there can be three results: '1', '0', and '10'. Each of these results will have a different effect towards sending a carry digit. '10' will itself generate a carry digit, so this result will be called "GC". The '1' does not generate a carry, but if it receives a carry, it will send a carry on. We call this "SC". The '0' result will not send a carry whether or not it receives one, and is called "NC".

Since SC and GC will always continue a carry if they receive one, any string of SC and GC cells which begins with GC, all propagate carrys. If we do this with busses, GC starts sending a signal, SC continues a bus, and NC terminates any bus (figure 82).

The addition operation is composed of three steps. First, the intermediate result cells are created (GC, SC, and NC). Second, bus signals determine carry for all cells. Third, the sum of the carry and the
82. Intermediate cells for carry

83. Intermediate carry

84. Binary Sum

85. Generate Carry

86. Binary Multiplication
intermediate cells generates the sum. The sum for each digit is determined by Table 1, below.

**Table 1 Sum of carry and intermediate**

<table>
<thead>
<tr>
<th>Carry</th>
<th>Intermediate</th>
<th>GC</th>
<th>SC</th>
<th>NC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

An example of this is the addition of $1101101_2$ to $10001011_2$. In figure 83 the intermediate cells are created. Figure 84 generates the carry for all cells. 85 adds the carry to intermediate results and generates the sum.

Adding multiple summands can also be done immediately. As this is needed in binary multiplication (with shifting) we discuss it in the next section.

7.5 **Binary Multiplication**

Binary multiplication poses problems over and above addition in that there may be more than one digit of carry.

Basically, as normal multiplication, the multiplicand is multiplied by each digit of the multiplier. Each of these results, appropriately shifted, is then added together (figure 85). Since each column of the sum
will contain only 0's and 1's, we are basically adding in stroke notation. Then the addition for each column is simple and obviously immediate. The difficulty comes when we have to carry. Binary addition had at most one digit to carry, while multiplication has the potential of many digits of carry. However, this problem can be resolved.

The problem is as follows: Given a sum in a column, find how to immediately add on the carry and generate the next carry.

The carry in base two is the sum of the column (call this the column sum), plus the carry, (call this sum the total column sum) divided by two and rounded down. Since we know that in stroke notation, addition and division are immediate, we can combine them. The result is in figure 87. A set of busses first adds in the carry, then connects to another set of busses which divide by two, generating the new carry. The carry addition busses are the same no matter how large the carry is, so they needn't change during the process. Similarly the busses for division by two remain constant no matter what number is being divided so they needn't change either. Then all we need do is connect the carry-out busses of each column to the carry-in busses of the next. Then we have one set of continuous busses which will propagate the carry as far as is necessary.
88. Basic Structure

87. Bus Pattern for Carry

89. Addition Busses

90. Division by 2 Busses
Consider the plane BA as becoming the base plane of a three-dimensional BA. To examine the carry in more detail, observe a plane perpendicular to the base plane, containing one column of the multiplication. This gives us an area to work in, to handle that column. We assume there are incoming signals on busses which send carries into this column. Our goal is to generate signals on the outgoing busses which provide carries to the next column (figure 88).

To add the carry-in to the column sum we needn't know it beforehand. Busses are constructed which add the carry-in signals to the column sum (figure 79).

To divide by two, the busses are always the same, so we construct them to take data from the column sum (figure 90). Now if we connect the busses determining the total column sum with the dividing busses we have a continuous set of busses from the carry-in to the carry-out.

If we connect the carry-out busses of each plane with the carry-in busses of the next (figure 91) the result is a spiraling structure of busses which will propagate the carry from the first to last column of the sum if necessary.

Notice that the result for each digit will be the remainder of the division by two. If the total column
sum is even, all of it will become the carry, and the
digit for that column is 0. To find this, the last digit
signaled in the total column sum becomes a zero if it
is an even digit and one if it is odd. Then for the
produce we use the last entry in each column (see Table 2).

Table 2 Multiplying 110101\text{two} by 11101\text{two}

<table>
<thead>
<tr>
<th>Column Sum</th>
<th>1 2 2 2 3 1 2 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carry-in</td>
<td>1 2 2 2 2 1 1 0 0 0</td>
</tr>
<tr>
<td>Total Column Sum</td>
<td>1 3 4 4 4 4 2 2 0 1</td>
</tr>
<tr>
<td>Result</td>
<td>1 1 0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>

To complete this routine, we need to construct the
process for determining the column sums. This process
is in two parts. First we multiply the multiplicand by
each digit of the multiplier, then we add each column
for the column sum.

Each column contains only 0's and 1's so we need
only compress the zero's out with a compression
algorithm to get the column sum. To multiply by the
digits of the multiplier, we make copies of the
multiplicand, shifting each copy one column (figures
86, 92), then multiply each copy by a digit of the
multiplier. Thus the first copy is shifted once
and is multiplied by the first digit of the multiplier,
and the second copy is shifted twice and multiplied by
91. Carry busses for column k

92. Shifted Multiplicand

93. Shifted copies of multiplicand

94. Busses for Multiplicand shift

95. Busses for Multiplier
the second digit of the multiplier, and so on. The shifting can be done by diagonal busses (figure 94). We reference each line with a bus from a digit of the multiplier (figure 95). Then to get the result, each cell has a diagonal bus from a digit of the multiplicand, and a horizontal bus from a digit of the multiplier, so each cell which gets a '1' signal on both busses is a 1, all others are zero (figure 86).

Once we have this, we construct the carry bus pattern. Then signals from the column sums will create the product.

Addition of multiple summands is obviously similar. It can be viewed as unshifted "multiplication" where each row has a different summand instead of copies of the same shifted number.
Chapter 8: Linear Pattern Processing

There are many immediate routines which are related to linear patterns. While the problems themselves are different, the methods of solution for many problems are quite similar. We have already seen that a pattern of busses used for the compression routine can be changed but slightly and used for multiplication. Many of the linear pattern routines in this chapter demonstrate these similarities.

8.1 Line Code Determination

We wish to calculate the Rothstein code of a line whose slope is p/q.

Knowing the slope, we may make several obvious observations. When a line with slope p/q has gone one unit horizontally, it will have gone p/qth of a unit vertically. When it has gone two units horizontally, it will be 2p/q units vertically, and so on. Also when the line has gone one unit vertically, it will have gone q/p units horizontally.

Now note that when the line is graphed (starting at the origin) for each k where kp/q is greater than a
unit, and \((k-1)p/q\) is less than that unit, this means that the line passed a vertical unit in the \(k\)th cell. In BA terms, it has crossed vertically into another cell. Also, it has done this in the \(k\)th (horizontal) cell. Then the "1" digits of the code are in the \(k_1, k_2, \ldots\) cells where \(k_i\) is defined by

\[
k_i \cdot p/q = (k_i-1) \cdot p/q.
\]

Rothstein and Weiman (1976) calculated the line code using two shift registers of length \(p\) and \(q\). The registers shift cell by cell simultaneously. As the \(p\) register completes each cycle, it indicates a horizontal move of one cell. (Note that after the first \(p\) cycle, the \(q\) register has completed \(p/q\)th of its cycle. This indicates that the line has gone \(p/q\)th of a unit vertically.) Then whenever the \(q\) cycle ended indicates a vertical move of one cell.

We can duplicate this process instantly using an interval marking algorithm. Note that if a line is marked at an interval of \(p\) cells, those cells could indicate the cycle of a \(p\)-length shift register. Then to show two registers, we mark the line at intervals of \(p\) and \(q\) (figure 96). The end of each \(p\) cycle, or the \(p\) marking, indicates a horizontal shift, which we will call for now a '0', and the \(q\) marking is a vertical shift or a '1'. Whenever we have both a \(p\) and \(q\) marking indicates a vertical and horizontal shift, and this
96. Line Code Determination

97. '1' row adds a '1' bus and absorbs a '0' bus

98. Code Compression

99. Generates line (p=3, q=5)

100. Run Cell

101. Rise Cell

102. Types of cells in a line pattern

103. Determining Code of Pattern
is the end of one cycle of the code.

To convert this to the line code, note that a '1' in the line code denotes a vertical and horizontal shift, while our '1' is only a vertical shift. Thus we must combine each of our '1's with a '0' after it, so we create a special compression routine which will delete a 0 for each 1 (figure 97). Once our code is thus compressed, it becomes the straight line code for a line of slope p/q.

Observe that the bus pattern is identical with that of the LCM routine.

The busses to determine the code of a line of slope 3/5 are shown in figure 98, and the generated line in figure 99.

8.2 Line Creation

Once we have the line code we can use it to create the line itself on the BA.

Each digit of the code determines the type of transition of the line through that row. However, the kth digit of the code does not in and of itself determine the cells that line runs through, rather the cells are determined by the kth and all preceding cells. We can create a configuration of busses which will generate the line from the code. In each row all cells create busses connecting themselves with cells in the
adjacent rows, according to that row's code digit. (If the digit is 0 then no rise is indicated so cells in the same row are connected, figure 100. If the digit is 1 then cells are connected so that the bus "rises" one cell, figure 101.)

Once this bus configuration is constructed, the origin cell of the line need only send a signal on these busses to signal the entire line.

8.3 **Line Recognition**

The complementary problem to line determination is the problem of recognising whether a pattern of cells actually is a line. If we have the pattern representing a line we can determine whether this represents (a complete segment of) the line's code.

There are two parts to this process. First we can find the end points of the pattern and using the line determination routine, find the code of the line which passes through those points. Then we can determine the code of the pattern itself and compare the two. If the codes match, the pattern is a line.

We already know how to determine the code from end-points. To determine the code of a pattern, each cell in the pattern examines its neighbors. As shown in chapter 2, each cell can be one of three types. It can be a run cell, or one of the two rise cells
(figure 102). If the cell has only neighbors to the right and left it is a run cell, and indicates a '0' in the code. If a cell has a neighbor above it, it is a rise cell and indicates a '1' in the code. If a cell has a neighbor below it, it is a rise cell, but the cell below it will indicate the code.

When each cell has determined its own state it sends a signal to the goal line showing its state. Then the goal line contains a copy of the code. This process is illustrated in figure 103.

Note that this process will only determine whether or not a pattern is a line segment which contains a complete copy of its code. If the pattern is a line but is not the complete copy of the code, the process will not necessarily work, but the line will fall within a wedge the size of which is determined by the scale of the BA (Rothstein and Weiman, 1976).

8.4 Average

The previous data compression routines treated the data as single points. That is, each cell could either be empty, or it could represent a data point. In some patterns we work with, a single cell can represent different values. For instance, if we have a pattern representing (statistically) a straight line, the position of the data point is as important as the fact that it
exists. This routine and the following routines will handle data of this type.

One of the first statistical functions is the average. Given k data points, $x_1, \ldots, x_k$, we want to find their sum divided by $k$. The sum of the $x$ coordinates is the first function we will examine. In later sections we will look at the sum of the squares of the $x$ coordinates, and then at the sum of the $x_i y_i$ products. (That is, the sum of the $x$ coordinate times the $y$ coordinate.)

We have a set of data points within some rectangle and we want to find the sum of their $x$ coordinates (figure 104). Each cell is worth a varying numeric value depending on which line it is in. We already have a routine which could compress the data if each data point had the same value. If we could construct a transformation which would change the data so that each cell in the $k$th line became $k$ cells, then we would have the sum of the $x$ coordinates and could compress it.

Let us first examine the lines separately. Since we use three dimensions, we can examine the $y$-$z$ plane which contains each separate line and work with each line in its own plane (figure 105). Now we can construct a compression routine in this plane to determine how many data points are in this line (figure 106).

Each separate plane represents a different value of $x$. We want to multiply the number of cells in each row by
104. Data Representing a line

105. y-z plane containing 3rd x-row

106. Compression, the number of points in the row

107. Summing: \( \sum x_i \), (this plane is \( x=3 \))

108. Data in x-z plane where \( y=3 \) (from figure 104)

109. Multiplication Busses for \( x=3 \)

110. \( x^2 \) (\( x=3 \))
the x value of that row. To determine the x value of each plane we send a diagonal bus (from the origin) along the planes. When it has reached the ith plane it will have a height of i cells. Using this as a starting point we can create busses to do a multiplication by i. After we compact the data points in the line, multiply them by i.

The overall busses to do this are shown in figure 107. This process is repeated for each row of data. Once this transformation is made, each cell represents one unit. Then we may use the area compaction routine to reduce the data to stroke notation.

An average can also be obtained from this routine. We have the sum of the x coordinates, and the number of data points is also immediately obtainable. Dividing the sum of the x's by the number of data points we have the average value of the $x_i$'s.

8.5 Variance

To compute a variance, we sum the squared deviation from the mean. Variance $= \frac{1}{N} \sum (x_i - \bar{x})^2$. We can extract the mean from the summing expression, which then becomes $\frac{1}{N} \sum (x_i)^2 - \bar{x}^2$.

The mean can easily be found with the previous routine, and both addition and multiplication are immediate, so the only thing we need, to compute variance, is the sum of the square of the x coordinates.
As before, each data point has a different value depending on its x coordinate. Again, all the data cells in each line have the same value. Separate the working area into separate planes for each value of x. We will compress the data for each line to find the number of data cells in that line. Now we want to multiply this by the squared value of x for this line.

Sending a signal on diagonal bus to each plane will give the plane its x value. In our plane let x=k (k=3 in figure 108). First we can obtain the multiplication busses for multiplication by k (figure 109). Now we send the number k (in stroke notation) through these busses, obtaining x squared (figure 110). Now with the number k² in stroke notation, we can obtain the busses for multiplication by k² (figures 111-112).

Having constructed busses for multiplying by k², we multiply this by the compressed data. The result is the sum of the x₁² for one value of x.

We will now describe the process step by step, and diagram it for the plane where x=2.

Step 1: A diagonal bus sends signals through the planes, giving each its x-value (figure 113).

Step 2: Multiplication busses are constructed for x (figure 114).

Step 3: Multiply x by x (figure 115).

Step 4: Duplicate x² for construction of multiplication Busses (figure 116).
86

111. Sending $x^2$

112. Busses for Multiplication by $x^2$
   (x=3)

113. Plane receives its x value

114. Multiplication busses for x=2

115. $x^2$

116. Duplicate $x^2$

117. Busses for Multiplication and Compression

118. Multiply data by $2^2$
Step 5: Construct busses for multiplication by $x^2$. Construct compression busses (figure 117).

Step 6: Compress data and multiply by $x^2$ (figure 118).

If we do this process for all lines, the data is transformed such that a data point with coordinate $x_i$ becomes $x_i^2$ cells. Then we have the sum of the squared $x$ coordinates in an area. This area is then compressed and we obtain the sum of $x_i^2$ in stroke notation.

8.6 **Sum of $x_i y_i$**

Another part of the regression function is finding the sum of each $x_i y_i$ product. The last two routines were relatively simple to solve as all the cells in one row had the same value, and thus there was a 2-dimensional plane to operate in for each row. In this case each cell has a separate value and thus there is not a separate plane in which to handle each cell.

The (sum of coordinates) routine, while it did not actually move the data, treated it such that all cells which had the same value (i.e. $x$-coordinate) were treated by the same set of busses, separately from the rest of the data. This is conceptually separating cells which have the same value.

This cannot be done the same way to separate the $x$-$y$ cells. Since each cell has a different value from its neighbors, the greatest unique region each cell
119. Sawtooth producing $1 \text{ Mod } N$
would have would be a single column of cells, rather than a plane of cells as in the last routines. A single column of cells is not enough to operate with.

Observe what happens when we use the first part of the area compression routine on this data. Let each cell be \( d_{ij} \). We transform the segment of the x-y plane to a line. The first elements in the line are from the first row. The next elements are from the second row, and their order in that row is unchanged. The third and further rows behave similarly, thus the line of data points is

\[
\begin{align*}
  d_{11} & d_{12} & d_{13} & \cdots \ & d_{1n} & d_{21} & d_{22} & \cdots & d_{2n} & d_{31} & \cdots & d_{nm}.
\end{align*}
\]

Thus the kth element of the line comes from the cell \((1 + k/n), (k \mod n)\). So the x multiplier of the first n cells is 1, the second n cells is 2 and so on. The y multiplier of each cell is the cell's ordinal mod \( k \).

This can be produced by a "sawtooth" configuration (figure 119) which is the same as that produced by the interval marking algorithm (figure 39).

We now only need to multiply by the x-multiplier, then by the y-multiplier, and use area compression to find the \( x_i y_i \) sum. Each of these operations is immediate, so the whole process is immediate.
Chapter 9: Immediate Straight Line Fitting

This chapter will conclude our investigation of straight line processing. We will construct a routine which will do the orthogonal regression.

First, we review the subprocesses necessary to the orthogonal regression. Next we show how two routines can be joined together so that they operate in sequence. Finally we construct a routine, which will fit a line to the data, by joining together 12 subroutines.

This final routine performs the orthogonal regression in immediate time while sequential computers take $O(n)$ time.

9.1 Straight Line Fit

As shown in chapter 4, the MLE of the line is to be determined by the orthogonal regression slope and the centroid. The centroid is nothing more than two averages, $(\bar{x}, \bar{y})$, so it can be computed immediately.

The equation determining the slope is again

$$\beta = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2 - \sum(y_i - \bar{y})^2 + \sum(x_i - \bar{x})^2 - \sum(y_i - \bar{y})^2 + 4 \sum(x_i - \bar{x})(y_i - \bar{y})^2}$$

The mean terms can be extracted from the three
different summations giving

\[ T_1 = \sum (x_i - \bar{x})^2 = \sum x_i^2 - N \cdot \bar{x}^2, \]
\[ T_2 = \sum (y_i - \bar{y})^2 = \sum y_i^2 - N \cdot \bar{y}^2, \]
\[ T_3 = \sum (x_i - \bar{x})(y_i - \bar{y}) = \sum x_i y_i - N \cdot \bar{x} \cdot \bar{y}, \]

and the slope is then

\[ \beta = \frac{T_3}{(T_1 - T_2 + 4T_3)^2}. \]

We have already produced in chapter 6 routines to do the addition, multiplication, division, and square root of terms, and chapter 8 contains routines for mean, variance, and the sum of \( x_i y_i \). Thus we have routines for each part of the straight line routine, and we only need to combine these routines.

9.2 Joining Two Routines

While many of the routines for the orthogonal regression can be done simultaneously, certain routines must be done sequentially (at the moment); for example, the square root above cannot be done until the sum \( T_1 - T_2 + 4T_3 \) is computed. Thus, we must be able to "join" two routines, \( r_1, r_2 \), so that when \( r_1 \) is finished, \( r_2 \) starts executing.

Moreover, we wish to join two immediate routines such that their (joined) result is computed immediately.

Every immediate routine, \( R \), starts with cells in initial states, \( r_1, r_2, \ldots, r_n \). Also each immediate routine will be completed within a fixed finite number
of cycles. For our routine let this number be represented by \( \text{time}(R) \).

When all the cells are in states \( r_1, \ldots, r_n \) the routine will execute properly. If we intend joining this after another routine we do not want it to begin executing until the first routine is done.

To accomplish this, we change the initial elements of this routine. Each of the cells was put into its state at some previous time. We will alter the initial creation of these states so that the cells are put into states \( Pr_1, \ldots, Pr_n \). Now we add a control cell in state \( Pr_0 \), and connect this to all the cells \( Pr_1 \) via a system of busses. The job of the control cell is (at the proper time) to send a signal to all the cells, \( Pr_1 \). When each cell receives the signal it changes from state \( Pr_1 \) to \( r_1 \). Then all the cells \( r_1, \ldots, r_n \) are present and the routine proceeds normally.

To complete the join of routine \( R \), we connect the control cell with the previous routine such that when the previous routine is completed, it sends the "starting" signal to the control cell. This signal changes \( Pr_0 \) to another state \( c_0 \). State \( c_0 \) then sends the signals to all the states \( Pr_1, \ldots, Pr_n \).

If we want to join this routine before another routine, this routine must send a signal to the next routine. This is also handled by the control cell.
The control cell goes through a series of states $c_0, c_1, \ldots, c_{\text{time}(R)}$. Each state $c_i$ transforms in the next step into state $c_{i+1}$ ($i < \text{time}(R)$). When state $c_{\text{time}(R)}$ is reached, the routine is by definition finished, so state $c_{\text{time}(R)}$ sends a signal to the control cell of the next routine.

Thus to join two routines, we transform them as detailed above, plane them in separate areas of the BA and connect their control cells. The operation of one routine will not interfere with the operation of the other. The first routine ends in step $\text{time}(r_1)$. At $\text{time}(r_1)+1$, $r_2$'s control cell sends signals to all of $r_2$. In step $\text{time}(r_1)+2$, routine $r_2$'s cells are all in the initial states and $r_2$ begins. At $\text{time}(r_1)+\text{time}(r_2)+1$, routine 2 ends normally. The join of these two routines is done in a fixed number of steps so the join of these routines is also immediate.

9.3 Straight Line Routine

With the joining process we can combine routines to solve the straight line problem as follows.

1. $\bar{x}$
2. $\bar{y}$
3. $\sum x_i^2$
4. $\sum y_i^2$
5. $\sum x_i y_i$
6. $\sum x_i y_i - N \cdot \bar{x} \bar{y}$
7. $\sum x_i^2 - N \cdot \bar{x}^2$
8. $\sum y_i^2 - N \cdot \bar{y}^2$
9. $\sum (x_i - \bar{x})^2 - \sum (y_i - \bar{y})^2$
10. $\sum (x_i - \bar{x})^2 - \sum (y_i - \bar{y})^2 + \sum ((x_i - \bar{x})(y_i - \bar{y}))$

(Note that the line we want has a slope $p/q$ specified by $p=\#6$, $q=\#10$.)

11. Compute code of line with slope determined by $p$, $q$.

12. Plot line with slope $p/q$ through point $(\bar{x}, \bar{y})$.

Now we have a routine which is the combination of 12 immediate routines and so is itself immediate.

On a sequential computer several of the above steps take $O(N)$ time, so that the BA is faster.

Note that not all of the 12 routines must be executed sequentially. The first five sub-routines, for example, could be computed in parallel. This would not change the time of execution's order of magnitude.

It is straightforward to determine that all these routines can fit in the BA at once.
Chapter 10: Conclusion

The original goal of this research was to perform statistical pattern recognition in parallel, as exemplified by the problem of fitting the statistically best straight line to a given set of data points. This was accomplished, but the computational problems involved turned out to be so interesting and important that they filled most of the research period. Indeed, these parallel computation results are more important than the original problem. They have demonstrated that many important computations are much faster on a BA than on sequential computers.

10.1 Summary

In chapter five we showed that needed configurations of busses could be constructed immediately. This is important since it doesn't matter how fast a bus pattern can solve problems if the patterns take more time to construct than the conventional problem solving takes.

With this capability we can construct many routines which will solve given problems for a variety of input data. While this is not the same as a general purpose
programming language, it goes a long ways towards making the BA general in the sense that many programs are handled "quasi-architecturally" on one machine (Rothstein, 1978). This is very much in the spirit of the BA called JR (Rothstein, 1978).

Also, in chapter five we present several specific data manipulation routines. The line compression routine shows how a string of elements can be shortened by eliminating a designed class of elements of the string. (One example is eliminating blanks in a character string.) For a string of length n, the BA does this immediately where a sequential computer takes \( O(n) \) steps. It is easy to see that many other string manipulation functions can also be done immediately.

Chapter six performs arithmetic in stroke notation. Addition, multiplication, and division of integers are all done immediately.

Since stroke notation requires many cells, we also investigate positional base systems, particularly binary. Addition, subtraction, and multiplication are done immediately. Conversion is immediate from stroke to binary, and binary to stroke. (This easily generalizes to convert any number base to stroke, hence from any number base to any other base.) The ability to handle binary not only saves space, but also simplifies any possible communication with conventional computers.
Chapter eight solves several problems relating to straight lines. We show how to recognize and construct straight lines and their codes (immediately). Also there are several specific routines to compute statistical functions on data, relating to mean, variance, and covariance.

Finally chapter nine applies these routines to solve the problem of fitting a straight line to data. Again we find that a problem which would take $O(n)$ time on a sequential computer can be done in $O(1)$ (i.e. immediate) time on the BA.

This research has expanded on previous research and has gotten one step closer to a general purpose BA. The history of BA research has proceeded in three steps thus far. Early research by Rothstein (1976, August 1977, September 1977, and Rothstein and Weiman, 1976) originated the bus concept. Moshell and Rothstein (1979) formalized the bus concept and investigated its potential with respect to formal languages. This work developed the method by which a BA could produce general configurations of busses (chapter 5) and produced immediate BA's to solve many problems, both basic (arithmetic) and combined (e.g. statistical line recognition). Although there are many other areas to be investigated (e.g. Rothstein and Davis, 1979), a logical next step would be to develop a general purpose BA.
Rothstein and Moshell's work investigated the results of specific bus configurations. This work produced BA's which could solve general types of problems. The next level should produce a general BA which could be "programmed" much as general purpose digital computers are programmed. This requires the development of a general BA cell which could handle any problem. Most of the states of this cell might be taken from the states of routines in this work, however not all routines have been investigated here so more states may be necessary. All states must be available in one cell, but many can no doubt be merged.

This also suggests the development of a programming language for BA's. The eventual goal of the next step is a high level, parallel programming system. The initial language would be closer to an assembler, whose most useful features would include automatic relocation and joining of routines.

10.2 Characteristics of Non-Immediate Problems

Since most of this research has been constructive in nature, the class of non-immediate problems has not been investigated. It is illuminative to observe some distinguishing characteristics of non-immediate algorithms. By considering non-immediate characteristics we can gain insights on necessary techniques for
constructing immediate algorithms.

An algorithm is immediate if it takes $O(1)$ steps to complete. By way of example, consider an algorithm which has $n$ inputs. If the inputs are independent, and the results of each are not dependent on partial results after other inputs, then we essentially have $n$ separate problems which can be solved separately and simultaneously in different parts of the BA. Thus if we can handle each separate input immediately the result is immediate.

The case is different if the inputs must be handled sequentially. That is, if the first $j$ inputs must be handled before the $(j+1)^{th}$ input can be handled. In this case, if each separate input takes even one step the problem is linear. We now have $n$ routines (possibly identical) to handle the $n$ inputs. To make the result immediate, each of these routines must take less than one step to complete. This implies that there can be no state change in these routines. The only way we can accomplish anything in less than a state change is to pass an input through a bus configuration.

Most routines consist of constructing a bus configuration and passing data through this configuration. If we have to do this sequentially then we have to construct $n$ configurations (each of which takes more than one step), so it would be non-immediate. To solve this we have to construct all the configurations simultaneously.
Then the result from the first $i$ configurations is sent through the $(i+1)^{th}$ configuration. More explicitly, we connect the outputs of the $i^{th}$ configuration with the inputs of the $(i+1)^{th}$ configuration by busses so the only delay between routines is propagation time.

One example of this is that part of the binary multiplication routine which is equivalent to the addition of $n$ binary numbers. To solve this we initially create configurations for each column which add the incoming carry to this column's bits and computes the next carry. Thus, in one step the signals propagate through all the connected configurations.

Another way of characterizing the immediacy constraint is saying we have to avoid too many decisions in the middle of the routine. We have to construct an algorithm such that the process for each input cannot contain a decision based on that input and the results from the previous inputs. If such a decision is necessary then a state change at that point is necessary which, as we have seen, makes the routine non-immediate.

The difficulty of determining whether a problem is immediate is that the above constraint is dependent on the particular algorithm used. It is not always clear whether a faster algorithm is possible.

Consider the following problem handling a bank account. We assume $n$ inputs where each input contains
a code symbol (indicating deposit or withdrawal), and a dollar amount.

To handle each input a decision has to be made whether to add or subtract the amount from the balance. On first inspection it might appear that since a decision has to be made for each input the routine was non-immediate. A minor rearrangement of the algorithm makes it clear that the decisions are based only on the code symbol. These are all available at the start of the algorithm, so the decisions can all be made simultaneously. In this manner the problem can be solved immediately, but overdraft is similar to overflow, see below.

On the other hand, consider the problem of handling a k-place queue. There are n inputs, each of which is either an enqueue or a dequeue. To handle each input we must first decide whether add an element to the queue or remove an element from the queue. This can be done simultaneously for each input. Then the decision must be made as to whether an underflow or overflow condition exists (e.g. adding the k+i th element to the k-place queue). To make this decision we need to know the results from the previous input. Although I cannot at this time prove this problem is non-immediate, the decision criterion suggests this is the case.

From this discussion we see that to make an n-input problem immediate we must avoid the necessity for
\[ N_{i+1} = N_i + K_{i+1} \]

120. Stroke Addition
decisions between inputs. Thus each input must either be handled with a single bus configuration, or all input must be handled simultaneously.

10.3 Bus Automaton Speed

A cellular automaton, and thus a BA, can solve any problem a conventional computer can. The BA can always solve problems at least as fast as a sequential computer. There are also many routines which are orders of magnitude faster on the BA. Those discussed in the dissertation include line compression, least common multiple, and the statistical routines.

Some immediate routines can be combined, apparently sequentially, to give an immediate resultant routine. For example, generalize to addition of m numbers, \( k_1, k_2, \ldots, k_m \). Let \( N_i = k_j \). Then \( N_{i+1} = N_i + k_{i+1} \), and the problem is to compute \( N_m \) immediately. If we specify that the \( k_i \) are input on busses and the sum \( N_i + k_{i+1} \) is output on another set of busses, then the bus configuration to accomplish this is in figure 120. With this configuration, if we connect the input busses of one routine to the output busses of another there is no delay between them, and in one propagation cycle we can complete all additions.

In a similar manner we can combine different immediate arithmetic functions. For example, observe the
binary multiplication routine contains a series of alternating additions and halvings and is immediate.

In a similar fashion an arithmetic expression can be done in $O(1)$ time (in stroke notation). With a parallel combination of sequential computers, an arithmetic expression takes at least $\log(n)$ time (Kuck, 1977), and the least upper bound makes the computation time, for any number of sequential processors, $2.88 \log(n)$ (Muller and Preparata, 1976; Kuck and Marakoa, 1974; Kuck and Maruyama, 1975). Thus a BA can evaluate arithmetic expressions faster than any combination of sequential computers.

It may also be mentioned in passing that a sort may be done immediately, while sequential computers take $O(\log n)$ time (Batcher, 1968; Preparata, 1977). The immediate sort routines are detailed in appendix I.

When we say a routine is immediate on a BA we have implicitly made two assumptions. First, we have assumed the BA always has enough cells to perform each routine (no overflow), and second, that propagation time can be neglected, i.e. the busses are not long enough to make propagation time greater than the cycle time.

Since each cell is a FSA, many can be inexpensively housed on a chip. Overflow will surely occur for large enough problems in complete analogy to shift register or memory overflow in conventional computers. As cells are
both computing devices and storage units, the solution to the BA overflow problem is essentially the conventional one. One must either cascade more cells or restrict the size of the problem.

A computation problem in stroke notation can be large while its binary version is small, but this can often be ameliorated by immediate translation between notations.

Let us see what failure to propagate remotely within one cycle time means in terms of a specific routine. Examine the binary addition of two N digit numbers. A signal has to propagate approximately N cells in less than one cycle. So to add a number in the billions (30 binary digits) the propagation time must satisfy \( T_p \cdot 30 < T_s \). It is obvious that no matter what the ratio of \( T_p \) to \( T_s \), there will be some number of digits for a number above which propagation "failure" occurs. However, normal computers are also designed so that in one cycle they can only add numbers up to a certain size. Again we have similarity in the restrictions on BAs and conventional computers. For non-immediate problems on the BA, there may be much communication between cells in some region with corresponding growth in propagation time. How bad this problem becomes depends on the complexity of the problem involved, but little can be said, as yet, about such cases (Rothstein, 1976).
In the immediate case the BA is obviously much faster than the conventional computer for problems which are not too "large". For very large problems, the BA accomplishes the following: It effectively replaces the single cell by the largest BA block for which the concept "immediate" retains its usefulness. These "supercells" can then be considered as cells of a cellular automaton whose $T_S$ is given as $T$ above (section 2.2). There will still be advantages of the BA over the conventional computer, because the unit of computation, so to speak is a big fast one. This "super" cellular computer is a modular computer network, to which conventional parallel techniques apply as well as many BA procedures.
Appendix A: Parallel Sorting

A.1 Sorting a set of Stroke Numbers

We wish to sort a set of numbers in stroke notation. Assume the numbers are next to each other (figure 121). To sort them we compress each row of digits to the right. That is, look at a row containing the kth cell of each stroke number. (If the particular stroke number is less than k, that cell of the row is empty.) Compress all empty cells out of this row. If we do this to all the rows, the result will be as in figure 122.

Observe that 122 is the stroke numbers in sorted order. To prove that this routine will sort numbers observe the following: Let each number be $S_i$ and each stroke in $S_i$ be $s_{ij}$ ($j=1$ to $S_i$). Define cells as

$$c_{ij} = \begin{cases} 1 & \text{if } S_i \not= j, \\ 0 & \text{otherwise.} \end{cases}$$

Then $c_{ij}$ is one if the cell is a stroke. Each row has $\sum_j c_{ij}$ stroke cells in it so the compacted row will also have $\sum_j c_{ij}$ cells. Now if the kth compacted line contains the kth greatest number, we will have sorted the numbers. Let $H_k = S_m$ (for some $m$) be the kth greatest number. There are (no more than) $k-1$ numbers greater than the kth number. Then there are less than
k numbers who have height greater than $H_k$. This means $\sum_i c_{ij} < k$ for $j > H_k$. Then in the kth compacted line stroke cells greater than $H_k$ are blank. Since there are at least $H_k$ cells in the kth row, the $H_k$th cell in the kth line is a stroke and all cells before $H_k$ are strokes. Thus the kth line has exactly $H_k$ cells, and we have sorted the set of numbers.

Essentially this routine "shoves" the numbers over, so that the higher numbers take precedence over smaller. Note that the strokes may not be the same in the compacted number, but each stroke is identical so it makes no difference.

A.2 Sorting Digits

To sort a list of keys where each key is not a stroke number requires an extension of the above algorithm. We will generate a number (in stroke notation) for each key which will be a numeric measure of how many other keys are less than or equal to this key. Then these numbers can be sorted by the above algorithm and the keys will also be sorted.

Assume we are sorting a set of keys $k_1, \ldots, k_n$, and that there is a strict ordering on the keys. We create a matrix which matches each key with every other key (figure 123). There will be two lists of the keys along the edges of the matrix and busses connect the
121. Numbers to be Sorted

122. Compacting to Sort

123. Sorting Digits
lists with the interior so that an interior cell, \((i,j)\) gets signals from keys \(k_i\) and \(k_j\). The cell \((i,j)\) then becomes a "1" if \(k_i \leq k_j\). Thus, the \(i\)th row will have a number of "1" cells showing how many keys are less than or equal to key \(i\). (If key \(i\) is the greatest, then every cell is "1".)

If the cells in each row are compacted, we will have, for each key, a stroke number indicating how great it is. Then the stroke algorithm can be used on these numbers to put the keys in order.

Sorting strings of digits (base \(k\) numbers) or general strings of characters can be handled in similar fashion.

A.3 String Sort

The general sorting problem is to put a set of strings in order. The strings, \(s_1, \ldots, s_n\), are each composed of \(k\) characters, \(c_{il}, \ldots, c_{ik}\), with a strict ordering imposed on the characters of the string. Then the string ordering is defined

\[
\text{s}_i < \text{s}_j \quad \text{if} \quad c_{il} < c_{jl}, \quad \text{or} \\
\quad \quad \quad \quad c_{im} = c_{jm} \quad \text{for} \quad m=1-n, \quad \text{and} \quad c_{i(m+1)} < c_{j(m+1)}.
\]

Thus the lowest order character for which \(c_{im} \neq c_{jm}\) determines the order of \(s_i\) and \(s_j\).

Now if the strings are placed on the sides of a three-dimensional matrix so that the \(i\)th plane of the
matrix contains the \( i \)th characters of each string. Then just as we compared digits in the digit sorting routine, we can compare single characters. So for each column of the matrix, if the entries in the first plane are "equal" the order is determined by the higher digits. Then we can create a comparison of every string with every other string. This is exactly the same as the results of the first half of the digit sorting algorithm, thus we can sort strings of characters immediately.

A.4 Sort Time

To sort stroke numbers, we need only create the computation busses; this is immediate, \( O(1) \), so stroke sorting is immediate.

The sorting of digits requires the following:

1: Creation of Matrix Busses; \( O(1) \),
2: Determination of matrix elements; \( 1 \),
3: Compaction; \( O(1) \),
4: Stroke Sorting; \( O(1) \).

Thus the total time is \( 1 + 3O(1) \) which is \( O(1) \). So sorting on BA's is immediate.

Bartlett, M. S., "Fitting a Straight Line when both variables are subject to error", Biometrics, V. 5, pp. 205-212, 1949.


Fedorov, V. V., "Regression Problems with Controllable Variables subject to error", Biometrika, V. 61, pp. 49-56, 1974.


Halprin, "Fitting of Straight Lines and Prediction when both Variables are subject to error", JASA, V. 56, pp. 657-669, 1961.


Muller, D. E. and Preparata, F. P., "Restructuring of Arithmetic Expressions for Parallel Evaluation", 


