Design and implementation of a systolic array to solve the Algebraic Path Problem with the specific instance of the transitive and reflexive closure of a binary relation

David Gene McCall

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
McCall, David Gene, "Design and implementation of a systolic array to solve the Algebraic Path Problem with the specific instance of the transitive and reflexive closure of a binary relation" (1990). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Design and Implementation
of a Systolic Array
to Solve the Algebraic Path Problem
with the Specific Instance
of the Transitive and Reflexive
Closure of a Binary Relation

by

David Gene McCall

A Thesis Submitted
in
Partial Fulfillment
of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Prof. ________________________________
(Thesis Advisor)

Prof. ________________________________

Prof. ________________________________

Prof. ________________________________
(Department Head)

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
MAY 1990
TITLE OF THESIS: The Design and Implementation of Systolic Array to Solve the Algebraic Path Problem with Specific Instance to the Transitive and Reflexive Closure of a Binary Relation.

I, David Gene McCall, hereby grant permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use of profit.

DATE 5/7/40   David McCall
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>2.0</td>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>3.0</td>
<td>Formal Description of Algebraic Path Problem</td>
<td>4</td>
</tr>
<tr>
<td>4.0</td>
<td>Corresponding Systolic Array</td>
<td>8</td>
</tr>
<tr>
<td>5.0</td>
<td>Systolic Array Function</td>
<td>12</td>
</tr>
<tr>
<td>6.0</td>
<td>Mapping of Algorithm onto a Systolic Array</td>
<td>23</td>
</tr>
<tr>
<td>6.1</td>
<td>Description of Lines</td>
<td>25</td>
</tr>
<tr>
<td>7.0</td>
<td>Design and Implementation of Systolic Array</td>
<td>28</td>
</tr>
<tr>
<td>7.1</td>
<td>Logic Equations</td>
<td>28</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Circle Processor</td>
<td>29</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Double Square Processor</td>
<td>29</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Square Processor</td>
<td>30</td>
</tr>
<tr>
<td>7.2</td>
<td>Circuit Design</td>
<td>32</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Circle Processor</td>
<td>32</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Double Square Processor</td>
<td>33</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Square Processor</td>
<td>33</td>
</tr>
<tr>
<td>8.0</td>
<td>Processor Testing</td>
<td>35</td>
</tr>
<tr>
<td>8.1</td>
<td>Circle Processor</td>
<td>35</td>
</tr>
<tr>
<td>8.2</td>
<td>Double Square Processor</td>
<td>37</td>
</tr>
<tr>
<td>8.3</td>
<td>Square Processor</td>
<td>41</td>
</tr>
<tr>
<td>9.0</td>
<td>Array Design and Testing</td>
<td>42</td>
</tr>
</tbody>
</table>
# Table of Tables

<table>
<thead>
<tr>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Comparisons of Systolic Arrays for Solving APP</td>
</tr>
<tr>
<td>2</td>
<td>Circle Processor Operation</td>
</tr>
<tr>
<td>3</td>
<td>Double Square Processor Operation (R=0)</td>
</tr>
<tr>
<td>4</td>
<td>Double Square Processor Operation (R=1)</td>
</tr>
<tr>
<td>5</td>
<td>Propagation Delays for Processing Cells</td>
</tr>
<tr>
<td>6</td>
<td>Processor Propagation Delays</td>
</tr>
<tr>
<td>7</td>
<td>Final Propagation Delays</td>
</tr>
</tbody>
</table>
Table of Figures

1 Applications for the Algebraic Path Problem........... 10
2 Systolic Array for APP (n=4)........................... 13
3 Circle Processor...................................... 14
4 Square Processor...................................... 15
5 Double Square Processor............................... 16
6 Input to First Row of the Array....................... 22
7 Output of First Row of the Array..................... 22
8 Algorithm to Solve APP............................... 23
9 The Mapping of Line #3 onto the Systolic Array...... 24
10 The Mapping of Line #5 onto the Systolic Array..... 24
11 The Mapping of Line #9 onto the Systolic Array..... 26
12 The Mapping of Line #10 onto the Systolic Array.... 26
13 Simulation Output for Double Square Processor (R=0) 36
14 Simulation Output for Circle Processor................. 36
15 Simulation Output for Double Square Processor (R=1) 38
16 Simulation Output for Square Processor................ 40
17 Test 1 for 2x2 Array................................ 43
18 Test 2 for 2x2 Array................................ 44
19 Test 3 for 2x2 Array................................ 45
20 Simulation Output for Test 1 on 2x2 Array.......... 46
21 Simulation Output for Test 2 on 2x2 Array.......... 47
22 Simulation Output for Test 3 on 2x2 Array.......... 48
23 2x2 Systolic Array.................................. 49
24 Test 1 for 3x3 Array................................ 50
25 Test 2 for 3x3 Array........................................... 51
26 4x4 Systolic Array............................................ 53
27 Test 1 for 4x4 Array.......................................... 54
28 Test 2 for 4x4 Array.......................................... 55
29 Test 3 for 4x4 Array.......................................... 56
30 Simulation Output for Test 1 on 4x4 Array............. 57
31 Simulation Output for Test 2 on 4x4 Array............. 57
32 Simulation Output for Test 3 on 4x4 Array............. 58
33 Steps to Convert to MOSIS3 Standard Cells and
   Produce Fabrication File...................................... 61
34 Circle Processor Using MOSIS3 Standard Cells........ 62
35 Square Processor Using MOSIS3 Standard Cells........ 63
36 Double Square Processor Using MOSIS3 Standard Cells 64
37 4x4 Systolic Array Using MOSIS3 Standard Cells...... 66
38 Clock Skew after the Fanout Delay is Added (1-0)... 69
39 Clock Skew after the Fanout Delay is Added (0-1)... 69
40 Clock Skew after Back Annotation and Wiring Delays are Added (0-1).............................................. 71
41 Clock Skew after Back Annotation and Wiring Delays are Added (1-0).............................................. 71
42 Simulation for Modified Test 1 on 4x4 Array with Fanout Delays Added............................................. 72
43 Simulation for Modified Test 1 on 4x4 Array with Back Annotation and Wiring Delays Added................. 72
44 Simulation for Test 3 on 4x4 Array with Back Annotation and Wiring Delays Added............................ 74
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Simulation for Test 2 on 4x4 Array with Back Anotation and Wiring Delays Added</td>
</tr>
<tr>
<td>46</td>
<td>Simulation for Modified Test 1 on 4x4 Array at Maximum Frequency (8.3 MHz)</td>
</tr>
<tr>
<td>47</td>
<td>Simulation for Test 3 on 4x4 Array at Maximum Frequency (8.3 MHz)</td>
</tr>
<tr>
<td>48</td>
<td>Simulation for Test 2 on 4x4 Array at Maximum Frequency (8.3 MHz)</td>
</tr>
<tr>
<td>49</td>
<td>Clock Wave Form after Skewing</td>
</tr>
<tr>
<td>50</td>
<td>Chip Layout of 4x4 Systolic Array</td>
</tr>
</tbody>
</table>

- VI -
1.0 Abstract:

The Algebraic Path Problem (APP) has many practical instances to be solved. The general solution by Robert and Trystram (1986) will be discussed along with the mapping and operation of the algorithm to a systolic array. The specific instance of the APP, the transitive and reflexive closure of a binary relation, will be implemented with a discussion of the different stages ranging from the logic equations to a method of the fabrication.
2.0 Introduction:

A systolic array has been defined by Will Moore as "a regular array of processing elements all doing the same calculation and passing results on to their nearest neighbors every cycle." (Robert 1986) Although this definition is not strictly adhered to by many systolic architectures, it is still the basic underlying theme.

The systolic array that will be presented here is a regular array of three different processing elements performing similar functions. The interconnections are only to the closest horizontal or vertical neighbors. It will solve the general Algebraic Path Problem (APP).

The discussion will begin with the theory of the APP followed by the presentation of Yves Robert and Denis Trystram's (1986) algorithm for its solution. Their mapping of the algorithm and its operation as implemented on a systolic array will be explained and clarified.

A specific instance of the APP, the transistive and reflexive closure of a binary relation, will be selected and implemented. The complete stages will be stepped through from circuit
design and testing to the integrated circuit layout of MOSIS CMOS3 (MOSIS3) standard cells. At the end of the discussion a fabrication file will be completed allowing the chip to be fabricated.
3.0 **Formal Description of Algebraic Path Problem:**

The Algebraic Path Problem (APP) deals with the problem of finding the distances between all vertices of a weighted graph over a specified semiring. To begin the discussion several definitions will be given along with some corresponding examples to clarify the concepts. The first definition is that of a semigroup, the building block of a semiring.

**Definition 1:**

A pair \((G, (\cdot))\) is a semigroup if and only if:

a) \(G\) is a (nonvoid) set.

b) \(G \times G \to (\cdot)\) \(G\).

c) \((\cdot)\) is associative.

d) there exists a unique neutral element \((e)\) \(\in G\) such that, \(g (\cdot) (e) = (e) (\cdot) g = g\) for every \(g \in G\).

e) the group is commutative iff:

\[g_1 (\cdot) g_2 = g_2 (\cdot) g_1\] for all \(g_1, g_2 \in G\).

(Hartnett 1963, p. 160)

The semigroup is equivalent to a moniod which has a variation on the notation, that is \((G, (\cdot)) = \langle G, (\cdot), (e)\rangle\) (Kuich 1986, p. 5). The definition of a semiring will use the definition of a semigroup.
Formal Description

Definition 2:

A triple \((H, (+), (\ast))\) is a semiring if and only if:

a) \((H, (+))\) is a commutative semigroup with neutral element \((0)\).

b) \((H, (\ast))\) is a semigroup with neutral element \((1)\).

c) for all \(h_1, h_2, h_3 \in H\),
   \[ h_1 \ast (h_2 + h_3) = (h_1 \ast h_2) + (h_1 \ast h_3) \]
   \[ (h_1 + h_2) \ast h_3 = (h_1 \ast h_3) + (h_2 \ast h_3) \]

d) for every \(h \in H\),
   \[ (0) \ast h = h \ast (0) = (0) \]
   (Kuich 1986, pp. 5 - 6)

For those who understand the concept of a ring, notice that a semiring would be a ring if the additional property of an "inverse" for \((+)\) were added (Hartnett 1963, p. 173). Below are two examples of semirings.

Example 1:

The set of real numbers having ordinary addition with the natural neutral element \(0\) and multiplication with neutral element \(1\), \(H_1=\mathbb{R}, (+, \ast)\), is a semiring. This can be shown by first proving that \(\mathbb{R}, (+)\) is a commutative semigroup.

Clearly, for all \(r_1, r_2 \in \mathbb{R}\) where \(\mathbb{R}\) is a nonvoid set, \(r_1+r_2\)
Formal Description

$E \in \mathbb{R}$, $r_1 + 0 = r_1 + 0 = r_1$. Likewise it can be shown that
the real numbers are commutative and associative. The same
holds true for $(\mathbb{R}, \times)$ and it is known that real numbers are
distributive and zero absorptive, i.e. part (d) of the def-
nition. (Rote 1985, p. 194)

Example 2:

$H_2= (\mathbb{R} \cup (-\infty, +\infty), \min, +)$ is a semiring with zero $(0)= +\infty$
and unity $(1)=0$, if $(+\infty)+r=(+\infty)$, $(-\infty)+r=(-\infty)$, for all
$r \in \mathbb{R}$, and $(-\infty)+(+\infty)=(+\infty)$ are defined. Here it is not as
obvious that $H_2$ is a semiring. However, the min operation as
defined normally is commutative, $\min(r_1, r_2)=\min(r_1, r_2)$, and
likewise associative. Since $(-\infty)$ and $(+\infty)$ are included with
the above definitions $+$ is also commutative and associative.
Parts (d) and (e) of the semiring definition also follow to
be true.

If $r_1, r_2, r_3 \in (\mathbb{R} \cup (-\infty, +\infty))$

(d) $r_1 + \min(r_2, r_3) = r_1 + r_2$ OR $r_1 + r_3$ depending on which
of $r_2$ and $r_3$ are smaller

= $\min((r_1 + r_2), (r_1 + r_3))$

(e) $(+\infty) + r_1 = r_1 + (+\infty) = (+\infty)$ by the definitions
above.
Formal Description

(Rote 1985, p.194)

The definition of the APP as given by Robert and Trystram (1986) follows which includes the definition of a weighted graph.

Given a weighted graph \( G=(V,E,w) \) where \( V \) is a finite vertex set, \( E \) an arc set, a function \( w: E \to H \) with weights from a semiring \((H, (+), (\times))\) with zero \((0)\) and unity \((1)\), find for all pairs of vertices \((i,j)\) the quantities

\[
d_{ij} = (+) w(p), \quad p \in M_{ij},
\]

where \( M_{ij} \) denotes the set of all paths from \( i \) to \( j \).

Associated with the weighted graph described above there is a weight matrix \( A = (a_{ij}) \), where \( a_{ij} = w(i,j) \) if \((i,j)\in E \) and \( a_{ij} = 0 \) otherwise. The set of all paths \( M_{ij} \) is modified to \( M_{ij}^{(k)} \), the set of all paths from \( i \) to \( j \) which contain only vertices \( x \) with \( 1 \leq x \leq k \) as intermediate vertices. This will also modify the weight matrix calculations to

\[
a_{ij}^{(k)} = (+) w(p) \quad p \in M_{ij}^{(k)}
\]

where \( d_{ij} = a_{ij}^{(n)} \).
4.0 Corresponding Systolic Array:

A general algorithm was developed by Robert and Trystram (1986) that solves the APP for any particular instance by using only the semiring operations (+), (\*), and *, where

\[ c^* := (+(c^i \cdot (1) +(c(\*)c)) (c(\*)c(\*)c) \cdots \]

\[ i>0 \]

The above definition is the generalized infinite geometric series for \(1/(1-c)\) giving the semiring an "inverse". The algorithm is equivalent to Rote's (1985) algorithm with an improvement in time (Robert 1986, pp. 173, 179). The algorithm is given below:

```
for k <- 1 to n
begin
    a_{kk}^{(k)} <- (a_{kk}^{(k-1)})^*
    for i <- 1 to n, i\neq k
        a_{ik}^{(k)} <- a_{ik}^{(k-1)} (\*) a_{kk}^{(k)}
    for j <- 1 to n, j\neq k
    begin
        for i <- 1 to n, i\neq k
            a_{ij}^{(k)} <- a_{ij}^{(k-1)} (+) a_{ik}^{(k)} (\*) a_{kj}^{(k-1)}
            a_{kj}^{(k)} <- a_{kk}^{(k)} (\*) a_{kj}^{(k-1)}
    end
end
```
Systolic Array

The algorithm can solve a variety of applications depending upon the definitions of (+), (\star), and \star in the semiring. A chart of several applications is given in figure 1. Four additional applications were briefly described in Robert and Trystram (1986) and three are expanded upon here:

1. The determination of the inverse of a real matrix \( A \) can be performed by defining (+) and (\star) in the usual manner in \( \mathbb{R} \) and we have seen that \((\mathbb{R},+,{\star})\) is indeed a semiring. Since the definition of \( \star \) is sum of increasing power it can be defined if \( c\neq 1 \) to be \( c^* = 1/(1-c) \). This definition may seem odd since \( 1/(1-c) \) is not convergent for \( \text{abs}(c)>1 \), but the solution in this case would not exist which is a perfectly adequate solution. The algorithm actually computes \((I-A)^{-1}\) in this case but a simple modification can be done to permit \( A^{-1} \) to be computed directly (Rote 1985).

2. The shortest distance in a weighted graph has the definition as follows: \( a_{ij} \) are the weights taken in \( H=\mathbb{R} \cup \{-\infty, +\infty\} \), (+) is the minimum operation with zero \( (0)=+\infty \), (\star) is addition in \( \mathbb{R} \) extended to \( H \) (with \(-\infty \) (\star \) \(+\infty = +\infty \)) with \( (1) = 0 \), and \( \star \) is defined by if \( c>0 \) then \( c^* = 0 \) else \( c^* = -\infty \). Here the semiring is not as obvious however, in example 2 above this was shown to be a semiring.
Types of problems solved

- **Problems of connectivity**
  - $\{0, 1\}$
  - Max
  - Min

- **Enumeration**
  - **Elementary paths**
  - $S^{-} \cup \mathcal{X'}$
  - Union
  - Latin multiplication

- **Multicriteria problems**
  - Set of efficient paths
  - Union

- **Generation of regular languages**
  - **Kleene**
  - Set of words
  - Union
  - Concatenation

- **Optimization Path**
  - **Of maximum capacity**
  - $U^*$
  - $\omega^*$

- **Path with minimum number of arcs**
  - $\text{Min}$

- **Shortest path**
  - $\text{Min}$

- **Longest path**
  - $\text{Max}$

- **Path of maximum reliability**
  - $\{a | 0, ^{a1}\}$

- **Reliability of a network**
  - Idempotent polynomials
  - Symmetric difference

- **Counting**
  - Counting of paths
  - $f^*$
  - $X^*$

- **Markov chains**
  - $\{o | 0^\mathcal{X'}\}$
  - Union

- **Optimization and post-optimization problems of /cth path**
  - **i;-optimal paths**

- **Cone**
  - **Of the smallest elements**
  - Of two vectors
  - Of sums of pairs

- **Sequence of smallest elements up to**
  - Of two sequences

- **Figure 1. Applications of the APP**
  - Gondran 1984
3. The reflexive and transitive closure of a binary relation can easily be computed with the algorithm. The weight matrix's components $a_{ij}$ are defined to be boolean, (+) and (\*) are respectively the OR and AND operations, and * is defined by $c^* = \text{true}$ for all $c$, see the definition of $c^*$. The weight matrix is the relation matrix for binary relation. Clearly from the concepts of boolean algebra this is a semiring with the zero $(0) = 0$ and unity $(1) = 1$.

Application 3 described above will be the one designed and implemented on a single chip for the CMOS3 process. A practical use for the reflexive and transitive closure is determining if there is a path between any two vertices within a graph. The graph could correspond to cities, nodes in a circuit, etc.
5.0 **Systolic Array Function:**

The algorithm described above is implemented on a two-dimensional array of \(n\) by \(n+1\) orthogonally connected processors (see figure 2). Each row of the array, \(k\), and has \(n+1\) processors labelled \(P_{k1}, \ldots, P_{kn+1}\). The weight matrix \(A\) followed by \(I_n\), which will be represented by \(C\), is fed into the array one row at a time in a staggered fashion (see figure 6).

There are three types of processors which perform different functions. The circle processors implement the \(*\)-operation on the first input and afterwards act as delay cells (see figure 3 parts (c) and (d)). The square processors initialize their registers with the modification of the first input and afterward act as multiply-and-add cells (see figure 4 parts (c) and (d)). The double square processors are similar to the square processors except the register value is initialized differently (see figure 5 parts (c) and (d)). (Robert 1986, p. 174; Robert 1987, p. 187)

Each row \(k\) of the array performs the \(k^{th}\) phase of the algorithm. Processors \(P_{k1}\) transmit the input data arriving from the top to the right. As the data travels to the right, the square processors merely pass this data along unaffected. They modify
Figure 2. Systolic Array for APP
**Figure 3. Circle Processor**

(a) truth table

<table>
<thead>
<tr>
<th>Init</th>
<th>Ain</th>
<th>Bout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) circuit implementation

```
IF Init = TRUE THEN
  Bout := Ain
  Init := .FALSE
ELSE
  Bout := Ain
END IF
```

(c) symbol

(d) operation
(b) circuit implementation

IF \( \text{Init} = \text{TRUE} \) THEN
   \( A := \text{Bin} \) and \( \text{Bin} \)
   \( \text{Out} := \text{Bin} \)
   \( \text{Out} := \text{NIL} \)
ELSE
   \( \text{Out} := \text{Bin} \)
   \( \text{Out} := \text{Bin} \)
END IF

Figure 4. Square Processor

(c) symbol

(d) operation

(a) truth tables
Figure 5.
Double Square Processor

(a) truth tables

(b) circuit implementation

(c) symbol

(d) operation
and store the first data arriving from the top but modify and pass downward the following data arriving from the top.
Likewise the $P_{k,n+1}$ processors store the first input data and modify and pass the following data downward. As seen in figures 3, 4, and 5, the control of the processor's current operation is dependent upon the variable $Init$.

It can be seen that after $2n$ input data has gone through the array, its length has been shortened to $n$, a row of output data. The shortening takes place because each cell that passes data downward does so with its first datum at half the rate in which all data arrives. As the data flows through the array, it is reordered at each row. The element $a_{k,k}^{(k)}$ is computed in the circle processor $P_{k,1}$ and moves rightward to be stored in the double-square processor $P_{k,n+1}$. The non-diagonal elements $a_{i,k}^{(k)}$, $i \neq k$ are computed by the square processors $P_{k,2}$, $\ldots$, $P_{k,n+1}$ where element $a_{i,k}^{(k)}$ is stored in processor $P_{k,\lfloor i-k+1 \rfloor \mod n}$. Therefore row $k$ of the array outputs matrix $C^{(k)}$ in row order with the leftmost row $k+1$. (Robert 1986, pp. 174 - 177; Robert 1987, pp. 211 - 213)

The operation of row 1 will be detailed through several steps to aid in the understanding of the data flow.
Array Functioning

time step $P_{i} \quad P_{i+1} \quad P_{i+2} \quad P_{i+3} \quad P_{i+4} \quad P_{i+5}$

1. $a_{i+1}^{(1)} = (a_{i+1}^{(0)})^*$

   remark: $P_{i+1}$ receives $a_{i+1}^{(0)}$ and computes $a_{i+1}^{(1)}$ and passes it to the right

2. $a_{i+2}^{(0)} \quad a_{i+1}^{(1)}$

   remark: $P_{i+1}$ passes $a_{i+2}^{(0)}$

   $P_{i+2}$ computes $a_{i+1}^{(1)} = a_{i+1}^{(0)}(x)a_{i+1}^{(1)}$ and passes $a_{i+1}^{(1)}$ to the right.

3. $a_{i+3}^{(0)} \quad a_{i+2}^{(1)} \quad a_{i+1}^{(1)}$

   remark: $P_{i+1}$ passes $a_{i+3}^{(0)}$

   $P_{i+2}$ computes $a_{i+2}^{(1)} = a_{i+2}^{(0)}(x)a_{i+2}^{(1)}$ and passes it down while passing $a_{i+2}^{(0)}$ to the right

   $P_{i+3}$ computes $a_{i+1}^{(1)} = a_{i+1}^{(0)}(x)a_{i+1}^{(1)}$ and passes $a_{i+1}^{(1)}$ to the right

4. $a_{i+4}^{(0)} \quad a_{i+3}^{(1)} \quad a_{i+2}^{(1)} \quad a_{i+1}^{(1)}$

   remark: $P_{i+1}$ passes $a_{i+4}^{(0)}$

   $P_{i+2}$ computes $a_{i+3}^{(1)} = a_{i+3}^{(0)}(x)a_{i+3}^{(1)}(x)a_{i+3}^{(0)}$ and passes it down while passing $a_{i+3}^{(0)}$ to the right

   $P_{i+3}$ computes $a_{i+2}^{(1)} = a_{i+2}^{(0)}(x)a_{i+2}^{(1)}(x)a_{i+2}^{(0)}$ and passes $a_{i+2}^{(0)}$ to the right

   $P_{i+4}$ computes $a_{i+1}^{(1)} = a_{i+1}^{(0)}(x)a_{i+1}^{(1)}$ and passes $a_{i+1}^{(1)}$ to the right

5. $a_{i+5}^{(1)} \quad a_{i+4}^{(1)} \quad a_{i+3}^{(1)} \quad a_{i+2}^{(1)} \quad a_{i+1}^{(1)}$
remark: P_{11} passes the first 1 of the identity matrix to the right

P_{12} computes a_{24}^{('1')} passing it downward while passing a_{14}^{('0')} to the right

P_{13} computes a_{33}^{('1')} passing it downward and passes a_{13}^{('0')} to the right

P_{14} computes a_{42}^{('1')}=a_{42}^{('0')}+a_{31}^{('1')}(x)a_{12}^{('1')} and passes it down while passing a_{12}^{('0')} to the right

P_{15} stores a_{11}^{('1')}

\begin{bmatrix}
6 & 0 & a_{21}^{('1')} & a_{34}^{('1')} & a_{43}^{('1')} & a_{12}^{('1')}
\end{bmatrix}

remark: P_{11} passes 0 from the identity matrix

P_{12} computes a_{21}^{('1')}=1(a_{21}^{('1')}(x)1 passing it downward and passing 1 to the right

P_{13} computes a_{34}^{('1')} passing it downward and passes a_{14}^{('0')} to the right

P_{14} computes a_{43}^{('1')} passing it downward and passes a_{13}^{('0')} to the right

P_{15} computes a_{12}^{('1')}=a_{11}^{('1')}(x)a_{12}^{('0')} and passes it downward

The above description is summarized in figures 6 and 7, where the first is the input data and the latter is the output of the row.
Array Functioning

The time necessary for the array to process the input data is $5n+2$ where $n$ is the array size (Robert 1986, p. 174). As previously described, the systolic array size is $n(n+1)$ which will give a size of order $n^2$. The time and size can be compared to other algorithms and array sizes, see table 1 below.

From the comparisons in table 1, it can be seen that the present algorithm is one of the best in terms of time and array size. Plus, it is a general solution of the APP. The particular instance, the transitive and reflexive closure, that will be implemented with this algorithm has an equal order of time to the one specifically design for it. In fact the last entry in the table, Kung-Lo-Lewis, is a better implementation because it can begin a new solution every $n$ time steps whereas Robert-Trystram's needs $2n$ steps (Benaini 1989, p. 74).
<table>
<thead>
<tr>
<th>Application</th>
<th>Area</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guibas transitive closure</td>
<td>n^2</td>
<td>6n</td>
</tr>
<tr>
<td>Kung-Lo shortest paths</td>
<td>n^2</td>
<td>7n</td>
</tr>
<tr>
<td>Kramer-Leeuwen matrix inversion</td>
<td>n^2</td>
<td>6n</td>
</tr>
<tr>
<td>Nash-Hansen matrix inversion</td>
<td>3n^2/2</td>
<td>5n</td>
</tr>
<tr>
<td>Robert-Tchuente matrix inversion</td>
<td>n^2</td>
<td>5n</td>
</tr>
<tr>
<td>Rote (1985) general</td>
<td>n^2</td>
<td>7n</td>
</tr>
<tr>
<td>Robert-Trystram general</td>
<td>n^2</td>
<td>5n</td>
</tr>
<tr>
<td>(1986)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kung-Lo-Lewis general</td>
<td>n^2</td>
<td>5n</td>
</tr>
</tbody>
</table>

(Benaini 1989, p.74)

Table 1: Comparisons of systolic arrays for solving APP
Figure 6. 
Input to first row of the Systolic Array

Figure 7. 
Output of the first row of the Systolic Array
6.0 **Mapping of Algorithm onto a Systolic Array:**

The mapping of Robert and Trystram's algorithm for solving the Algebraic Path Problem (APP) for the specific instance of reflexive and transitive closure of a binary relation is done by

Robert and Trystram's Algorithm to Solve APP

```plaintext
1: for k <-- 1 to n
2: begin
3:   a_{kk}(k) <-- (a_{kk}(k-1))
4:   for i <-- 1 to n, i\neq k
5:    a_{ik}(k) <-- a_{ik}(k-1) and a_{kk}(k)
6:   for j <-- 1 to n, j\neq k
7:    begin
8:      for i <-- 1 to n, i\neq k
9:       a_{ij}(k) <-- a_{ij}(k-1) or a_{ik}(k) and a_{kj}(k-1)
10:      a_{kj}(k) <-- a_{kk}(k) and a_{kj}(k-1)
11:    end
12: end
```

**FIGURE 8**

the use of quasi-dependence graphs which look similar to the systolic array depicted in figure 2 except the double square processor is denoted as a single square. The algorithm which is
Mapping of Line 3

Figure 9. The mapping of line #3 onto the Systolic Array

Mapping of Line 5

Figure 10. The mapping of line #5 onto the Systolic Array
listed above in figure 8 has several nested loops. Each relevant instruction line will be discussed separately. In the following description, a relation with four elements will be used. This leads to a 4x4 array. The choice of size 4 is partly because, it is the largest size which can be easily explained, as in the previous section. Also, it limits the number of pins needed on an integrated circuit chip to a reasonable number when the array is fabricated.

6.1 Description of lines:

The first instruction, line #3, has the quasi-dependance graph as illustrated in Figure 9. The value \( a_{kk}(k) \) only depends on \( a_{kk}(k-1) \) but there needs to be \( n \) of these because of the for loop in line #1. In Figure 9, the relevant nodes that perform this function are in bold and depicted on the final systolic array to give a feeling of how the mapping will proceede and where the data is flowing.

The next instruction, line #5, has two dependences \( a_{kk}(k-1) \) and \( a_{kk}(k) \) with one output \( a_{kk}(k) \). This relation needs to be repeated for all \( i \) between 1 and \( n \) except for \( i=n \). The quasi-depence graph is depicted in Figure 10. Once again there are \( n \) repetitions because of line #1 and the nodes that perform the
Mapping of Line 9

Figure 11.
The mapping of line #9 onto the Systolic Array

Mapping of Line 10

Figure 12.
The mapping of line #10 onto the Systolic Array
operation are bold. Here $a_{ik}^{(k)}$ is shown within the node to indicate that it will be stored there for future use in the final systolic array.

The final loop, line #8, encompasses an instruction that when mapped has three inputs, $a_{ij}^{(k-1)}$, $a_{ik}^{(k)}$, $a_{kj}^{(k-1)}$, and one output, $a_{ij}^{(k)}$. The quasi-dependence graph notation is shown in Figure 11. Here the value that was stored in the previous loop is used as one of the inputs. As seen by the for statement, line #8, there are $n-1$ nodes. The other loops are taken care of by the data flow.

The last instruction, line #10, is depicted in Figure 12. Here one of the two inputs has been stored within the node from previous data flow in the array. This line is in fact taking care of the exclusion of $i=k$ in the previous loop.
The choice of the transitive and reflexive closure of a binary relation as the instance for the systolic array to solve was done because of its ease of implementation and the ability to describe its functioning in simple terms. There are other specific algorithms for solving the stated instance, however their time and size are of the same order as this solution $O(n)$ and $O(n^2)$, respectively (see Table 1).

In the following subsections, the necessary equations and circuits are derived and constructed for the systolic array. All the input and output data paths of the processors in figure 2 are actually 2 bits wide as can be seen in figures 3 to 5 part (c). One of the bits is the data, which is labeled either $Axx$ or $Bxx$ for vertically or horizontally flowing, while the other, which is labeled $Dxx$, signals whether the data is valid. The $xx$ is replaced by in or out for input and output, respectively.

7.1 Logic Equations:

The logic equations for the three different types of cells were derived from the functions each cell needed to perform. In figures 3 through 5 part (d), a programming type implementation
Design and Implementation

is given at the bottom for the circle, square, and double square processors respectively. The necessary logic equations are derived below in order of processor complexity.

7.1.1 Circle Processor:

The circle processor first performs the \(^\ast\)-operation and then acts as a delay cell. These functions are summarized in the truth table at the upper left corner of figure 3(a). When Init is true Bout should pass 1, the value of the \(^\ast\)-operation, otherwise it should pass Ain. The value of Init should only remain true if there has not been any data and it is presently true. Valid Data is signalled by Din and therefore Dout should directly follow Din. From these truth tables and the previous description, the following logic equations are derived for the outputs of the circle processor.

\[
\begin{align*}
\text{Bout} &= \text{Ain or Init} & [1] \\
\text{Dout} &= \text{Din} & [2] \\
\text{Init} &= -\text{Din and Init} & [3]
\end{align*}
\]

7.1.2 Double Square Processor:

The double square processor's functioning is depicted at the bottom of figure 5(d). Initially Bin is stored within the pro-
Design and Implementation

Processor in R for future computations and no valid value is
passed through Aout. Afterwards, Aout is computed and passed.
Since no data is passed the first time Dout does not directly
follow Din however, Init's function remains the same. The truth
table summarizing these actions are given in the upper left
corner of figure 5(a). The value of Aout is only given definite
values when Init and Din are false and true, respectively. The
rest are don't cares, X. The reason for this is that Din sig-
nals when there is valid data. The storage of the first re-
received value R should remain the same when Init is false but,
should latch the value of Bin when Init and Din are both true.
Karnaugh maps were used to reduce terms in the equations and
are given below the truth table in figure 5(a). The following
logic equations were derived:

\[
\begin{align*}
A_{out} &= R \quad \text{and} \quad Bin \quad [4] \\
D_{out} &= -\text{Init} \quad \text{and} \quad Din \quad [5] \\
R &= (\text{Init and Bin}) \quad \text{or} \quad (-\text{Init and R}) \quad [6] \\
\text{Init} &= \text{Init and } -\text{Din} \quad [7]
\end{align*}
\]

7.1.3 Square Processor:

This is the most complex processor since there are two inputs
and two outputs. As shown at the bottom of figure 4(d), the pro-
cessor AND's the initial two inputs, Ain and Bin, and stores it
Design and Implementation

in R, while at the same time passing Bin to the right and nothing down. After the first values have been processed, Bin is still passed to the right through Bout and the value passed downward, Aout, is the sum and product expression for Ain, Bin, and R. The Init value depends on two data valid flags, DAin and DBin, and should only remain true when it is true and either of the latter values is false. Since the Aout value is delayed one time step its data valid flag DAout is exactly the same as that for the double square processor. Bin is always passed uneffect-ed and therefore Bout and DBout directly follow their input counterparts Bin and DBin. These conclusions are summarized in the three truth tables on the left side of figure 4(a). Once again Karnaugh maps were used to minimize terms for the complicated equations, R and Aout. There are two equations for R given below its Karnaugh map. The first is the equation directly derived from the mapping while the second has the additional data valid flags added to ensure R only changes when valid data is available. The equations are summarized below:

\[
\begin{align*}
A_{\text{out}} &= \text{Ain} \text{ or } (\text{Bin and R}) \\
D_{\text{Aout}} &= -\text{Init and DAin} \\
B_{\text{out}} &= \text{Bin} \\
D_{\text{Bout}} &= \text{DBin} \\
\text{R} &= (\text{Init and DBin and DAin and Ain and Bin}) \\
&\quad \text{or } (-\text{Init and R})
\end{align*}
\]
Design and Implementation

\[ \text{Init} = \text{Init and } (-\text{DBin or } -\text{DAin}) \]  \hspace{1cm} [13]

7.2 Circuit Design:

The circuits for the three different processors were designed directly from the derived logic equations. A mixed logic approach was used with NAND and NOR gates. The choice to use exclusively NAND and NOR gates was due to the fact that the design was going to be implemented with CMOS technology and these gates are more easily made. The inputs were all latched using D-flip/flops. This was necessary for proper operation, plus it gave both the input and its inverted value. The D-flip/flops were also used to store the internal processor variables, Init and R. A reset line has been included to set all input latches and R to the false state while Init is set to true.

7.2.1 Circle Processor:

The three equations for the circle processor when converted to mixed logic notation become:

\[ [1] \quad \text{Bout},h = \text{Ain},l \text{ or Init},l \]  \hspace{1cm} [14]
\[ [2] \quad \text{Dout},h = \text{Din},h \]  \hspace{1cm} [15]
\[ [3] \quad \text{Init},h = -\text{Din},l \text{ and Init},l \]  \hspace{1cm} [16]

In the circuit diagram at the upper right corner of figure 3(b)
Design and Implementation

these equations are directly implemented.

7.2.2 Double Square Processor:

Similarly, the equations for the double square processor are
converted to mixed logic notation for direct implementation
using NAND and NOR gates.

\[
\begin{align*}
[4] \quad & A_{out,h} = R_{1} \text{ and } B_{in,1} \quad [17] \\
[5] \quad & D_{out,h} = \neg I_{init,1} \text{ and } D_{in,1} \quad [18] \\
[6] \quad & R_{h} = (I_{init,h} \text{ and } B_{in,h}),\text{ or} \quad [19] \\
& \quad \quad \quad \quad (-I_{init,h} \text{ and } R_{h},)\text{, or} \\
[7] \quad & I_{init,h} = I_{init,1} \text{ and } \neg D_{in,1} \quad [20]
\end{align*}
\]

Refer to the upper right corner of figure 5(b) for the circuit
diagram.

7.2.3 Square Processor:

The final equations for the square processor are converted
below. However, in the upper right corner the circuit diagram
of figure 4(b), an 8 input NAND gate is used because TTL parts
were used during this phase of the design. The 8 input NAND was
used as an 5 input NAND by tying together the proper lines.

\[
\begin{align*}
[8] \quad & A_{out,h} = A_{in,1} \text{ or } (B_{in,h} \text{ and } R_{h}),1 \quad [21] \\
[9] \quad & D_{Aout,h} = \neg I_{init,1} \text{ and } D_{Ain,1} \quad [22]
\end{align*}
\]
Design and Implementation

[10] \( \text{Bout},h = \text{Bin},h \) [23]


[12] \( \text{R},h = (\text{Init},h \text{ and } \text{DBin},h \text{ and } \text{DAin},h \text{ and } \text{Ain},h \text{ and } \text{Bin},h),l \)

or \( (-\text{Init},h \text{ and } \text{R},h),l \) [25]

[13] \( \text{Init},h = \text{Init},l \text{ and } (-\text{DBin},h \text{ or } -\text{DAin},h),l \) [26]
8.0 Processor Testing:

Each of the processor cells were simulated by an exhaustive test of the complete truth table for the particular cell. This was done by doing Quicksim simulations on the gate-level circuit designs.

8.1 Circle Processor:

Since there are only two inputs and outputs for this cell a relatively simple set of input forcing functions produced the small truth table given below, table 2. The input forcing function event times in nanoseconds are also given to be compared to the Quicksim simulation output in figure 14.

<table>
<thead>
<tr>
<th>time</th>
<th>Ain</th>
<th>Din</th>
<th>Bout</th>
<th>Dout</th>
<th>-Init</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Circle Processor Operation
Figure 13. Simulation output for the Double Square Processor (R=0)

Figure 14. Simulation output for the Circle Processor
Processor Testing

The -Init value is the -Q output of the D-flip/flop and should be one time step behind.

This simulation test also proved the proper functioning of the -Init signal because it did only change when there was valid data available. In other tests not shown, the other two variations on inputs Ain and Din when Init is true were tested and verified. Please refer to figure 3 for the originally created truth tables.

8.2 Double Square Processor:

Similarly to the circle processor, the double square processor has two inputs and outputs. The truth table, table 3, given below has the same format as the one in table 2. The values can be compared to the Quicksim simulation output in figure 13. The Init and R values are the Q outputs of the D-flip/flops and should be one time step behind the other values.

Similar test were done to make R true and are given below, table 4, and in figure 15.

These tests also proved the proper functioning of the Init and R signals. The described tables, 3 and 4, can also be compared
Figure 15. Simulation output for the Double Square Processor (R=1)
to the original truth tables created in figure 5.

<table>
<thead>
<tr>
<th>time</th>
<th>Bin</th>
<th>Din</th>
<th>Aout</th>
<th>Dout</th>
<th>Init</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Double Square Processor Operation (R=0)

<table>
<thead>
<tr>
<th>time</th>
<th>Bin</th>
<th>Din</th>
<th>Aout</th>
<th>Dout</th>
<th>Init</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>12000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>13000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>16000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: Double Square Processor Operation (R=1)
Figure 16. Simulation Output for the Square Processor
8.3 Square Processor:

As with the previous processors, simulation tests were conducted on the square processor. The simulation output is shown in figure 16. A detailed truth table will not be presented here because the listing would be too long and the reader can verify the results using the original truth table in figure 4(a).
9.0 Array Design and Testing:

The processing cells were connected to form several different array sizes for testing. Initially a 2x2 array was created and then a 3x3 and a 4x4 followed. The testing was done using Quicksim for the logic simulations and with a computer program, listed in Appendix A, that strictly followed the program implementation. These two outputs were compared against one another and against hand calculations for correctness. During this phase of the testing it was discovered that there had been an error within one of my main texts (Robert 1986). Luckily the problem was the switching of AND and OR operations for the transistive and reflexive closure, which was thought to be a problem before any implementation began.

The clock period was set at 1000ns to avoid any problems that might occur because of propagation delays or rise and fall times.

9.1 2x2 Array Testing:

Three different input matrices were used for testing the 2x2 array. The matrices, computer program output, and directed graphs are given in figures 17 to 19. The computer program
Figure 17. Test #1 for 2x2 Array
Program Input
Start: 1 -1
0 0
1 0
0 0
End:-1 1

Program Output
Start: 1 -1
0 0
End:-1 1

Given Matrix
\[
\begin{bmatrix}
1 & 0 \\
0 & 0
\end{bmatrix}
\]

Transitive & Reflexive Closure
\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\]

Figure 18. Test #2 for 2x2 Array
Program Input

Start: 0 -1
1 0
1 0
0 0
End: -1 1

Program Output

Start: 1 -1
0 0
End: -1 1

Given Matrix

\[
\begin{bmatrix}
0 & 0 \\
1 & 0 \\
\end{bmatrix}
\]

Transitive & Reflexive Closure

\[
\begin{bmatrix}
1 & 0 \\
1 & 1 \\
\end{bmatrix}
\]

Figure 19. Test #3 for 2x2 Array
Figure 20. Simulation output for test #1 of 2x2 Array
Figure 21. Simulation output for test #2 for 2x2 Array
Figure 22. Simulation output for test #3 for 2x2 Array
Figure 23. 2x2 Systolic Array
Program Input

Start: 0 -1 -1  
   1  0 -1  
   1  0  0  
   1  1  0  
   0  0  0  
   0  1  0  
   -1  0  0  
End: -1 -1 1

Program Output

Start: 1 -1 -1  
   1  0 -1  
   1  1  0  
   -1  1  0  
End: -1 -1 1

Given Matrix

\[
\begin{bmatrix}
0 & 1 & 1 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix}
\]

Transitive & Reflexive Closure

\[
\begin{bmatrix}
1 & 1 & 1 \\
0 & 1 & 1 \\
0 & 0 & 0
\end{bmatrix}
\]

Figure 24. Test #1 for 3x3 Array
Figure 25. Test #2 for 3x3 Array
Array Design

showed the data flow into and out of the array and simulated invalid data as a -1. This can be compare with figures 6 and 7 to see the skewed data input and output.

The Quicksim simulation outputs are respectively given in figures 20 to 22. These simulations show the values of all inputs, outputs, and intermediate points. The matrices in figures 17 to 19 were compared to the simulations.

The symbolic diagram of the array is depicted in figure 23. At this stage all lines were tested for correct operation. The computer program was able to print the state of the array at each time interval and was used to compare against the Quicksim output for correctness. Thorough testing on this 2x2 array prevented major problems with any larger sizes.

9.2 3x3 Array Testing:

Two input matrices were tested against a 3x3 array, see figures 24 to 25. The Quicksim simulations are not included for sake of brevity, however they were done and compared with the computer simulations and hand calculations.

9.3 4x4 Array Testing:
Figure 26. 4x4 Systolic Array
Figure 27. Test #1 for 4x4 Array
Program Input

Start: 0 -1 -1 -1
1 1 -1 -1
0 0 0 -1
1 1 1 1
1 1 0 1
0 0 1 1
0 1 0 0
0 0 0 0
-1 0 1 0
-1 -1 0 0
End: -1 -1 -1 1

Program Output

Start: 1 -1 -1 -1
1 1 -1 -1
1 1 1 -1
1 1 1 1
1 1 1 1
-1 1 1 1
-1 -1 1 1
End: -1 -1 -1 1

Given Matrix

Transitive & Reflexive Closure

Figure 28. Test #2 for 4x4 Array
Program Input

Start:
1 -1 -1 -1
1 0 -1 -1
1 0 0 -1
1 0 0 0
0 0 0 0
0 1 0 0
0 0 0 0
-1 0 1 0
-1 -1 0 0
End: -1 -1 -1 1

Program Output

Start:
1 -1 -1 -1
1 0 -1 -1
1 1 0 -1
1 0 0 0
-1 0 1 0
-1 -1 0 0
End: -1 -1 -1 1

Given Matrix

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Transitive & Reflexive Closure

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 29. Test #3 for 4x4 Array
Figure 30. Simulation output for test #1 on 4x4 Array

Figure 31. Simulation output for test #2 on 4x4 Array
Figure 32. Simulation output for test #3 on 4x4 Array
The circuit diagram for a 4x4 array is depicted in figure 26. It is merely the logical expansion of the 2x2 array. During the array testing, the intermediate points were not displayed in the Quicksim simulations because they were checked during the 2x2 testing and the information would be too overwhelming for easy and accurate verification. The three input matrices tested are depicted in figures 27 to 29 along with the Quicksim simulation in figures 30 to 32, respectively.
10.0 Conversion of Design to MOSIS3 Standard Cells:

The 4x4 array can be made into an integrated circuit using MOSIS CMOS3 (MOSIS3) standard cells. Mentor Graphic's Corporation's Cell Station design automation tools running on the Apollo workstations were used to layout the design. The chip can be fabricated with these standard cells using the MOSIS facility. The TTL component design therefore must be converted to standard cell formats, see figure 33.

The NAND and NOR gates were directly converted except that the mixed logic symbols were not available, see figures 34 to 36. In the square processor, the 8-input NAND gate which was being used as an 5-input NAND gate had to be converted to a two stage implementation using two NANDs and one NOR, see figures 3 and 35.

Most of the D-flip/flops were changed to ones that had only the necessary outputs and inputs, i.e. only Q and/or only R. Latches were going to be used instead of the D-flip/flops, but after checking propagation delays and other timing requirements it was discovered that the flip/flops were as fast or faster than the corresponding latch with only a small increase in area size (Heinbuch 1988).
### Conversion to MOSIS3

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MOSIS_NETED file</td>
<td>NETED version for MOSIS3</td>
</tr>
<tr>
<td>2 MOSIS_EXPAND_COMP file CMOS3</td>
<td>Builds netlist</td>
</tr>
<tr>
<td>3 MOSIS_DESIGN_CHECKER file CMOS3</td>
<td>Electrical Check</td>
</tr>
<tr>
<td>4 MOSIS_EXPAND_DESIGN file CMOS3</td>
<td>Flattens Design</td>
</tr>
<tr>
<td>5 QUICKSIM file</td>
<td>Simulates Circuit</td>
</tr>
<tr>
<td>6 MOSIS_ADD_DELAY file CMOS3 -FO</td>
<td>Adds Fanout Delay</td>
</tr>
<tr>
<td>7 QUICKSIM file</td>
<td>Simulates Circuit</td>
</tr>
<tr>
<td>8 MOSIS_LAYOUT LOGIC_ENTRY file CMOS3</td>
<td>Creates physical design</td>
</tr>
<tr>
<td>9 MOSIS_LAYOUT CELLFLOOR file CMOS3</td>
<td>Generates floorplan</td>
</tr>
<tr>
<td>10 MOSIS_LAYOUT EDIT_PARMS file CMOS3</td>
<td>Edit floorplan</td>
</tr>
<tr>
<td>11 MOSIS_LAYOUT CELLFLOOR file CMOS3</td>
<td>Must rerun after edit</td>
</tr>
<tr>
<td>12 MOSIS_LAYOUT CELLPLACE file CMOS3</td>
<td>Placement of cells</td>
</tr>
<tr>
<td>13 MOSIS_LAYOUT CELLPOWER file CMOS3</td>
<td>Routes power networks</td>
</tr>
<tr>
<td>14 MOSIS_LAYOUT CELLRoutes file CMOS3</td>
<td>Routes signal networks</td>
</tr>
<tr>
<td>15 MOSIS_LAYOUT CELLSQUEEZE file CMOS3</td>
<td>Removes excess in routing channels</td>
</tr>
<tr>
<td>16 MOSIS_LAYOUT MINROUTE file CMOS3</td>
<td>Minimizes use of poly</td>
</tr>
<tr>
<td>17 MOSIS_ADD_DELAY file CMOS3 -BA -LO</td>
<td>Adds back annotation and wiring delays</td>
</tr>
<tr>
<td>18 QUICKSIM file</td>
<td>Simulates Circuit</td>
</tr>
<tr>
<td>19 MOSIS_LAYOUT PREGraph file CMOS3</td>
<td>Generates working file</td>
</tr>
<tr>
<td>20 MOSIS_LAYOUT CELLGRAPH file CMOS3</td>
<td>Allows manual editing</td>
</tr>
<tr>
<td>21 MOSIS_LAYOUT CELLVERIFY file CMOS3</td>
<td>Final validity check</td>
</tr>
<tr>
<td>22 MOSIS_LAYOUT GDS2_OUTPUT file CMOS3</td>
<td>Generates layout information in proper format for fabrication</td>
</tr>
</tbody>
</table>

Figure 33. Steps to convert to MOSIS3 standard cells and produce fabrication file
<table>
<thead>
<tr>
<th>Init</th>
<th>Ain</th>
<th>Bout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Init</th>
<th>Din</th>
<th>Init</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

IF Init = TRUE THEN
Bout := Ain
Init := FALSE
ELSE
Bout := Ain
END IF

Figure 34. Circle Processor using MOSIS3 Standard Cells
Figure 35.
Square Processor using MOSIS3 Standard Cells
Figure 36.
Double Square Processor using MOSIS3 Standard Cells

IF Init = TRUE THEN
R := Bin
Init := FALSE
Aout := NIL
ELSE
Aout := R and Bin
END IF
Conversion to MOSIS3

The I/O inputs for the TTL implementation had to be changed to I/O pads for the chip layout. Additionally, Vdd and Vss pads needed to be included within the circuit diagram for proper layout. The circuit diagram is given in figure 37 and was developed using Mentor Graphic's Schematic Editor, NETED, with CMOS3 circuit elements. The editor can be called into operation using the command MOSIS_NETED. A complete listing of the additional commands and steps necessary for the conversion to a MOSIS3 standard cell layout is given in figure 33.

The 4x4 array was chosen for the chip layout because the 20 pins required will fit within the standard 24 pin chip. The next array size which would be useful, an 8x8, would require 70 pins.

The array's outputs were not latched. This may give glitches on these lines during the zero state of the clock. The reasons for not latching the outputs are: (1) More than one chip could be connected together then, the latching would be redundant; (2) The latching can be done externally, especially for the additional design which would be discussed later where several chips could be connected together to form any array size.
Figure 37. 4x4 Systolic Array using MOSIS3 Standard Cells
11.0 **Final Simulation and Testing:**

To perform the final simulations and testing, initially Cell Station commands in steps 1 through 5 of figure 33 were completed. The design was converted to use MOSIS3 standard cells, expanded, and checked. During checking it was discovered that additional buffers were needed to drive the clock and reset lines because the fanout was over thirty. Compare figures 26 and 37 for the changes.

The same tests that were run before on the 4x4 array were again completed. These proved to agree exactly. At this point propagation delays were measured for the individual processors. The table below gives the results of the worst case.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Aout</th>
<th>DOut</th>
<th>Bout</th>
<th>DBout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>--</td>
<td>--</td>
<td>31ns</td>
<td>20ns</td>
</tr>
<tr>
<td>Square</td>
<td>31ns</td>
<td>32ns</td>
<td>20ns</td>
<td>20ns</td>
</tr>
<tr>
<td>Double Square</td>
<td>32ns</td>
<td>32ns</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**Table 5: Propagation Delays for Processors**

These values were determined using Quicksim's ability to list times at which a signal changes state.
11.1 Fanout delay:

Since the array was logically functioning properly, the fanout delays were added next, using steps 6 to 7 of figure 33. The clock period remained at 1000ns to avoid any problems with propagation delays and rise and fall times. The previously mentioned delays and times were measured to ensure no problems were actually occurring. The propagation delays are summarized below.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Aout</th>
<th>DAout</th>
<th>Bout</th>
<th>DBout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>--</td>
<td>--</td>
<td>31.6ns</td>
<td>20.7ns</td>
</tr>
<tr>
<td>Square</td>
<td>31.7ns</td>
<td>34ns</td>
<td>21.1ns</td>
<td>23.9ns</td>
</tr>
<tr>
<td>Double Square</td>
<td>33.0ns</td>
<td>34.1ns</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 6: Processor Propagation Delays

The above values in table 6 were once again calculated from Quicksim's listing output.

The rise and fall times of the clock signal determined by Cell Station tools were checked during this step and subsequent steps to ensure proper signal timing. The other signals were not critically checked. The typical fanout was three. Whereas the clock signal was connected to at least 20 devices.
Figure 38. Clock skew after the fanout delay is added (0-1).
N$633$ is the pad output. N$634$ is the buffer output.

Figure 39. Clock Skew after the fanout delay is added (1-0).
N$633$ is the pad output. N$634$ is the buffer output.
Final Testing

Once the fanout was added, the clock signal, which drove the processors, rise time to change from 4ns to 13.933 and its fall time from 4.19ns to 12.656ns. The clock skew which occurred because of the input pad and additional driving buffer was also considered. A pictorial representation is given in figures 38 and 39 of the skewing. The input pad delayed the zero to one transition 6ns. This compounded with the drive buffer produced a total delay of 20.1ns. Similarly the one to zero transition was delayed 20.5ns. All of these restrictions on the clock signal will be considered when the maximum clock frequency is calculated after all delays are added.

A simulation output for a single test case is given in figure 42. The test case is similar to that depicted in figure 27 except that the extra relation 2->4 was added, i.e. 2 is mapped to 4.

11.2 Back Annotated and Wiring Delay:

The standard cells making up the array's integrated circuit were placed and then routed on the chip floor plan. The delays due to the routing wires were back annotated into the simulation properties. These wiring delays were the final ones
Figure 40. Clock Skew after wiring delay and back annotation are added (0-1). N$633$ is the pad output. N$634$ is the buffer output.

Figure 41. Clock Skew after wiring delay and back annotation are added (1-0). N$633$ is the pad output. N$634$ is the buffer output.
Figure 42. Simulation output for modified test #1 on 4x4 array with fanout delays added.

Figure 43. Simulation for modified test #1 on 4x4 array with wiring delays and back annotation added.
to be added for simulation in steps 8 to 18 of figure 33. The clock skew had increased to 22.8ns and 23.8ns for one to zero and zero to one transitions, respectively, see figures 40 to 41. The rise and fall times were also increased to 17.106 and 15.362, respectively. The new propagation delays for the processors are given below.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Aout</th>
<th>DAout</th>
<th>Bout</th>
<th>DBout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>--</td>
<td>--</td>
<td>31.7ns</td>
<td>21.3ns</td>
</tr>
<tr>
<td>Square</td>
<td>32.0ns</td>
<td>34.7ns</td>
<td>21.4ns</td>
<td>21.3ns</td>
</tr>
<tr>
<td>Double Square</td>
<td>33.3ns</td>
<td>34.5ns</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 7: Final Propagation Delays

Simulation outputs for the three test ran are given in figures 43, 44, and 45 which correspond to the matrices given in figures 27 (with 2->4 added), 29, and 28, respectively. The 1000ns clock period was used during these tests also. In the following section, the minimum clock period will be derived and used.

11.3 Minimum Clock Period:

The minimum clock period needs to assimilate the propagation delays of the processors along with the rise and fall times of
Figure 44. Simulation for test #3 on 4x4 array with wiring delays and back annotation added.

Figure 45. Simulation for test #2 on 4x4 array with wiring delays and back annotation added.
the clock signal. The one state of the clock should be at least 40ns with a fall time of 20ns to completely cover the maximum propagation delay of the processors and the new fall time. Likewise, the zero state should be at least 30ns, minimum allowed is 29ns (Heinbuch 1988), with a 20ns rise time. This would yield a 110ns period, refer to figure 49. These values will ensure the proper operation of the array. Simulations were run with a period of 120ns, the zero state was increased to 40ns, see figures 46 to 48.

11.4 Data Timing Requirements:

The clock signal has been skewed by the input pad and drive buffer. This requires the data to be placed on the lines at a certain period. From examination of figure 49, one can see that the best place is 20ns after the zero to one transition.
Figure 46. Simulation for modified test #1 on 4x4 array at maximum frequency (8.3 MHz)

Figure 47. Simulation for test #3 on 4x4 array at maximum frequency (8.3 MHz)
Figure 48. Simulation output for test #2 on 4x4 array at maximum frequency (8.3 MHz)
Figure 49. Clock wave form after skewing
Fabrication and Future Endeavors

12.0 GDS2 OUTPUT:

After fully simulating the systolic array and verifying proper functioning, the final steps were taken to obtain a file to have the chip fabricated. Steps 19 through 22 of figure 33 were completed producing the chip layout in figure 50 and the GDS2_OUTPUT file for fabrication.

13.0 Future Endeavors:

There are several more steps which could be completed on this thesis.

13.1 Fabrication and Testing:

Ideally the chip would be fabricated and tested. The test vectors that could be used are those that have already been presented along with matrices of all 0’s and all 1’s. These additional matrices would test if the R registers are stuck at either 1 or 0.

13.2 Three Chip Design:

The systolic array could be expanded to use three chips with
Figure 50. Chip layout of 4x4 systolic array
Fabrication and Future Endeavors

each chip enclosing a 4x4 section of a larger array. This would allow any array size to be built up. The first chip would have all the processors except for the rightmost, double square processors. The second chip will be an array of only square processors. The third chip will be an array with the leftmost circle processors removed.
14.0 Conclusion:

Starting from the theory of the Algebraic Path Problem, this thesis has presented a general algorithm by Robert and Trystram (1986) for its solution. The approach and operation of the algorithm was explained and clarified which led to the discussion of the specific systolic array implementation. From this systolic array, a single instance for the APP, the transitive and reflexive closure of a binary relation, was designed and laid out for a CMOS3 standard cell chip design. A minimum clock period of 120ns or maximum frequency of 8.3MHz was determined. The chip is ready to be fabricated, tested and used.
REFERENCES


Appendix A

Computer Program to Simulate Systolic Array

1 Print%=0 :REMark Flag to output to printer
2 StepThrough%=0 :REMark Flag to print individual steps
3 Algo%=0 :REMark Flag to pick which algorithm
4 :REMark 0 circuit, 1 program implem.
5 Size%=4 :REMark Array size
6 DATA 1,1,1,1 :REMark array
7 DATA 0,0,0,0
8 DATA 0,0,0,0
9 DATA 1,0,0,0

10 REMark Initialization
11 DIM Input%(Size%-1,Size%-1), Ain%(Size%-1,Size%),
Bout%(Size%-1,Size%), R%(Size%-1,Size%),
Init%(Size%-1,Size%), DataStreamIn%(3*Size%-2,Size%-1),
DataStreamOut%(2*Size%-2,Size%-1), II%(Size%-1,Size%-1)
12 DIM DAIN%(Size%-1,Size%), DOUT%(Size%-1,Size%),
DBIN%(Size%-1,Size%), DBOUT%(Size%-1,Size%)
13 IF Print% THEN OPEN#3,prt
14 RESTORE 6
15 :REMark Setup matrix input stream
16 FOR i=0 TO Size%-1
17 FOR j=0 TO Size%-1
18 IF i=j THEN
19 II%(i,j)=1
20 ELSE
21 II%(i,j)=0
22 END IF
23 READ temp%
24 Input%(i,j)=temp%
25 END FOR j
26 END FOR i
27 SetUpDataStream
28 PrintDataStreamIn
29 Systolic
30 PrintDataStreamOut
31 IF Print% THEN CLOSE#3
32 STOP :REMark Program end
33 :
34 : REMark Sets up the input data stream
35 :
36 DEFINE PROCEDURE SetUpDataStream
37 LOCAL i,j
38 : REMark Insert NIL's
39 FOR i=0 TO Size%-2
40 FOR j=0 TO Size%-1
41 DataStreamIn%(i,j)=-1

- 84 -
Appendix A

```
42     DataStreamIn%(2*Size%+i,j)=-1
43     END FOR j
44   END FOR i
45   :  REMark Insert Data & I matrix
46   FOR i=0 TO Size%-1
47     FOR j=0 TO Size%-1
48     DataStreamIn%(i+j,j)=Input%(j,i)
49     DataStreamIn%(i+j+Size%,j)=II%(i,j)
50   END FOR j
51   END FOR i
52 END DEFINE SetUpDataStream
53 :
54 DEFINE PROCEDURE PrintDataStreamIn
55  LOCAL i,j
56   CLS#1
57   FOR i=0 TO 3*Size%-2
58     FOR j=0 TO Size%-1
59       IF Print% THEN PRINT#3,TO j*3;DataStreamIn%(i,j);
60       AT#1,i,j*3:PRINT #1,DataStreamIn%(i,j)
61     END FOR j
62     IF Print% THEN PRINT#3
63   END FOR i
64   IF Print% THEN PRINT#3,\\
65 END DEFINE PrintDataStreamIn
66 :
67 :
68 DEFINE PROCEDURE PrintDataStreamOut
69  LOCAL i,j
70   CLS#1
71   FOR i=0 TO 2*Size%-2
72     FOR j=0 TO Size%-1
73       IF Print% THEN PRINT#3,TO j*3;DataStreamOut%(i,j);
74       AT#1,i,j*3:PRINT #1,DataStreamOut%(i,j)
75     END FOR j
76     IF Print% THEN PRINT#3
77   END FOR i
78 END DEFINE PrintDataStreamOut
79 :
80   : REMark Performs the complete array computations
81 :
82 DEFINE PROCEDURE Systolic
83  LOCAL Time%,i,j
84   Time%=0
85   SetUpCells
86   REPeat Time%
87     TransferOutIn Time%
88     IF StepThrough% THEN
```
Appendix A

90    DisplayArray
91    PAUSE
92    END IF
93    FOR i=0 TO Size%-1
94    FOR j=0 TO Size%
95      UpdateCell i,j
96    END FOR j
97    END FOR i
98    Time%=Time%+1
99    IF Time%>5*Size%-2 THEN EXIT Time%
100   END REPeat Time%
101   END DEFINe Systolic
102 :
103 :
104 DEFINe PROCedure TransferOutIn(T%)
105   LOCal i,j
106   FOR j=0 TO Size%
107     IF j<>Size% THEN
108      IF T%<=3*Size%-2 THEN
109        IF DataStreamIn%(T%,j)=-1 THEN
110          Ain%(0,j)=0:DAin%(0,j)=0
111        ELSE
112          Ain%(0,j)=DataStreamIn%(T%,j):DAin%(0,j)=1
113        END IF
114      ELSE
115        Ain%(0,j)=0:DAin%(0,j)=0
116      END IF
117     END IF
118     IF j THEN
119       Bin%(0,j)=Bout%(0,j-1)
120       DBin%(0,j)=DBout%(0,j-1)
121     END IF
122   END FOR j
123   FOR i=1 TO Size%-2
124     FOR j=0 TO Size%
125       IF j<>Size% THEN
126         Ain%(i,j) =Aout%(i-1,j+1)
127         DAin%(i,j)=DAout%(i-1,j+1)
128       END IF
129     IF j THEN
130       Bin%(i,j) =Bout%(i,j-1)
131       DBin%(i,j)=DBout%(i,j-1)
132     END IF
133   END FOR j
134   END FOR i
135   FOR j=0 TO Size%
136     IF j<>Size% THEN
137       Ain%(Size%-1,j) =Aout%(Size%-2,j+1)

DAin%(Size%-1,j)=DAout%(Size%-2,j+1)
END IF
IF j THEN
Bin%(Size%-1,j) =Bout%(Size%-1,j-1)
DBin%(Size%-1,j)=DBout%(Size%-1,j-1)
IF T%>=3*Size% AND T%<5*Size%-2
DataStreamOut%(T%-3*Size%,j-1) =
Aout%(Size%-1,j)*((DAout%(Size%-1,j)=1) -
(DAout%(Size%-1,j)=0)
END IF
END IF
END FOR j
END DEFine TransferOutIn

DEFine PROCedure SetUpCells
LOCAL i,j
FOR i=0 TO Size%-1
FOR j=0 TO Size%
Ain%(i,j) =0 :Aout%(i,j) =0
DAin%(i,j)=0 :DAout%(i,j)=0
Bin%(i,j) =0 :Bout%(i,j) =0
DBin%(i,j)=0 :DBout%(i,j)=0
Init%(i,j)=1 :R%(i,j) =0
END FOR j
END FOR i
END DEFine SetUpCells

DEFine PROCedure UpdateCell(X%,Y%)
LOCAL y
y=Y%
SELECT ON y
ON y=0
Circles X%,Y%
ON y=1 TO Size%-1
Square X%,Y%
ON y=Size%
Dsquare X%,Y%
END SELECT
END DEFine UpdateCell

DEFine PROCedure Circles(X%,Y%)
IF Algo% THEN
IF DAin%(X%,Y%) THEN
IF Init%(X%,Y%) THEN
Bout%(X%,Y%)=1
Appendix A

```plaintext
184       DBout%(X%, Y%) = 1
185       Init%(X%, Y%) = 0
186       ELSE
187       Bout%(X%, Y%) = Ain%(X%, Y%)
188       DBout%(X%, Y%) = 1
189       END IF
190       ELSE
191       DAout%(X%, Y%) = 0; DBout%(X%, Y%) = 0
192       END IF
193       ELSE
194       Bout%(X%, Y%) = Ain%(X%, Y%) \& Init%(X%, Y%)
195       DBout%(X%, Y%) = DAin%(X%, Y%)
196       Init%(X%, Y%) = NOT(DAin%(X%, Y%)) \& Init%(X%, Y%)
197       END IF
198       END DEFINE Circles
199 :
200 :
201 DEFINE PROCEDURE Dsquare(X%, Y%)
202     IF Algo% THEN
203         IF DBin%(X%, Y%) THEN
204             IF Init%(X%, Y%) THEN
205                 R%(X%, Y%) = Bin%(X%, Y%)
206                 Init%(X%, Y%) = 0
207                 Aout%(X%, Y%) = 0
208                 DAout%(X%, Y%) = 0
209             ELSE
210                 Aout%(X%, Y%) = R%(X%, Y%) \& Bin%(X%, Y%)
211             END IF
212         ELSE
213             DAout%(X%, Y%) = 0; DBout%(X%, Y%) = 0
214         END IF
215     END ELSE
216     DAout%(X%, Y%) = 0; DBout%(X%, Y%) = 0
217     END IF
218     DAout%(X%, Y%) = NOT(Init%(X%, Y%)) \& DBin%(X%, Y%)
219     R%(X%, Y%) = (Init%(X%, Y%) \&& DBin%(X%, Y%)) \&
                  (NOT(Init%(X%, Y%)) \&& R%(X%, Y%))
220     Init%(X%, Y%) = Init%(X%, Y%) \& NOT(DBin%(X%, Y%))
221     END IF
222     END DEFINE Dsquare
223 DEFINE PROCEDURE Square(X%, Y%)
224     IF Algo% THEN
225         IF DBin%(X%, Y%) AND DAin%(X%, Y%) THEN
226             IF Init%(X%, Y%) THEN
227                 R%(X%, Y%) = Bin%(X%, Y%) \&& Ain%(X%, Y%)
228                 Init%(X%, Y%) = 0
229             Aout%(X%, Y%) = 0; DAout%(X%, Y%) = 0
230             Bout%(X%, Y%) = Bin%(X%, Y%); DBout%(X%, Y%) = 1
```
Appendix A

231 ELSE
232    Aout%(X%,Y%) = Ain%(X%,Y%) && R%(X%,Y%) && Bin%(X%,Y%)
233    DAout%(X%,Y%)=1
234    Bout%(X%,Y%)=Bin%(X%,Y%): DBout%(X%,Y%)=1
235 END IF
236 ELSE
237    DBout%(X%,Y%)=0: DAout%(X%,Y%)=0
238 END IF
239 ELSE
240    Aout%(X%,Y%)=(Bin%(X%,Y%)&&R%(X%,Y%)) && Ain%(X%,Y%)
241    DAout%(X%,Y%)=NOT(Init%(X%,Y%)) && DAin%(X%,Y%)
242    DBout%(X%,Y%)=DBin%(X%,Y%)
243    Bout%(X%,Y%)=Bin%(X%,Y%)
244    R%(X%,Y%) = (Init%(X%,Y%) & & DBin%(X%,Y%) & &
              DAin%(X%,Y%) & & Ain%(X%,Y%) & &
              Bin%(X%,Y%)) || NOT(Init%(X%,Y%)) &&
              R%(X%,Y%))
245    Init%(X%,Y%) = Init%(X%,Y%) && (NOT(DBin%(X%,Y%)) ||
                 NOT(DAin%(X%,Y%)))
246 END IF
247 END DEFINE Square
248 :
249 :
250 DEFINE PROCEDURE DisplayArray
251 LOCAL i,j
252 CLS#1
253 FOR j=0 TO Size%-1
254    INK#1,4*3*DAin%(0,j)
255    AT#1,0,10*j:PRINT#1,'A:';Ain%(0,j);DAin%(0,j)
256 END FOR j
257 FOR i=0 TO Size%-1
258    FOR j=0 TO Size%
259        INK#1,4*3*DAout%(i,j)
260        AT#1,5*i+2,10*j:
261        PRINT#1,'A:';Aout%(i,j);DAout%(i,j)
262        INK#1,4*3*DBout%(i,j)
263        AT#1,5*i+3,10*j:
264        PRINT#1,'B:';Bout%(i,j);DBout%(i,j)
265    END FOR j
266    INK#1,4*3*Init%(i,j)
267    AT#1,5*i+4,10*j:PRINT#1,'R:';R%(i,j)
268    END FOR i
269 FOR j=1 TO Size%
270    INK#1,4*3*DAout%(Size%-1,j)
271    AT#1,5*(Size%)+3,10*j:
PRINT#1,'A:';Aout%(Size%-1,j);DAout%(Size%-1,j)
272   END FOR j
273   INK#1,7
274   PRINT#1,'Time: ';Time%
275 END DEFINE DisplayArray