ROBUST MATRIX-MULTIPLICATION\textsuperscript{1}
ALGORITHMS FOR VLSI

Peter J. Varman, I. V. Ramakrishnan
and Donald Fussell

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-221 March 1983

\textsuperscript{1}This research was partially supported by NSF Grant Numbers MCS 81-09489 and MCS 81-04017.
Index Terms

VLSI, Fault Tolerance, Systolic Algorithms, Matrix Multiplication, Reconfigurability, Wafer-Scale Integration.
Abstract

Parallel matrix-multiplication algorithms have been proposed for execution on a rectangular or hexagonal mesh of identical processors on a single VLSI chip. These algorithms impose stringent interconnection requirements between processors of the host network. The occurrence of production faults during the manufacturing process disrupts the data paths and may cause the algorithm to fail even if a sufficient number of non-faulty processors are available.

This paper presents a systolic algorithm for matrix multiplication that is robust in the face of production flaws. Each processor consists of an "Inner-Product" step computing unit and shift registers. By appropriate interconnection of the shift registers, the processors of the host network are configured so that all non-faulty processors accessible from the I/O port can be utilized. Irrespective of the structure of the fault-free component obtained due to the random fault patterns, the behaviour at the I/O port, through which all external communication occurs, is unchanged. The I/O bandwidth is independent of the problem size and the multiplication of two \( n \times n \) matrices requires \( O(n) \) processors and has a time complexity of \( O(n^2) \) cycles, bounded by the I/O bandwidth.
1 INTRODUCTION

Advancements in integrated-circuit technology have stimulated research on designing special-purpose computing devices to solve specific problems. Systolic Arrays [10] were proposed by Kung and Leiserson as an attractive alternative for solving compute-bound problems [9], and several algorithms based on this concept have since been discovered [3,5,6,8,11]. The attractive features of the algorithms are the regular data-flow requirements and simple control structures for individual processors that make them particularly suitable for VLSI implementation.

In this paper we present an algorithm for multiplying two nxn matrices in a systolic manner, using (3n-2) processors. Each processor is composed of shift registers and a simple "inner-product-step" computing unit. Interconnection of shift registers between processors provides the data flow between the computing units.

One processor serves as the port for the network and all input/output with the external world occurs via the port. The I/O bandwidth is a constant, independent of the size of the problem, differing from earlier solutions which required an I/O bandwidth proportional to n [8]. The time complexity for the computation is $O(n^2)$ cycles, which is a lower bound for a fixed-bandwidth system.

A novel feature of this algorithm is its fault-tolerant capability. Specifically, the algorithm operates on any connected component of processors of the required size that is dictated by the random fault patterns. In contrast, the earlier proposed solutions [8] necessitate stringent interconnection requirements between processors of the host network. The occurrence of production faults during the manufacturing process disrupts the data paths required by the computation, either causing the algorithm to fail or severely limiting its applicability. Since the probability of a fatal
production flaw increases exponentially with the size of the circuit on the chip [13], building a sufficiently large matrix-multiplication array using the rectangular or hexagonal mesh algorithms becomes infeasible.

Several approaches to the problem of dealing with faults in VLSI arrays have been considered previously [2,4,7,12], the basic idea being to extract a fault-free array (linear or rectangular) from a larger mesh (rectangular or hexagonal) with randomly distributed faults. An attempt to extract a fault-free rectangular mesh from a larger rectangular mesh was shown by [12] to result in the waste of a large number of non-faulty processors. Also the solution in [7] required every processor on the row and column of a faulty processor to act as a switch to connect together non-faulty regions of the chip. Since O(n) processors are wasted per faulty processor, a large number of non-faulty processors go unutilized using this approach thereby precluding a straightforward method of achieving fault-tolerant implementations of the mesh matrix-multiplication algorithms. In [2,4], solutions for growing a linear array in a mesh with faults were presented, and it was argued in [4] that the semantics of the algorithm to be mapped onto the chip could be exploited to design robust implementations.

Fault tolerance is achieved by the algorithm presented here, due to its ability to execute efficiently on any connected set of processors. No complicated addressing schemes are needed to route data, and the algorithm does NOT change depending on the topology of the fault-free component.

In section 2, the host network on which the algorithm will operate is described. Sections 3 and 4 contain the algorithm description and its correctness proof respectively, and the tradeoffs involved in its implementation are discussed in Section 5.
2 NETWORK MODEL

The underlying host network is some nearest-neighbor mesh-network, like the rectangular or hexagonal mesh which are particularly suited for VLSI implementation. One node (processor) in the mesh serves as the I/O port.

During the manufacturing process, some subset of the nodes in the mesh will be faulty. We focus our attention on the largest connected component of non-faulty nodes that includes the I/O port. In [4], an algorithm for growing a spanning tree in an arbitrary connected component was presented. The algorithm operates in time proportional to the number of nodes in the component and establishes a path from the I/O port to every fault-free node in the component.

Depending on the mesh and the fault distribution, trees with arbitrary structure may be grown. The idea of a robust algorithm is to ensure correct and efficient execution irrespective of the actual tree structure.

A conceptual model of a processor $P_i$ is shown in Figure 1. PE is a processing element that performs an inner-product-step computation using the elements at its input ports, $I_A^i$, $I_B^i$ and $I_C^i$ and places the results at the corresponding output ports $O_A^i$, $O_B^i$ and $O_C^i$. The computation by the PE in $P_i$ is specified by:

$$O_A^i := I_A^i; \quad O_B^i := I_B^i; \quad O_C^i := I_C^i + I_A^i \cdot I_B^i.$$  

$a_i$, $b_i$ and $c_i$ are "forward" buffers used by the elements of the 'A', 'B' and 'C' matrices respectively. The "reverse" buffer for the 'A' matrix elements is $A_i$ and that for the 'C' matrix is the array $C_i[1..k]$, $k=2n+1$. $A_i$, $b_i$ and $C_i[1]$ are connected to the ports $O_A^i$, $I_B^i$ and $O_C^i$ respectively.

Consider an N-node rooted tree. The nodes represent processors with the root serving as the I/O port. The edges of the tree represent communication
links between adjacent nodes.

Each node has an index \('j\)', equal to that obtained by some depth-first search [1] of the tree starting from the root. (The node with index \('j\)' will be called \(P_j\)). The depth-first traversal also defines a father-son relationship between any pair of adjacent nodes. (Fig. 2).

Let \(P_i\) be the father of \(P_j\) and let the sons of \(P_j\) be \(P_{j_1}, P_{j_2}, \ldots, P_{j_r}\), with \(j_1 > j_2 > \ldots > j_r = j+1\). Note that \(r=0\) implies that \(P_j\) is a leaf node in the tree. The interconnection of buffers at each \(P_j\), \(j=1, \ldots, N\) is given in Table 1. Figure 3 illustrates the interconnection for a particular seven node tree.

The operation of the machine is as follows. The elements of the matrices \('A', 'B'\) and \('C'\) (initially \(c_{ij}^0 = 0\), for \(i,j=1,\ldots,n\)) are fed into the buffers \(a_1, b_1\) and \(c_1\) of the I/O port. At every clock cycle all the elements move forward to the next buffer in its path. Elements that are at the input ports of PEs are transformed and their new values clocked into the buffers connected to the output ports. The elements are updated and transferred from the input to the output ports in one cycle.

3 ALGORITHM

In this section the algorithm to compute \(C = A \times B\), where \(A\), \(B\) and \(C\) are \(n \times n\) matrices, on a tree of \(N = 3n-2\) processors is described.

The values of the matrix \(C\) are computed using the following recurrence:

\[
c_{ij}^0 = 0
\]

\[
c_{ij}^{k+1} = c_{ij}^k + a_{ik} \times b_{kj}, \quad k=0, \ldots, n-1, \quad i,j=1, \ldots, n.
\]

All the entries of the matrix \(C\) are initialized to 0 before being pumped
into the machine. The initial and final values of $c_{ij}$ will be referred to as $c_{ij}^0$ and $c_{ij}^n$ respectively.

Let the time at which $c_{ij}^0$ is fed into $c_1$ be $\emptyset$. The algorithm (including initialization of the array and extraction of the results) is given in Algorithm 1.

**Algorithm 1**

1. Pump $c_{ij}^0$ (value $\emptyset$) into $c_1$ at time $t = 2n(i+j-2) + 2(i-1)$
2. Pump $b_{ij}$ into $b_1$ at time $t = 4(n^2-1) + 2(n+1)(i-1) + 2(j-1)$
3. Pump $\emptyset$ into $a_1$ for all times $\emptyset \leq t < 4(n(n-1))$ (which is the time when $a_{11}$ is pumped into $a_1$), and for all times $t > 2(n-1)(3n+1)$ (which is the time when $a_{nn}$ is pumped into $a_1$).
4. Pump $a_{ij}$ into $a_1$ at time $t = 2n(2n-3) + 2(n^2+i-1)$
5. Extract $c_{ij}^n$ from $C_1[k]$ at time $t = 2(3n-2)(n+1) + 2n(i+j-2) + 2(i-1)$.

**4 PROOF OF CORRECTNESS**

Definition: A processor $P_k$ has a distance equal to $'r'$ if there are exactly $'r'$ edges between $P_1$ and $P_k$, in the tree.

Lemma 1.1: Suppose $P_k$ has a distance $'r'$. If $t_a, t_b, t_c$ are the times at which elements are pumped into $a_1, b_1$ and $c_1$ respectively, then the times $t_A, t_B$ and $t_C$ at which they reach $I_A^k, I_B^k$ and $I_C^k$ respectively, are given by:

1. $t_A = t_a + 2(3n-k-2) + r$
2. $t_B = t_b + r$
3. $t_C = t_c + 2(3n-k-2)(n+1) + r$

Proof:

(1): Let the element $'x'$ be pumped into $a_1$ at time $t_a$. From the tree interconnection, it can be seen that $x$ moves through the buffers $A_j$,.
j = 3n-2, 3n-3, ..., k+2, k+1 before reaching $i_A^k$. This involves a delay of (3n-k-2).

For every $A_j$, j = 3n-2, ..., k+1 that 'x' traverses, it has to move through the corresponding buffer $a_j$, incurring an extra delay of (3n-k-2).

Also the 'r' edges between $P_1$ and $P_k$ are traversed exactly once, via the $a_1$ buffers, involving an additional delay of (r).

Summing the delays: $t_A = t_a + 2(3n-k-2) + r$.

(2): Let the element 'x' be pumped into $b_1$ at time $t_b$. From the interconnection it can be seen that one copy of 'x' moves directly along the 'r' $b_1$ buffers separating $P_1$ from $P_k$.

Since there is a unit delay through each $b_1$, $t_B = t_b + r$.

(3): The same argument as for (1), except that the the delay in traversing $C_j[1] \ldots C_j[2n+1]$ is (2n+1).

Therefore, $t_C = t_c + (3n-k-2) + (3n-k-2)(2n+1) + r$.

$t_C = t_c + 2(3n-k-2)(n+1) + r$.

**Lemma 1.2:** For any i, j the initial value $c_{ij}^0$ of $c_{ij}$, remains unchanged as it travels from $c_1$ till it reaches the input-port $i_C^k$ of $P_k$, where $k = n + i + j - 2$.

**Proof:**
Suppose $P_s$ has distance 'r', and $n+i+j-1 \leq s \leq 3n-2$.

Let $t_C$ be the time when $c_{ij}$ reaches $i_C^8$. Since $c_{ij}$ was pumped into $c_1$ at $t_C = 2n(i+j-2) + 2(i-1)$ (Algorithm 1), from Lemma 1.1
\[ t_C = 2n(i+j-2) + 2(i-1) + 2(3n-s-2)(n+1) + r. \]

Let the element at \( r_A^s \) at \( t_C \) be \( 'x' \), and \( t_a \) be the time at which \( 'x' \) was pumped into \( a_1 \). From Lemma 1.1

\[ t_C = t_a + 2(3n-s-2) + r. \]

If \( x=\emptyset \) then the computation at \( P_s \) will leave \( c_{ij} \) unchanged. From Algorithm 1, if \( t_a < 4n(n-1) \) then \( x=\emptyset \). Hence, we must show that

\[ t_a = t_C - 2(3n-s-2) - r < 4n(n-1) \]

Substituting for \( t_C \) and reducing we require to show that

\[ s > (n+i+j-2) + (i-1)/n. \]

Since

\[ n + (i+j-1) > (n+i+j-2) + (i-1)/n \]

for \( i \leq n \), and

\[ s > n + (i+j-1), \]

the result follows.

**Lemma 1.3:** For any \( i,j \), the final value \( c_{ij}^n \) of \( c_{ij} \) remains unchanged as it travels from the Output-Port \( O_C^k \) of \( P_k \), where \( k = i+j-1 \), till it reaches \( C_i[k] \).

**Proof:**

Suppose \( P_s \) has distance \( 'r' \) and \( 1 < s < i+j-2 \). Let \( t_C \) be the time when \( c_{ij} \) reaches \( r_A^s \). Let \( 'x' \) be the element at \( r_A^s \) at \( t_C \), and suppose \( 'x' \) was pumped into \( a_1 \) at \( t_a \). Then

\[ t_C = t_a + 2(3n-s-2) + r. \]

If \( t_a > 2(n-1)(3n+1) \) then \( x=\emptyset \) (Algorithm 1), and \( c_{ij} \) will remain unchanged at \( P_s \). Hence, we must show that

\[ t_C - 2(3n-s-2) - r > 2(n-1)(3n+1). \]

Substituting for \( t_C \) and reducing requires that

\[ s > i+j-2 + i/n \]
Since \(i+j-1/n > i+j-2\) for \(i>0\), and \(s \leq i+j-2\), the result follows.

**Lemma 1.4**: For any \(i, j\) and for any \(k, 1 \leq k \leq n, a_{ik}, b_{kj} c_{ij}\) reach the input-ports \(I^A_i, I^B_i\) and \(I^C_i\) of \(P_S, s = n+(i+j)-(k+1)\), at the same time.

**Proof:**
Suppose \(P_S\) has distance 'r'. Let \(t_A, t_B\) and \(t_C\) denote the times at which \(a_{ik}, b_{kj}\) and \(c_{ij}\) reach \(I^A_i, I^B_i\) and \(I^C_i\) respectively. From Lemma 1.1 and Algorithm 1,

\[
t_A = 2n(2n-3) + 2(nk+i-1) + 2(3n-s-2) + r.
\]

\[
t_B = 4(n^2-1) + 2(n+1)(k-1) - 2(j-1) + r.
\]

\[
t_C = 2n(i+j-2) + 2(i-1) + 2(3n-s-2)(n+1) + r.
\]

Each of these expressions (which is the sum of the time the element entered the array and the delay) reduces to:

\[4n^2 + 2nk - 2n - 2j + 2k + r = 4.
\]

Hence, \(t_A = t_B = t_C\).

**Lemma 1.5**: For any \(k, 1 \leq k \leq n\), the value of \(c_{ij}\) at \(O^C\) is \(\sum_{m=1}^{k} a_{im} * b_{mj}\), where \(s = n+(i+j)-(k+1)\).

**Proof:**
(By induction).

**Base-Step**: \(k = 1\)

From Lemma 1.4, \(a_{i1}, b_{1j}\) and \(c_{ij}\) arrive at the same time at \(I^A_i, I^B_i\) and \(I^C_i\), where \(d = n+i+j-2\). By Lemma 1.2, the value of \(c_{ij}\) at \(I^C_i\) is \(\phi\), and hence the value at \(O^C\) is \(a_{i1} * b_{1j}\). Thus the lemma holds for the base step.

**Induction Step**: Assume the lemma holds for some \(k, 1 \leq k < n\). By Lemma 1.4, \(a_{ik+1}, b_{k+1j}\) and \(c_{ij}\) arrive at the same time at \(I^A_i, I^B_i\) and \(I^C_i\), where \(d = n+(i+j)-(k+2)\).
By the Induction Hypothesis, the value of \( c_{ij} \) at \( O_C^d \) is \( \sum_{m=1}^{k} a_{im} * b_{mj} \) and this is the value at \( I_C^d \) (interconnection structure).

Thus the value of \( c_{ij} \) at \( O_C^d \) is \( \sum_{m=1}^{k+1} a_{im} * b_{mj} + a_{ik+1} * b_{k+1j} = \sum_{m=1}^{k+1} a_{im} * b_{mj} \). Hence, the lemma holds for \( k+1 \).

**Theorem 1:**

\[ c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj} \] when it leaves \( C_1[2n+1] \).

**Proof:**

From Lemmas 1.3 and 1.5.

**Theorem 2:**

The time complexity of the algorithm is \( O(n^2) \).

**Proof:**

The first value \( c_{11} \) is pumped at \( t=0 \). The final value \( c_{nn} \) is extracted at \( t = 2(3n-2)(n+1) + 2(n-1)(2n+1) \). Hence the time complexity is \( O(n^2) \).

An example of a 3x3 matrix multiplication on the network of Fig. 3 is illustrated in Fig. 4.

5 DISCUSSION

The processor model used in describing the algorithm was largely for purposes of clarity. In an actual implementation some optimizations of memory and time can be achieved as discussed below.

The C-buffers of size \( 2n+1 \) implement both the delay and storage for intermediate \( c_{ij} \) values in a processor. Analysis of the algorithm indicates that no more than \( n \) of the \( c_{ij} \) values are ever present in any one processor's C-buffer at any time. By incorporating a limited amount of decision-making ability in each processor the C-buffer can be replaced by local memory of size
n.

In determining the time complexity of the algorithm it was assumed that the time for an inner-product-step computation was equal to the rate at which data items could be made available. However, by building each PE as a k-stage pipelined unit (where k is the ratio of the inner-product-step computation time to the data-access time) the total time for the algorithm is still $O(n^2)$ cycles, where a cycle is now the data-access time fixed by the supportable I/O bandwidth.

Finally, explicit initialization of the array achieved by pumping 0 into $a_1$ for $t < 4n(n-1)$, can be dispensed with if each processor can initialise its c and C buffers to zero on starting. The only change in the algorithm is that $4n(n-1)$ is subtracted from the time each element is pumped into or extracted from the array. A continuous stream of zeroes is fed into $c_1$ starting from $t=0$, which is now the time when $a_{11}$ is pumped into the array. This reduces the constant factor in the dominant term for the time complexity of the algorithm by a factor of 3/5.

In summary we have presented a systolic algorithm for multiplication of two $n \times n$ matrices using $O(n)$ simple processors in time $O(n^2)$. (The algorithm is easily extendible to non-square matrices). A unique feature of the algorithm is its ability to execute efficiently on any connected component of processors of the required size, thus providing it with fault-tolerant capability. It should therefore be possible to build a large wafer-sized array of processors and utilize the non-faulty processors for the computation, thereby achieving an implementation that is robust in the face of production flaws.
References


<table>
<thead>
<tr>
<th>CONNECT TO</th>
<th>a_j</th>
<th>c_j</th>
<th>I_A^j</th>
<th>I_C^j</th>
<th>O_B^j</th>
<th>A_j_s</th>
<th>c_j_s[k]</th>
</tr>
</thead>
<tbody>
<tr>
<td>r ≥ 1</td>
<td>a_j</td>
<td>c_j</td>
<td>A_j_r</td>
<td>C_j_r[k]</td>
<td>b_j_s</td>
<td>a_j_s+1</td>
<td>c_j_s+1</td>
</tr>
<tr>
<td></td>
<td>s=1,..,r</td>
<td></td>
<td>s=1,..,r-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r = 0</td>
<td>a_j</td>
<td>c_j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1
Figure 1: A Processor

Figure 2: Numbering by Depth-First Search
Figure 3: Interconnection for the seven-node tree of Figure 2.
Fig. 4: Example of 3x3 multiplication on the network of Fig. 3.

\[ A = [a_{ij}]_{3 \times 3} \quad B = [b_{ij}]_{3 \times 3} \quad C = [c_{ij}]_{3 \times 3} = A \times B \]

The time instants at which elements are pumped into and extracted from the port are given below (from Algorithm 1).

<table>
<thead>
<tr>
<th>( j )</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>24</td>
<td>30</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>26</td>
<td>32</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>28</td>
<td>34</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( j )</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>30</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>38</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>46</td>
<td>44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( j )</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>14</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>22</td>
<td>28</td>
</tr>
</tbody>
</table>

Pump 0 into \( a_1 \) for \( t < 24 \) and \( t > 40 \). (In the \( C \) Table, the top value is the input and the bottom value the output time for each \( c_{ij} \) value).

We shall trace the computation of \( c_{22} \) as illustration.

- Column 1 is the time instant of interest,
- Column 2 the location of \( c_{22} \) at that time
- Column 3 the value of \( c_{22} \)
- Column 4 contains the index of the processor at whose input port \( c_{22} \) is present at that instant, and
- Columns 5 and 6 are the values at the input ports \( I_A \) and \( I_B \) respectively, of the corresponding processor. Starred time instants are those at which \( c_{22} \) passes through a PE and has its value updated by an inner-product-step computation.
<table>
<thead>
<tr>
<th>t</th>
<th>$c_{22}$ present at</th>
<th>value</th>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>a₂₁</th>
<th>b₁₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>c₁</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>c₁</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>c₁</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>c₂</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>c₃</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>c₅</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>c₆</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>c₆</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>c₅</td>
<td>0</td>
<td>5</td>
<td>a₂₁</td>
<td>b₁₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>c₅</td>
<td>0</td>
<td>5</td>
<td>a₂₁</td>
<td>b₁₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>c₁</td>
<td>c₂₂</td>
<td>4</td>
<td>a₂₂</td>
<td>b₂₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>c₁</td>
<td>c₂₂</td>
<td>4</td>
<td>a₂₂</td>
<td>b₂₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>c₃</td>
<td>c₂₂</td>
<td>3</td>
<td>a₂₃</td>
<td>b₃₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>c₃</td>
<td>c₂₂</td>
<td>3</td>
<td>a₂₃</td>
<td>b₃₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>c₂</td>
<td>c₂₂</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>c₂</td>
<td>c₂₂</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>c₁</td>
<td>c₂₂</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>c₁</td>
<td>c₂₂</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extract $c_{22}$ at $t=70$. 