MONTANA STATE UNIVERSITY

Heuristic interactive synthesis of cellular arrays : based on Polish notation
by George Albert Berges

A thesis submitted to the Graduate Faculty in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY in Electrical Engineering
Montana State University
© Copyright by George Albert Berges (1970)

Abstract:
The object of the research' contained in this thesis is to develop a workable design tool for embedding arbitrary Boolean switching functions in a class of cellular arrays. Such a system is produced in an interactive mode, using a heuristic approach, and based upon the notational system called Polish notation.

A correspondence is developed between arbitrary Polish notation representations of Boolean functions and a class of cellular arrays. Heuristic manipulative porcedures for Polish strings are developed. These procedures are applied to reduce the Polish string to a form corresponding to a near minimal number of cells in the array realizing the function. Options available to the designer are presented, with the purpose of further reducing the array, using the gestalt forming abilities of the human operator. The class of arrays is shown to be inadequate for all switching problem formulations and a new class of arrays, the buss arrays, is introduced to overcome this deficiency. An algorithm is developed to embed Polish string representations of Boolean functions in buss arrays. A heuristic interactive approach to applying and modifying this algorithm is presented. Examples are produced, including a cellular multiplier.

© Copyright 1970 by George A. Berges, Jr.
Bozeman, Montana. All rights reserved.
No part of this work may be reproduced in
any form, or by any means, without permis-
sion in writing from the author.

HEURISTIC INTERACTIVE SYNTHESIS OF CELLULAR ARRAYS, BASED ON POLISH NOTATION

by

GEORGE ALBERT BERGES, JR.

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

of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

Approved:

Paul E. Wahlrich
Head, Major Department

Chairman, Examining Committee

K. Goering
Graduate Dean

MONTANA STATE UNIVERSITY
Bozeman, Montana

August, 1970

ACKNOWLEDGMENT

The author wishes to express his undying appreciation to the many people who have offered assistance during the research for and writing of this thesis. In particular, thanks are due to Professor R. C. Minnick without whose support and council as the author's major professor, this effort would have been in vain. Thanks also to Professor Donald A. Rudberg, who struggled through the rough drafts of this work, and to Drs. Paul Uhlrich and Louis Schmittroth, who allowed the author use of the facilities of their departments. The efforts of Phil Winninghoff and Kenneth Trieu in preparing the final copy are appreciated as are those of Cheryl Grote and Margaret Parker. Last and certainly not least, thanks to Vicky, the patient wife, who not only lived with the author through this work, but was instrumental in preparing the final draft.

The author gratefully acknowledges the financial support received from the Department of Electrical Engineering at Montana State University, the United States Air Force Cambridge Research Laboratories under contract AF-19628-67-C-0293, the United States Air Force Advanced Research Projects Agency under contract AF-30-601-4277 (subcontract from the University of Utah), and a National Aeronautics and Space Agency Traineeship.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
	I.1 Scope and Purpose	1
	I.2 Historical Context	3
	I.3 Organization of Remaining Chapters	6
II.	SURVEY OF LITERATURE	7
	II.1 Developments in Cellular Logic	8
	II.1.1 Economics of LSI	8
	II.1.2 Cellular Arrays	10
	II.1.3 Some Types of Cellular Arrays	11
	II.1.4 Cost Measure of Various Cellular Array Types	16
	II.1.5 Known Cellular Synthesis Techniques	17
	II.2 Polish Notation	39
	II.2.1 Evaluation of Polish Strings	40
	II.2.2 Conversion of Infix Strings to Polish Strings	43
	II.3 Heuristic Systems	47
III.	SYNTHESIS OF CELLULAR ARRAYS	50
	III.1 Notation and Definitions	51
	III.1.1 Definition	51
	III.1.2 Notation	54
	III.2 Correspondence of P-strings	56
	III.3 Manipulation of P-strings	82
	III.3.1 Manipulative Procedures	85
	III.3.2 Examples of P-string Manipulation	111
	III.3.3 P-strings for Register Logic	133
	III.3.4 Summary	135

Chapter Page

III.4 Buss Synthesis 138

III.4.1 Buss Arrays 138
III.4.2 Comparison of Buss Arrays and Other Arrays 140
III.4.3 Solution of Iterated Problems with Buss Arrays 143
III.4.4 Constraints Imposed by Iterated Problems 149
III.4.5 An Approach to Embedding Functions in Buss Arrays 156
III.4.6 An Algorithm for Embedding Functions in Buss Arrays 158
III.4.7 An Engineering Example - A Cellular Parallel Multiplier 182
III.4.8 An Interactive, Heuristic Approach to Buss Array Synthesis 218
III.4.9 Summary 229

IV. SUMMARY, CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE RESEARCH 230

IV.1 Summary 230

IV.2 Conclusions 233

IV.3 Future Research 234

BIBLIOGRAPHY 238

LIST OF TABLES

Table		Page
1.	Cutpoint Cell Functions	15
2.	Cost Parameters of General Cellular Arrays	20
3.	Tests for Synthesis Example	26
4.	Polish Strings and Equivalent Strings	40
5.	Evaluation of a Polish String	42
6.	Evaluation Hierarchy for Switching Algebra	44
7.	Example of Translation to Polish Notation	45
8.	Functions of Two Variables	63
9.	Operator Pairs for Complementation of a Portion of a Cascade	65
10.	Manipulative Procedures	86
11.	Discriptive Categorization of Manipulative Procedures	89
12.	Examples of Use of MOVEOP	92
13.	Fundamental Algebraic Laws of ELRULE in P-notation for the Operator Set $\{\times, +, \oplus, '\}$	97
14.	Transformation Used in DNF Algorithm	100
15.	Failure Modes and Recovery Procedure for the Buss Synthesis Algorithm	180
16.	Buss Synthesis Algorithm Modifications	226

LIST OF FIGURES

Figure		Page
1.	A Functional Module	10
2.	A NOR-Based Array	12
3.	A Maitra Cascade	13
4.	A Cutpoint Array	14
5.	A Cobweb Array	16
6.	A Cutpoint Cell	18
7.	A Cobweb Cell	19
8.	Type A Cobweb Array	19
9.	Cascade for Synthesis Example	27
10.	Algorithm for Synthesis of Redundant Maitra Cascades	30
11.	Cutpoint Synthesis Example	33
12.	Cutpoint Realization of Three Functions in One Array	35
13.	Cutpoint Realization of Arbitrary Three-Variable and Four-Variable Functions	36
14.	Necessary Components for Translation to Polish Notation	43
15.	Two-Level Realization of f = uw'x'+uv+u'v'x+wx'+vwx	57
16.	A Cellular Array	58
17.	A Five-Variable Example Cascade	59
18.	Embedding a P-String in a Collector Array	64

Figure		Page
19.	A Style 1 Array	66
20.	A Style 2 Array	68
21.	Embedding Example	73
22.	Embedding Example for a Style 2 Array	74
23.	Synthesis Example for a Style 1 Array	77
24.	Realizations of $f = \Sigma(3,6,7,11,12,13,14,15)$	84
25.	Flow Diagram for EQUAL and COMP	90
26.	Example Realization for Use of ORDER	94
27.	Flow Diagram for PLUSE	99
28.	Flow Diagram for DNF	102
29.	Examples of DNF Procedure	103
30.	Flow Diagram for CONC	107
31.	Flow Diagram for SHANNON	109
32.	The Half Adder	112
33.	A Cellular Half Adder	113
34.	The Full Adder	115
35.	Style 1 Realizations of the Full Adder Carry Function	118
36.	Style 1 Realization of the Full Adder	118
37.	The Double Adder	120
38.	Cellular Array for the Low Order Sum of the Double Adder	121
39.	Cellular Array of the Double Adder	125

Figure		Page
40.	Relationships for Variables in Synthesis Example	131
41.	Style 1 Realization of Example 4	136
42.	Cannon's General Register Cell	137
43.	Style 1 Realization of Cannon's General Register Cell	137
44.	Two Buss Arrays	139
45.	A Style 5 Buss Array	142
46.	Style 4 Realization of Example 4	143
47.	An 18 Bit Adder	144
48.	Cannon's Processor Cell	147
49.	Cannon's Parallel Multiplier	148
50.	Constraints for Buss Synthesis	150
51.	Iterated Two-Bit Adder	155
52.	Flow Diagram for Buss Synthesis	162
53.	Flow Diagram for GETLIT	169
54.	Illustration of the Routing Algorithm	171
55.	Classification of Nodes in Figure 53	172
56.	Corner Criterion for a New Path in the Routing Algorithm	174
57.	A Style 6 Array	184
58.	Functional Decomposition of Cannon's Multiplier	187
59.	Block Diagram of the Cellular Realization of Cannon's Multiplier	188

Figure		Page
60.	Realization of DFF2	189
61.	Realization of DSRFF11	194
62.	Unsuccessful Attempt at Realization of LOGICA	195
63.	Realization of LOGICA	196
64.	Unsuccessful Attempt at Realization of DFF3	198
65.	Realization of DFF3	199
66.	Realization of DSRFF8	202
67.	Realization of ADD6	203
68.	Realization of DTFF9	205
69.	Realization of DTFF10	207
70.	Realization of DTFF11	209
71.	Realization of DSRFF12	210
72.	Realization of DFF4	211
73.	Realization of DTFF12	215
74.	Realization of LOGICB	216
75.	Cellular Realization of Cannon's Parallel Multiplier	217
76.	Algorithmic Realization of Example 4	219
77.	A Fourth Realization of Example 4	220
78.	A Fifth Realization of Example 4	222
79.	A Sixth Realization of Example 4	223
80.	Final Realization of Example 4	224
81.	A Modified Realization of DFF4	225

ABSTRACT

The object of the research contained in this thesis is to develop a workable design tool for embedding arbitrary Boolean switching functions in a class of cellular arrays. Such a system is produced in an interactive mode, using a heuristic approach, and based upon the notational system called Polish notation.

A correspondence is developed between arbitrary Polish notation representations of Boolean functions and a class of cellular arrays. Heuristic manipulative porcedures for Polish strings are developed. These procedures are applied to reduce the Polish string to a form corresponding to a near minimal number of cells in the array realizing the function. Options available to the designer are presented, with the purpose of further reducing the array, using the gestalt forming abilities of the human operator. The class of arrays is shown to be inadequate for all switching problem formulations and a new class of arrays, the buss arrays, is introduced to overcome this deficiency. An algorithm is developed to embed Polish string representations of Boolean functions in buss arrays. A heuristic interactive approach to applying and modifying this algorithm is presented. Examples are produced, including a cellular multiplier.

CHAPTER I

INTRODUCTION

I.1 Scope and Purpose

This thesis sets forth a technique for synthesizing cellular arrays which realize arbitrary Boolean switching functions. The technique is based upon the Polish notation representation of Boolean functions and is heuristic in approach.

A correspondence is defined between Polish representations and certain cellular arrays. Sufficient procedures are defined to guarantee that a Polish string representing a Boolean function can be embedded in a particular style cellular array. Additional procedures for manipulating Polish strings are introduced which, when applied heuristically, may improve the realization. A new family of arrays is defined and a heuristic approach to embedding the Polish representation in them is detailed. The new family of arrays will, in may cases, permit realizations of functions having fewer cells than previous array styles.

The techniques developed in this thesis differ from past results in one major way. A heuristic approach to cellular logic has, to the author's knowledge, never been developed. Previous efforts have been aimed at deterministically synthesizing all functions in a particular array style. Further efforts have been made to determine the optimal

solution by some criterion. The resultant array is quite independent of the logical designer and depends only upon the algorithm developed by the theoretician. In contrast, this study defines procedures for embedding functions in a class of arrays (the Buss arrays) in a heuristic way. No guarantee is made of an optimum solution by any criterion. In fact, the result will be highly dependent upon the skill of the logical designer. He must define his own criteria and use the procedures to find a "good" though not necessarily optimal solution. In this manner it is hoped that the power of algorithms may be meshed with the intuition of the logical designer to realistically and efficiently solve the cellular synthesis problem.

The techniques developed are amenable to implementation on a computer graphics system, such as the one at Montana State University, and comprise a workable engineering approach to cellular synthesis. In Chapter IV are recommendations for further research; it is suggested that implementation on such a system is necessary to realize the full power of this approach.

I.2 Historical Context

Historians will certainly never unanimously agree, in their delineation of great epochs, on the efficacy of the various events which usher in an age. However, for purposes of engineering and electronics technology, the author suggests that four events constitute the boundaries of four decisively different eras. The first event considered is the invention of the Flemming Valve. The amplifying capability of this device made possible--if however bulky and unreliable--the world of modern communication including ultimately the modern digital computer, envisioned since the age of Babbage[1]*.

The second momentous discovery to affect the world of electronics and computing was the well-known and successful production of the first transistor at Bell Laboratories. Gradually and not without agony and frustration, computers of greater power moved from large rooms to small benches.[1] As size diminished so did cost for machines of comparable power while reliability and speed improved.

The third innovation, an outgrowth of the transistor, is the integrated circuit (IC). Recently used in commercial computers, the IC has further reduced size and costs. The fourth innovation is a natural outgrowth of IC technology as more and more circuits are

---

*References to the bibliography are signified by superscripts.

incorporated into one substrate. When several hundred gates are realized on one substrate the general term large scale integrated circuit (LSI) is used. LSI has opened the door for many exciting possibilities in computing (including the respecifiable computer).* Presently manufacturing technology is in its infancy for LSI. Manufacturers are talking presently in terms of 1000 logic gates per chip in 1970 and 10,000 by 1976[10].

Each technological breakthrough has created a plethora of problems challenging to the technology and knowledge of its time. History records how the large capacitance of the triode was conquered by introduction of a second and a third grid, enabling amplification at megahertz. The transistor was an expensive, limited-use device until manufacturing technology advances enabled parameters to be controlled adequately during fabrication. With LSI comes the problem of connection to the outside world. While hundreds of transistors can be fabricated on a small substrate, only enough connections can be made between the substrate and the outside world to directly control a small fraction of the transistors.

As each new epoch has evolved, new mathematics have been invented to describe the actions of the new devices, for example: equivalent circuits, graph techniques, switching theory. Each of these was

---

*The author uses this term to indicate a computer which can change its functional structure while operating.

developed to describe the powers of the devices available, and each has been molded into the cohesive whole which comprises electronics. Such is now the case with LSI. Efficient mathematical representations of LSI structures and procedures are being sought which allow efficient synthesis of arbitrary Boolean functions. These representations and procedures are becoming the tools of the cellular logical designer.

This thesis develops a representation and design procedure for synthesis of a class of cellular arrays.

I.3 Organization of Remaining Chapters

Chapter II sets forth pertinent background information on cellular logic, Polish notation and heuristic systems. A reader familiar with any of these subjects could skip the appropriate sections. Chapter III presents the significant work of the thesis. Chapter IV outlines the author's ideas for future work in this area, and presents the author's conclusions and summary.

CHAPTER II

SURVEY OF LITERATURE

In this chapter is a brief survey of the developments in cellular logic. Only elements germane to this thesis are discussed in detail. For a more comprehensive exposition and excellent bibliography, the reader is referred to Minnick's "A Survey of Microcellular Logic"[19].

Additionally Polish notation is introduced and its relationship to infix ("normal") representation of algebraic quantities is discussed. Finally heuristic systems are introduced.

II.1 Developments in Cellular Logic

II.1.1 Economics of LSI

In the usual production of integrated circuits, a silicon wafer is formed, many identical IC's are fabricated on it, the wafer is diced, and the circuits are individually packaged. Tests are made at various points along the process. These circuits are then assembled on cards and connected together using large amounts of interconnection wiring, including backplane wiring.

In contrast, using LSI modules, the wafer is not diced but the entire interconnection structure is produced on the substrate and the chip is placed in a single package, saving backplane wiring and space. An additional benefit is an increase in reliability expected with the elimination of most of the metal-to-substrate bonds and solder connections in the backplane. Also, the close proximity of the circuits increases the speed of propagation limited systems.

On the debit side with the increased complexity of a module, there are increased chances that one or more of the circuits may be faulty. Methods must be found for location, detection, and avoidance of these faults. Additionally, since mask fabrication presently is an expensive task and large-complexity modules producing a fixed set of functions have low density use, it is desirable that methods be found to allow run-time specification of the functions to be

realized by the module. At least, a method is needed which allows fixing the exact circuit late in assembly[19]. Furthermore, around 50 connections to the outside world can be presently made, (perhaps 100 in the near future);* explicit control of 1000 circuits is, in the forseeable future, impossible. Methods must be found to achieve control implicitly or indirectly.

Fortunately, the above-mentioned problems have been to some extent solved. Fault avoidance techniques for certain types of cellular arrays have been demonstrated by Minnick[16-18, 22], and Spandorfer and Murphy[29]; and location and detection algorithms have been produced by Kautz[7-9], and most recently and completely by Thurber[31, 32].

Brooking, King, and Giusti[2] have built an array using a punched card, light source, and photocell combination as function controllers. Minnick[17] has suggested that these might also be synthesized using flip-flops. Minnick also suggested that a fuse could be used in each of the interconnections, with some method of blowing those in the undesired paths - possibly using coincident currents. This has the disadvantage of being permanent. The third problem mentioned above, that of pin limitations, is dealt with in cellular logic.

---

*Private Communication.

II.1.2 Cellular Arrays

Cellular Logic deals in general with arrays of functional modules such as that of Figure 1. Each module is assumed to be capable of realizing one or more Boolean functions, f, of its inputs, x_i, and may have one or more outputs, y_i. These modules are connected together with some uniform interconnection structure. Data is entered only along the edges of an array. Certainly other approaches to design of LSI circuits can be and are being used, such as a brute-force method utilizing a new mask for each new set of output functions. However, the cellular-logic approach, with uniform interconnection structure and a fixed set of functions for each module allows an orderly development amenable to mathematical description and procedure.

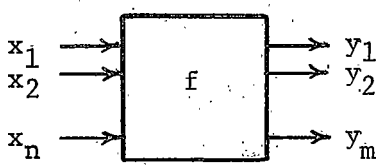


Figure 1. A Functional Module

The cellular approach reduces the pin problem considerably. For example, if each transistor on a chip of 800 transistors were connected directly to the outside world, 2400 pins would be required for logical connections. The same transistors, if used two per cell in an array of 40 cells such as Minnick's cutpoint array, could require only 60 pins for logic[16]. This appears to be within manufacturing capabilities.

II.1.3 Some Types of Cellular Array

Cellular arrays have been developed based on modules which perform a fixed function. An example of these is Spanderfer's NOR-based wiggle buss array shown in Figure 2[29]. Each module in the array realizes the function NOR, and the interconnection structure is specified by selectively closing the breaks in the lines between the cells. Other examples include the eight-neighbor arrays originated by Brooking and King[2,11].

The second basic type of array which has received much attention is the cascade array[20], so called because it is based on the Maitra or general-function cascade, developed independently by Maitra and Minnick in 1962[13]. Since the procedures developed in this thesis are designed for a type of cascade array, the Maitra cascade and its generalizations will be treated below in some detail.

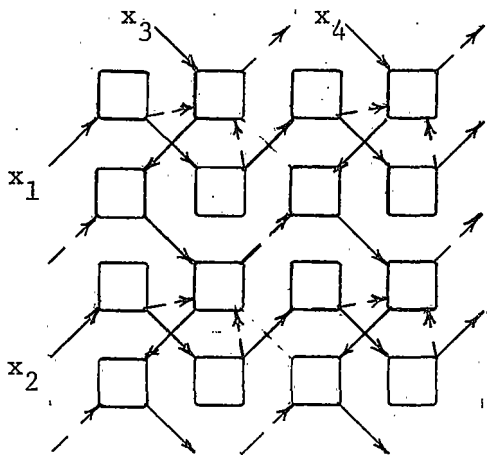


Figure 2. A NOR-Based Array

The basic Maitra cascade is shown in Figure 3. It consists of 17 functional modules or cells, each of which realizes any of the 16 Boolean functions of two variables of its two inputs. Maitra produced tests to determine what functions of n variables could be produced in a single cascade. In 1963, Sklansky[28] and in 1964, Levi Winder and Mott[12] produced algorithms for testing functions for cascade realizability. In 1964, Minnick[16,17,21] showed that only six of the 16 functions are necessary and further, that certain additional functions are realizable by redundantly entering some input variables.

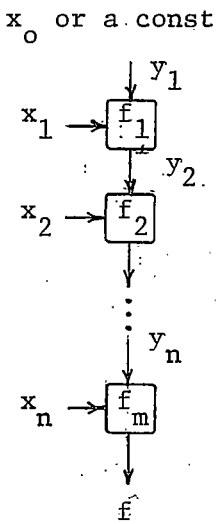


Figure 3. A Maitra Cascade

Minnick[16,21,22] produced perhaps the earliest array based on a Maitra cascade shown in Figure 4. His array uses cutpoint cells* as modules, and consists of several cascades of cutpoint cells with inputs bussed across the array. The bottom row is a cascade turned sideways and is called a collector row, since its function is to collect the subfunctions produced in each column. In Figure 4, a

---

*The cutpoint cell was so-named by Minnick because of the method of controlling the interconnection structure. Each cell realized the set of functions {1, y', x' + y', x'y', x + y, xy', x ⊕ y, SR Flip-Flop}.

number in a cell would indicate which particular cutpoint function is produced by the cell. The functions are listed in Table 1.

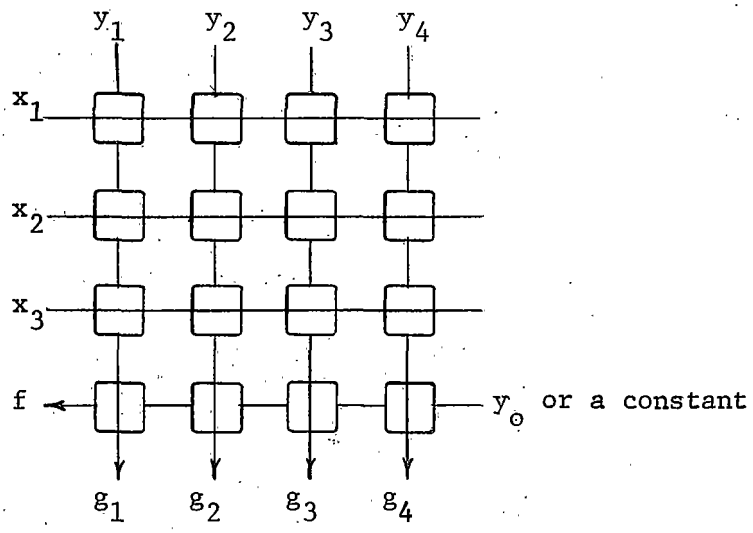


Figure 4. A Cutpoint Array

Minnick[16] also developed synthesis techniques for the cutpoint array, based on Reed and Shannon expansions and the disjunctive normal form. He produced many examples of combinational and sequential networks realized with cutpoint arrays. In 1965, Minnick developed the cobweb array shown in Figure 5. This array is produced by complicating the interconnection structure of the cutpoint array

and using additional cutpoints to select the interconnection paths used. Figures 6 and 7 show possible realizations for the cutpoint and cobweb cells. The cutpoints are labeled with letters.

TABLE 1. CUTPOINT CELL FUNCTIONS

Index	Function
0	1
1	y'
2	x' + y'
3	x'y'
4	x + y
5	xy'
6	x ⊕ y
7	0
13	x=S, y=R of an SR Flip-Flop

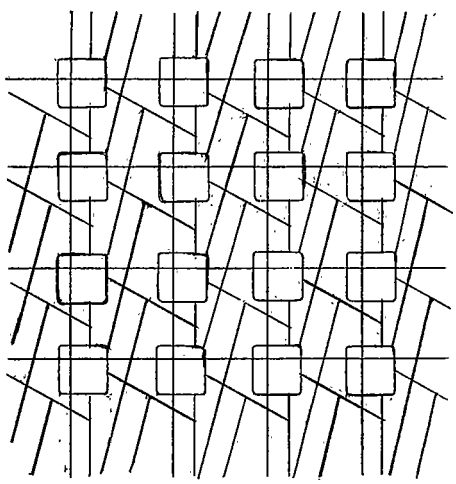


Figure 5. A Cobweb Array

Minnick later introduced a slight simplification of the cobweb array called the cobweb type A^{20}, shown in Figure 8. This thesis will present synthesis procedures for a modification of this array.

II.1.4 Cost Measures of Various Cellular Array Types

It is constructive to measure the costs of producing arrays. For this, the work of Minnick[20] is again cited. Table 2 lists costs for the arrays discussed as well as others. For a given array type, let M be the product of: the maximum number of cells needed to produce an arbitrary function of n variables, and the sum of the

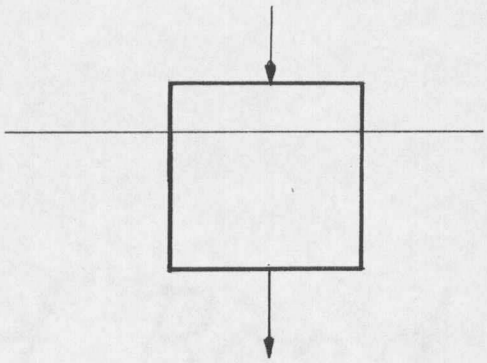
number of inverters and specification bits per cell. Let $O(M)^*$ be the leading term of this product for large n then $O(M) = Kn2^n$ for all arrays in Table 2. So $K = O(M)/n2^n$ can be called the overall cost measure. It is interesting to note that the type A cobweb array has the second lowest cost measure. Advantages accrue to the cobweb array when compared to the cutpoint in that examples have been produced[18] showing cobweb realization needing one-tenth of the number of edge connections and roughly the same number of cells. This would cause an order of magnitude reduction in backplane wiring.

II.1.5 Known Cellular Synthesis Techniques

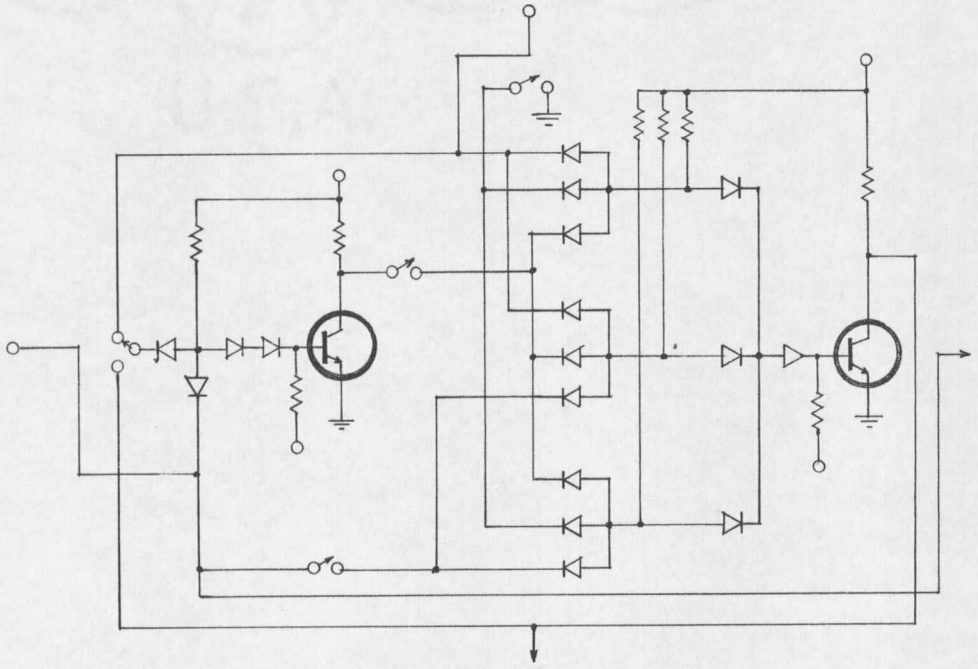
Several techniques are known for synthesizing particular cascade-based arrays such as the cutpoint and cobweb arrays. These deterministic methods are based upon the Maitra cascade or some related cascade. A natural question which arises in this process is whether or not the function or some subfunction is realizable in a single cascade. A positive answer to this question from a deterministic test should lead to a realization in one cascade of the function or subfunction. The known tests for cascade realizability are mentioned below. The weight synthesis technique is treated in detail, as it is, in the author's judgement, the best method known to date. Following this, techniques for synthesizing arrays are set forth. Proofs are omitted.

---

*Called "order of M" this function approximates M for large n.



(a)



(b)

Figure 6. A Cutpoint Cell

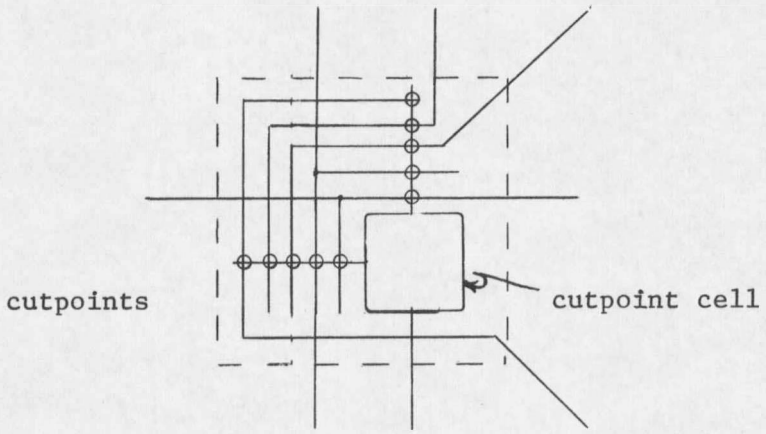


Figure 7. A Cobweb Cell

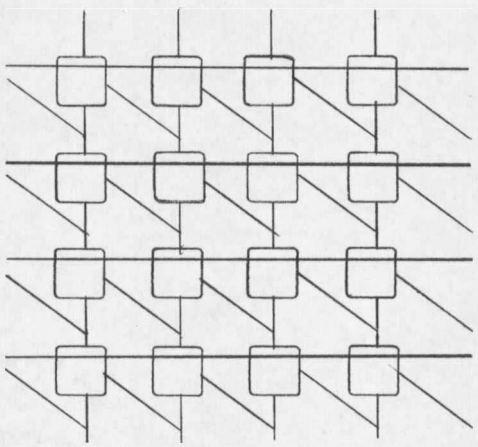


Figure 8. Type A Cobweb Array

TABLE 2. COST PARAMETERS OF SEVERAL CELLULAR ARRAYS*

Style	Description	Inverters per cell	Bits per cell	Inputs per cell	Outputs per cell	Cost measure K	Uniform inter-connection
0	Basic NOR array	1	0.75	1.5	1	3.50	No
1	NOR-based with three NOR's/ cell and one bus	3	3	3	3	3.00	Yes
2	NOR-based with two busses	4	7	5	2	2.75	Yes
3	Styles 0 or 1 embedded in 8-neighbor array	1	8	8	1	18.00	Yes
4	Style 2 embedded in 8-neighbor array with busses	1	8	8	1	9.00	Yes
5	Style 1 with electronic specification bits	16	0	8	8	8.00	Yes
6	Cutpoint with flip-flop. Collector row is a separate array	2	4	2	1	1.50	No
7	Type A cobweb with flip-flop	2	7	3	1	2.25	Yes

TABLE 2. CONTINUED

Style	Description	Inverters per cell	Bits per cell	Inputs per cell	Outputs per cell	Cost measure K	Uniform inter-connection
8	NOR realization of restricted function cells with flip-flop. Four NOR's/function	1	1.5	2.5	1	2.50	No
9	NOR realization of restricted function cells with flip-flop. Two or four NOR's/function	1	6	6	1	4.55	Yes

*From reference 20 with permission.

In 1960, Maitra[13] developed a method for embedding functions in a "Maitra Cascade". He used a two-dimensional map which had to be constructed after one-dimensional maps were constructed to show the function realizable. This method is cumbersome, needing up to N! maps to prove whether a function is realizable or not.*

In 1963, Sklansky[28] published a more efficient method of testing for cascade realizability and synthesis of Maitra Cascades. This method is based upon a matrix or table of the minterms for the function. Sklansky also restricts the needed functions within the cells to six, showing that the set of functions cascade realizable by using the original complete set (16) of two-variable functions and the set realizable using his six functions are the same. Further the length of the realization is the same. This method requires only one search per synthesis step, which is conceptually more complex than Maitra's, but computationally simpler. The primary disadvantage of this method is that for n variables a function may have 2^n minterms, a number which grows quite rapidly.

---

*One map for each ordering of the variables could be needed.

Levy, Winder and Mott published in 1964[12] an algebraic method of testing for Maitra cascade realizability. The following paragraphs including the example are adapted from their paper.

This test is not based upon minterms but upon the function equation as it is written. The function is assumed not to have superfluous variables.* A partial ordering is set up on the variables. Each variable in turn is set first to 1 and then 0. For each setting, the equation is examined to see if it is independent of some subset of the other variables. Then a variable, z, is said to "precede" y if z vanishes from the equation for a value of y, (written $z \leq y$). All other variables, say e and d, are said to "succeed" y, (written $y < e,d$). For a set of tests on a variable, there can be one of three results:

1. In both cases the resultant equation is dependent on all variables.
2. In one case the resultant equation is independent of some variables.
3. In both cases the resultant equation is independent of some variables.

If the third result occurs, the function is not cascade realizable. If the first result occurs, the variable must feed into a $\oplus$ gate and

---

*A superfluous variable is one which may be removed from the function without changing the function. For example, x is superfluous in $f = xy + x'y + z$.

no information is gained for the partial ordering. If the second result occurs, a precede-succeed or PS relation is written as explained above, and the variable will enter a + or × gate.

When all variables are tested, a partial ordering is established by combining the relations found. Any conflicts mean that the function is not cascade realizable. The partial ordering determines the order in which variables enter cells. Variables within a level of the partial ordering may be permuted.

The function of the cells must be determined. The function will be from the set {×, +, ⊕}. This set can easily be shown to produce the same set of realizable functions as Sklansky's set of {xy, x'y, x+y, x'+y, x⊕y, x'⊕y} if one allows x or x' as the input variable. Indeed once such a cascade is built, a transformation may be defined using the larger operator set, to change all x' entries to x. One begins with variables succeeding all others and works to the other end of the partial ordering. If for the variable x under consideration case one resulted from the test, the variable enters a ⊕ gate. Let f_n be the function of n variables, and f_{n-i} be the residual function of n-i variables after a variable is eliminated. Then if x is used as the input $f_{n-1} = f_n(x = 0)$; and if x' is used, $f_{n-1} = f_n(x = 0)^*$. Additionally $f_{n-1}(x = 0) = f'_{n-1}(x = 1)$ must be true or the function

---

*That is to say, the residual function coming in the other input must be the original function with x set to 0 or 1 respectively.

is not realizable. If the second test case occurs, and one setting of the variable leaves a residual function of 1, then the gate must be +. If one setting of the variable leaves a residual function of 0, the gate must be an ×. For a + gate if $x = 0$ yields $f_{n-1} = 1$ then x' is the input. Otherwise the input is x. For the × gate if $x = 1$ yields $f_{n-1} = 0$, then x' is the input. Otherwise x is the input. An example should help solidify the procedure. Table 3 illustrates the testing of $f = c'g + cg'(b + a'de' + (a + d')\, e) + b'c'(a'de + (a + d')\, e')$.

From Table 3 is determined the partial ordering: $a,d<e<b<g<c$. This fixes the inputs to the cells of the array as shown in Figure 9. Note that variables a and d could be interchanged. To complete the process, one decomposes the function beginning with variable c. First algebraic manipulation will show that $f_6(c=0) = f_6'(c=1)$. Choosing $f_5 = f_6(c=1)$, c' is the input to the ⊕ cell, leaving $f_5 = g'(b+a'de'+(a+d')e)$. Clearly $f_5(g=1) = 0$ and thus the function of cell 2 is ×, and g' is the input. Then the residual function is $f_4 = f_5(g=0) = b+a'de'+(a+d')c$. Again $f_4(b=1) = 1$, yielding cell 3 as +, b as the input, and $f_3 = f_4(g=1) = a'de'+(a+d')e$. The next variable is e, so check that $f_3(e=0) = f'_3(e=1)$. This is so. Choosing $f_2 = f_3(e=1) = a+d'$, the fourth cell is a ⊕ and e' is the input. The remainder follows easily.

TABLE 3. TESTS FOR SYNTHESIS EXAMPLE

Test	Residual Equation	Relation
a = 0	c'g + cg'(b + de' + d'e) + b'c'(de + d'e')	
a = 1	c'g + cg'(b + e) + b'c'e'	$d \leq a < b,c,e,g$
b = 0	c'g + cg'(a'de' + (a + d')e + c'(a'de + (a + d')e'	
b = 1	c'g + cg'	$a,d,e \leq b < c,g$
c = 0	g + b'(a'de + (a + d')e'	
c = 1	g'(b + a'de' + (a + d')e)	None
d = 0	c'g + cg'(b + e) + b'c'e'	
d = 1	c'g + cg'(b + a'e' + ae) + b'c'(a'e + ae')	$a \leq d < b,c,e,g$
e = 0	c'g + cg'(b + a'd) + b'c'(a + d')	
e = 1	c'g + cg'(b + a + d') + b'c'a'd	None
g = 0	c(b + a'de' + (a + d')e + b'c'(a'de + (a + d')e')	
g = 1	c + (b'c'(a'de + (a + d')e)	$a,b,c,d,e \leq g < c$

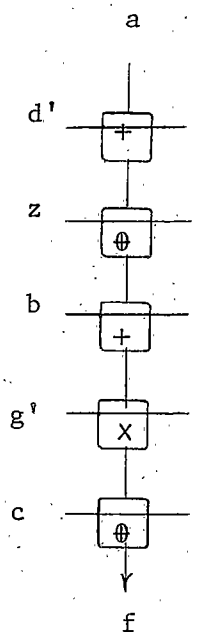


Figure 9. Cascade for Synthesis Example

The advantage of this method lies in the lack of exhaustive testing or listing of minterms. However, one must manipulate algebraic quantities, which takes some effort on a computer.

Stone and Korenjak[30] have developed a testing and synthesis method which makes use of the weight* of the function. This has been further refined by Minnick** and given the label of Weight Synthesis. This

---

*The weight of the function is the fraction of the minterms that are true values for the function.

**Unpublished.

method includes synthesis of redundant cascades, that is cascades which have a variable entering more than one cell. This method is stated in detail below. For a derivation and proof of this approach, the reader is referred to[30].

It is convenient to state the algorithm in a flow diagram as in Figure 10. In Figure 10, G_0 and G_1 are the Shannon subfunctions of the function. (That is, $f = xG_0 + x'G_1$) The current function is denoted f, while residual functions are denoted f_r. The algorithm ends when two variables are left. They are both inputs to the top cell.

Stone and Korenjak[30] proved some results which are important to the above algorithm. These are:

1 Lemma. If more than one variable can be decomposed at one stage in the realization, the successful realization and length of realization are independent of the order in which the variables are decomposed (redundant or irredundant).

2 Lemma. If a variable, x, appears as an input to a × or + cell, it never need appear further up the cascade.

3 Lemma. A function f is cascade realizable if and only if f' is. Further, the minimum length realization has the same number of cells for each one.

4 Canonical Theorem. Any cascade realizable function of n variables can be realized by a cascade consisting of (1) a cascade realizing a function of n-1 variables, and (2) either a $\oplus$ cell driven by the nth variable, or a $\times$ or + cell driven by the nth variable followed by a possibly empty set of $\oplus$ cells, each having a different input variable from the remaining set.

Lemma 1 states that when one has found a decomposition of the type detailed in the theorem, it is as good as any other one. The theorem recursively states what the cascade synthesized by the algorithm will look like. It also states where the extra $\oplus$ cells will occur in a redundant cascade.

Lemma 2 helps form the background for the theorem. It also indicates that all entries of a variable except the topmost one must be into $\oplus$ gates. Lemma 3 answers the obvious question about the possibility of using the complement of a function for a better realization.

Weiss[33] has developed a synthesis method which begins with the prime implicants representing a function. He builds from these a series of two tables and performs manipulation on them. He sets forth necessary and sufficient conditions on the tables for cascade realizability. From the table a "Standard Cascade Form" or "SCF", is built and from this the cascade is determined. In a later article[34] Weiss expands his work io include redundant cascades. This procedure appears to be the most efficient method to date. However, this method lends nothing to the understanding of cascades and its procedures are lengthly.

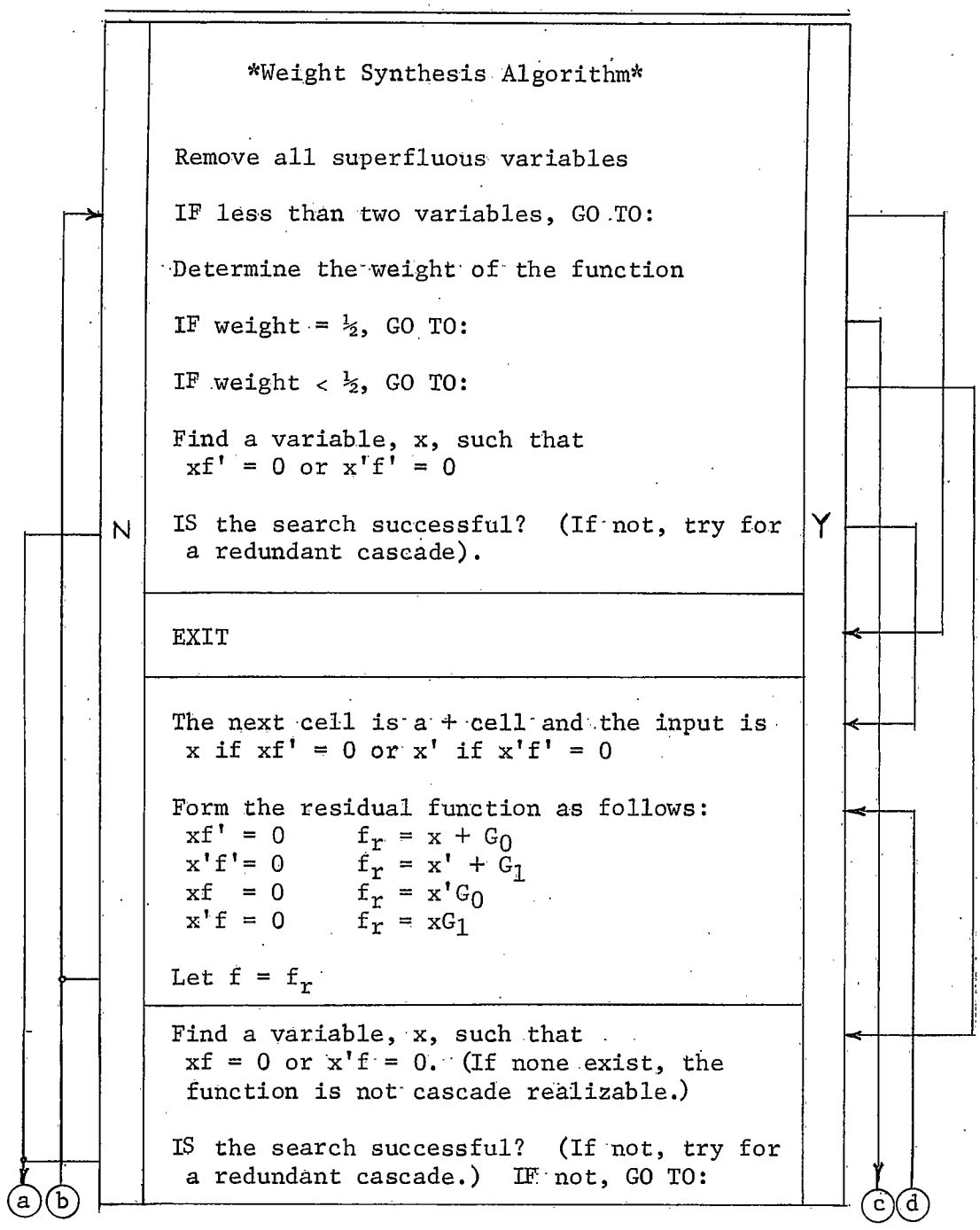


Figure 10. Algorithm for Synthesis of Redundant Maitra Cascades

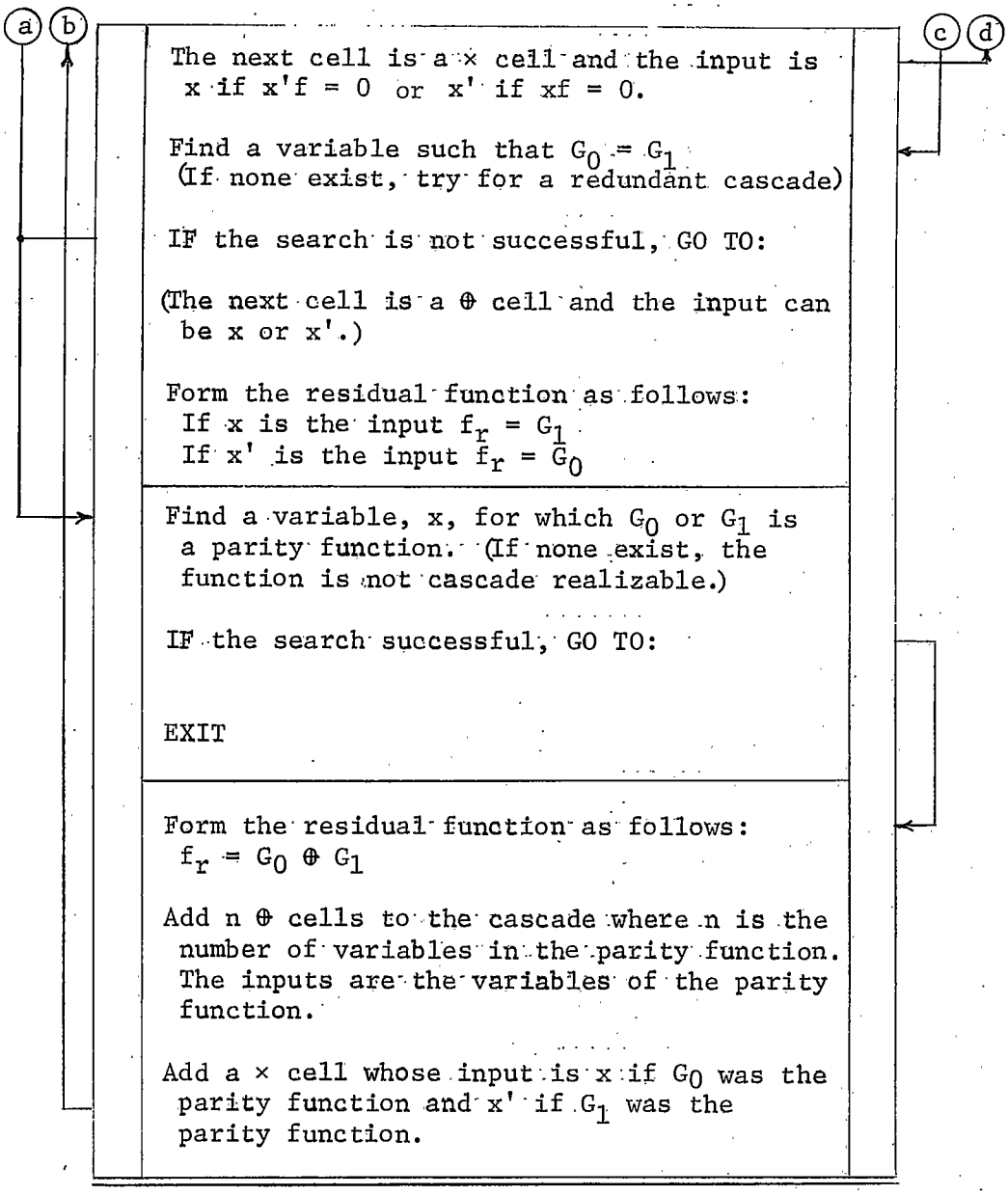


Figure 10. Continued

This completes the exposition of known tests for cascade realizability. Next one must consider the functions which are not realizable in a single cascade. Maitra[13] showed that a vanishingly small fraction of the total number of functions of n variables can be realized as n grows large. Nevertheless, an array of such cascades can realize all functions of n variables.

Several such arrays have been developed. While synthesis techniques have been developed to realize these arrays, no general technique is known which is suitable for all arrays. Presented below are techniques based on the cutpoint and cobweb arrays which were introduced in the previous section. These arrays have been shown to be of interest[17,19], and the procedures of this thesis are applicable to similar arrays.

The cutpoint array is due to Minnick. See Figure 4. The sequel is adapted from Minnick's paper[16].

One way of approaching synthesis of cutpoint or cobweb arrays is through use of minterms. The disjunctive canonical form of any Boolean function is a sum of minterms of that function. Any minterm can be realized in a Maitra cascade. The collector row can then be used to sum the minterms, yielding the desired function. One result of this method of synthesis is an upper bound on the size of the array needed for an arbitrary function of n variables. A function of n variables has 2^n or less minterms. Each minterm has n variables. Thus adding

one for the collector row, a cutpoint array for an arbitrary Boolean function of n variables need not have more than n+1 rows and 2^n columns. Thus the number of cells in the array as n grows large is $0(n2^n)$.

Any prime implicant can be realized in a Maitra cascade. There are usually fewer prime implicants for a function than minterms, so the number of columns can usually be reduced by realizing an irredundant normal form.

Figure 11 is an example of three realizations of the function,

$$f = \Sigma(1,3,5,6).$$

Figure 11(a) is the disjunctive canonical form realization. Figure 11(b) is a realization of an irredundant normal form. Figure 11(c) is derived by realizing that the function can be written as

$$f = x_1x_2 \oplus x_3.$$

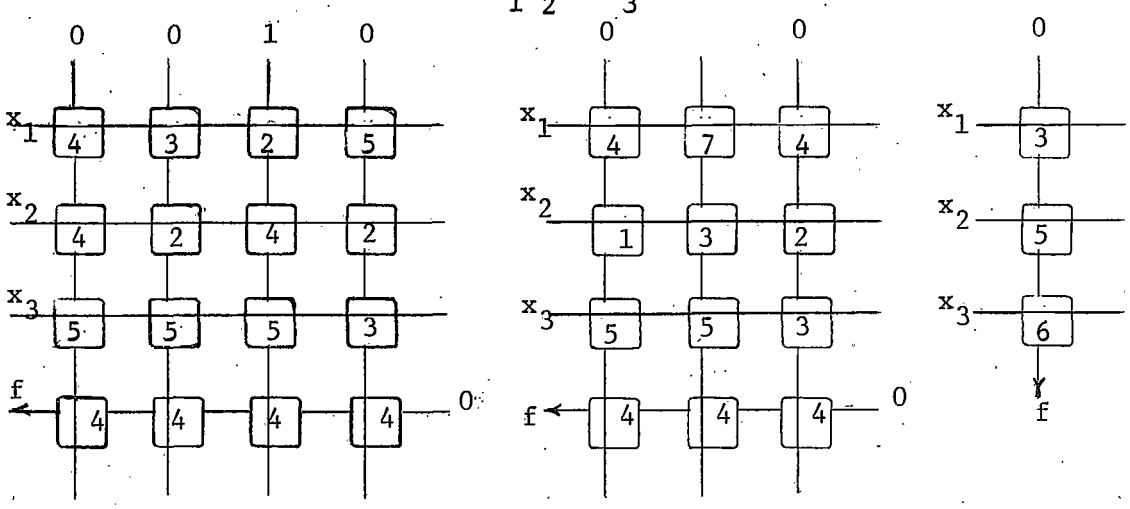


Figure 11. Cutpoint Synthesis Example

It is possible to realize more than one function in a cutpoint array by including an additional collector row in the array for each function. By judicious choosing of the locations in which the functions are realized, often several forms may be used in more than one function. This is possible since the output of a cascade can be an input to a row of cells. Figure 12 is such an example for the three equations:

$$f_1 = x_1x_2$$

$$f_2 = x_1x_2 + x_1'x_2'x_3'$$

$$f_3 = x_1'x_2'x_3' + x_3(x_1 \oplus x_2).$$

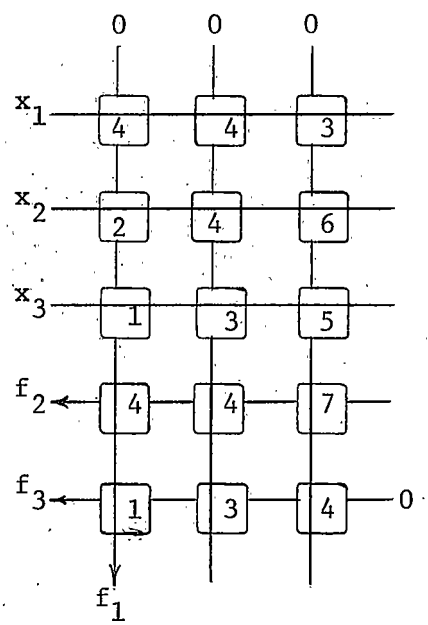


Figure 12. Cutpoint Realization of Three Functions in One Array

Another method of synthesis is based upon the well-known Shannon decomposition. Since one cell can produce any function of two variables, one can guarantee a realization of any three-variable function in two columns if one can write the function as two functions of two variables, independent of the third one, and compose them somehow with the third variable into the function. The Shannon decomposition is a method of doing this. The Shannon form for a three-variable function can be written as

$$f = x_i G_1 + x_i' G_0,$$

where x_i is one of the three variables, arbitrarily chosen. The functions G_1 and G_0 are independent of x_i. They are actually the functions:

$$f(x_i = 1, x_j, x_k) \text{ and}$$

$$f(x_i = 0, x_j, x_k),$$

respectively, where x_j and x_k are the other two variables. Figure 13(a) illustrates the general realization of any three-variable function through the Shannon decomposition.

Shannon's decomposition can be written for any number of variables. This allows the extension of this method for arbitrary functions of n variables. For example, any four-variable function can be written as

$$f = x_i x_j' G' + x_i x_j H + x_i' x_j' J' + x_i' x_j K.$$

Figure 13(b) illustrates this four-variable decomposition.

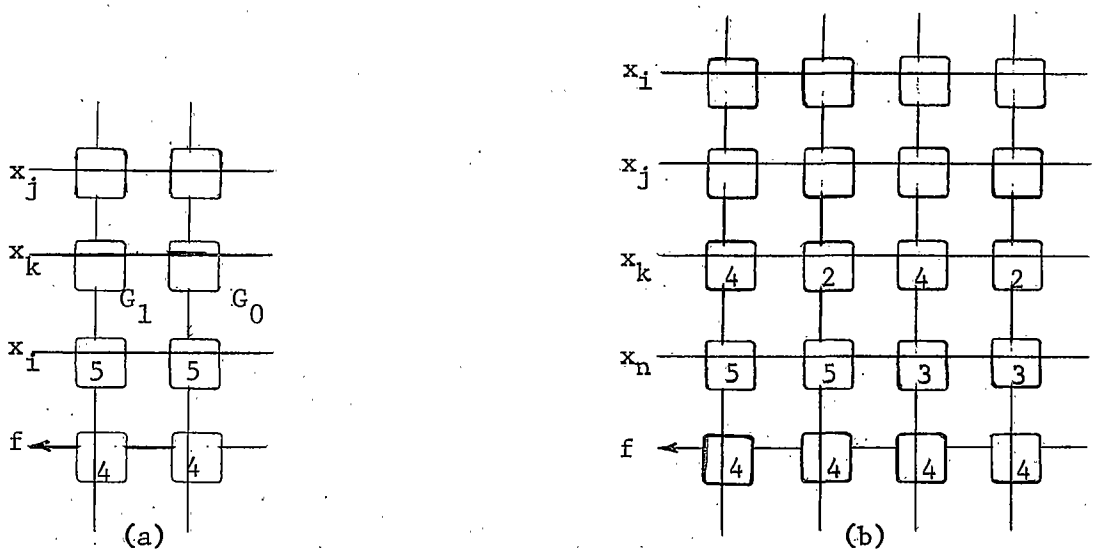


Figure 13. Cutpoint Realization of Arbitrary Three-Variable and Four-Variable Functions

The cutpoint array suffers from a wastage of cells for many problems. This occurs when a subfunction produced in one portion of the array could be used elsewhere, or when the function has a number of conjunctive terms with a high number of literals, with many terms requiring the same literals. The problem reduces to that of redirecting the flow of information within the array. It is desirable that information flow be changeable from vertical to horizontal or diagonal.

To overcome this problem, Minnick developed the cobweb array. The following is taken from his work[18,22]. Figures 5 and 7 illustrate the cobweb array and its cell which is based upon the cutpoint cell. Compared to the cutpoint array, the cobweb array has an additional vertical buss, one knight's move buss and a left connection to neighboring cells. These additions allow re-directing of information flow within the array and presenting information at remote locations without using intervening cells. An obvious advantage is that any row or part of a row may perform the collector function.

The synthesis techniques described above can be applied to cobweb arrays. The collection facility makes it possible to perform more efficient Shannon or Reed decompositions for four or more variables. In the cobweb array, the function may be decomposed on one variable and the residual function decomposed again, if necessary, forming a tree structure leaving vacant cells clustered together for later use. This is compared to the cutpoint array where the entire decomposition must be done in the collector row. Cobweb arrays normally require significantly fewer cells than cutpoint arrays.

Recently Weiss[33,34] has extended the two-level logic theory of Quine[25,26] to arrays. He defines and creates prime cascades as analogous to prime implicants and then picks a set of these, analogous to the irredundant normal form, which forms a minimal cover under the

criterion of a minimum number of cascades. The procedure allows redundant cascades and can be extended to include the requirement of ordering the variables in the same way for every cascade (so they will fit in the cutpoint array).

Weiss's method apparently is a reasonable approach which will yield an optimum result for arrays with a single collector row, provided each cell has an input from the outside world. If each row of cells has only one variable connected to the outside world (such as in the cutpoint array) the procedures may be extended; but in this author's judgement, the computation involved will become excessive.

II.2 Polish Notation

In this section, a brief background on Polish notation is set forth. In the next chapter, it will be shown to be useful in cellular logic because of its correspondence to certain arrays.

Polish notation was introduced in 1929 by the Polish logician, Jan Lukasiewitz[6]. The form used in this thesis is variously called Lukasiewitz notation, L-notation, P-notation, reverse Polish notation and postfix notation. Introduced as a tool for propositional calculus, P-notation is today also known in this country by compiler writers[27]. Its similarity to a pushdown stack makes it natural for this use; and at least one machine, the Burroughs B-5000 was designed around it.

In Polish notation, a string representing a function is written with operators after, not between, their arguments as in standard or infix notation.* The expressions thus formed are unambiguous and no parentheses are needed. In general N-adic operators can be allowed. However, in this thesis only dyadic operators and the monadic (') will be used. Table 4 gives several examples of P-strings and their infix equivalents.

---

*Alternately operators could be always written before arguments. The resultant representation is another Polish notation form, called forward Polish notation and prefix notation.

There are, of course, several ways to represent an algebraic quantity in P-notation as there are in infix notation. This is pointed out by the last three examples in Table 4.

TABLE 4. POLISH STRINGS AND EQUIVALENTS

	Infix String	Polish Notation Equivalent String
1	a'	a'
2	ab or a × b	ab×
3	a + bc'	abc'×+
4	(a + b)c	ab+c×
5	(a + b) + c	ab+c+
6	a + (b + c)	abc++
7	a + b + c	ab+c+

II.2.1 Evaluation of Polish Strings

Polish strings may be evaluated with an algorithmic process. Assume that each variable has a value assigned to it. Begin at the left end of the string and scan right to the first operator. If the operator is dyadic, (needing two arguments), use the two variables

immediately to the left as arguments. If the operator is monadic, (needing one argument), use the variable immediately to the left as the argument. Evaluate this substring and replace it with a new operand having its value. The substring will have the form abϕ or aϕ where ϕ is an operator. The value of abϕ equal to the value of the infix expression aϕb. The value of aϕ (here ϕ must be a (') in this thesis) equals the value of the infix expression aϕ. Repeat this process again until all such substrings are evaluated.

It should now be seen that the Polish strings in Table 4 are equivalent to their infix counterparts as to order of evaluation. For example, in row five, ab+ is evaluated first; in row six, bc+ is evaluated first and in row seven, ab+ is evaluated first. This corresponds to the normal evaluation of infix notation ordered by parentheses and by left to right. It should be noted that Polish notation is a positional notation. That is, order of evaluation depends only upon position within the substring. A direct consequence is that no ordering symbols such as parentheses are used. Table 5 illustrates the procedure for a more complex string. In Table 5, substrings are replaced, after evaluation, by Greek letters. The substring affected at each step is underlined.

TABLE 5. EVALUATION OF A POLISH STRING

STRING	COMMENT
ab+cd'×e⊕×	Begin at left. Find first operator.
ab+cd'×e⊕×	Find operands.
γcd'×e⊕×	Replace by value: γ = ab+ (= a+b)
γcd'×e⊕×	Find next operator.
γcd'×e⊕×	Find operand.
γcδ×e⊕×	Replace by value δ = d'
γcδ×e⊕×	Find next operator.
γcδ×e⊕×	Find operands.
γαe⊕×	Replace with value cδ×=α (= c×δ)
γαe⊕×	Find operator.
γαe⊕×	Find operands.
γβ×	Replace with value β = αe⊕ (= α⊕e)
γβ×	Evaluate final string.

II.2.2 Conversion of Infix Strings to Polish Strings

Conversion algorithms between infix and Polish notation are well known. The one to Polish presented here is adapted from reference 27.

Define three last-in-first-out stacks called 0, I and T and a controller. Figure 14 illustrates the necessary system.

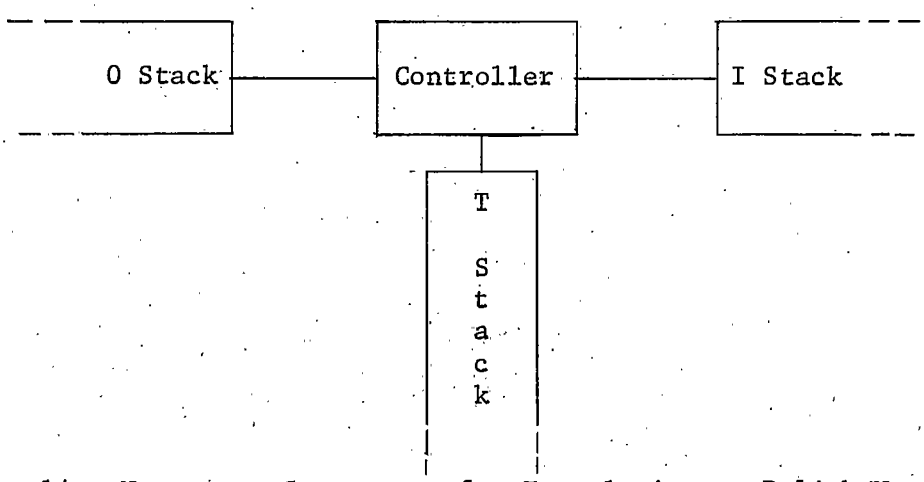


Figure 14. Necessary Components for Translation to Polish Notation

Assume that the infix string exists in the I stack, left end on top. The controller pops one literal at a time off the I stack for processing. If the literal is a variable, it is pushed into the 0 stack. If it is a dyadic operator additional information is needed. Table 6 lists the system of hierarchy used in switching algebra. This hierarchy is the order in which the operations are done in normal notation. For example, a $\times$ is processed before a + and a + before a $\oplus$.

TABLE 6. EVALUATION HIERARCHY FOR SWITCHING ALGEBRA

Symbol	Compare Priority	Stack Priority
(	none	0
)	1	none
$\oplus$	2	2
+	3	3
×	4	4
'	5	5

This information is used when an operator is popped from I. The controller places the operator in T after popping all operators in order until one of greater priority than the current operator is found. The popped operators are then pushed in order into 0 and the current operator is pushed into T. Left parentheses are always placed in the T stack without comparison. Right parentheses are compared but not placed in T. After processing a right parenthesis, its associated left parenthesis which is on top of T is discarded. Table 7 illustrates the procedure.

TABLE 7. EXAMPLE OF TRANSLATION TO POLISH NOTATION

Step	I Stack	T Stack	O Stack	Comment
	top bottom	bottom top	bottom top	
1	a+(b⊕c)×d	Ø*	Ø	Begin with left end.
2	+(b⊕c)×d	Ø	a	Put a in O.
3	(b⊕c)×d	+	a	Put + in T.
4	b⊕c)×d	+(	a	(is not compared but placed in T.
5	⊕c)×d	+(	ab	
6	c)×d	+(⊕	ab	
7	)×d	+(⊕	abc	
8	×d	+	abc⊕	) is compared and discarded. The (is then discarded.
9	d	×	abc⊕+	
10	Ø	×	abc⊕+d	
11	Ø	Ø	abc⊕+d×	Empty the T stack into the output.

*The null or empty string is represented by Ø.

In this section P-notation has been briefly introduced and compared to infix notation. A method of translation from infix notation to P-notation has been developed and an algorithm for evaluation of P-strings has been stated.

In Chapter III, P-strings will be shown to be a useful method of representing Boolean functions when one desires to embed the functions within a cellular array.

II.3 Heuristic System

A heuristic approach to problem solution may be examined against an algorithmic approach. If one solves a problem algorithmically, he is applying a recipe to a problem which, if faithfully and correctly executed, will yield a predictable result, every time. This result is defined as the desired solution. A requirement for this approach is that the problem and its solution be exactly defined.

In contrast Newell and Ernst[23] have defined a heuristic approach as a present state description, a set of operators, and a target state. The operators are transformations which may be applied to the description of the present state to form a new present state, hopefully closer to the target state. All operators may not be applicable to the present state, but will be applicable to some present state. The target state may be a multiplicity of states. A portion of a tree is then produced by applying operators in all possible orders until the target state is reached. In case that the target state is not completely defined (i.e. is a multiplicity of states) an entire tree may be created.

Additions to the heuristic approach allow for the exercise of intelligence. For example, a means-ends analysis may be included. In this case, a difference function between the present state and the target state is defined and only operators which reduce this difference are applied.

Translated into the terms of this thesis, there are two heuristic problems. The first is dealt with in Section III.3. The present state is a Polish string. The target state (an unknown) is a functionally equivalent Polish string which requires fewer array cells to realize. Operators are Boolean transformations which are applied to the Polish string. A criterion is defined to indicate what the most desirable P-string looks like. The designer, within some basic rules, is expected to use his intuition to select the operator to be applied. This is equivalent to defining a difference function and applying only those operators which lessen the difference.

The second heuristic problem is dealt with in Section III.4.7. In this case, the present state is a partial synthesis of an array. The operators are procedures which make the next step of the synthesis, and the target state is a set of completely synthesized arrays with a small number of cells (satisfying some additional constraints). The difference function is determined as the closeness of the final realization to one with a minimal number of cells (which is usually unknown). The operator is expected to apply operators (procedures) which minimize this difference.

Perhaps the greatest example of a heuristic system is GPS (General Problem Solver), created by Newell, Simon, Shaw, and Ernst[23]. This is a computer program which applies heuristic techniques to a

general class of problems, which includes chess playing and theorem proving. The interested reader is referred to reference 23 for further details and a bibliography on heuristic systems.

CHAPTER III

SYNTHESIS OF CELLULAR ARRAYS

The new work of this thesis is presented in this chapter. In the first section, notation and terms are defined. In the second, correspondences between Polish strings and certain arrays are developed, ending with an array in which the function of any Polish string can be embedded. In the third section, a sufficient set of procedures is presented for the implementation of the embedding process. Additional procedures are presented which when applied through the skill of the designer, a computer program, or both, may result in a better realization in the sense of fewer cells being required. In the last section, a new family of arrays, called the buss arrays, is introduced and procedures are developed to embed arbitrary functions in these arrays. Examples are produced.

III.1 Definitions and Notation

This section could profitably be skipped and referred to only if confusion arises. Definition of commonly known terms in switching theory have been omitted.

III.1.1 Definitions

1 Buss Array A buss array is a member of the family of arrays having a series of data carrying "buss" lines. These lines allow data to be created at one point in the array and used at another without using cells between the two points.

2 Buss Segment In buss arrays, a portion of an Hbuss (Vbuss) lying between two Vbusses (Hbusses) or between the edge of the array and a Vbuss (Hbuss).

3 Collector Array Any array which consists of single rail cascades with the outputs forming the x inputs to one or more similar cascades turned sideways. The sideways cascades collect the functions produced by the upright cascades and are called collector cascades or rows.

4 Collector Cascade See collector array.

5 Collector Row See collector array.

6 Column Used as in "row m and column n" of an array, column differs from cascade in that cascade sometimes may mean a column specified to a particular function. Also, column refers to the connection from the outside world to the y input of the top cell of an array, the outputs to the outside world from the bottom row of cells of an array, and the vertical output from the bottom Hbuss of an array.

7 Complex P-string Any P-string which is not elementary or simple.

8 Cutpoint A place where a buss may be broken. Also, a specification bit for an array cell.

9. Cutpoint Cell A single rail cell which can produce any of the functions of two variables listed in Table 1.

10 Disjunctive Normal Form In regular notation it is a sum of Boolean product terms with each variable appearing complemented or uncomplemented not more than once in each product term. In P-notation it appears as a sequence of elementary P-string having +'s for the level-one uniting operators and ×'s for all level-two operators. All +'s appear at the tail of the P-string.

11 Dyadic Operator An operator which requires two arguments.

12 Elementary P-string A P-string consisting of a sequence of variables, possibly including monadic operators, followed by a sequence, possibly empty, of dyadic operators, possibly including some monadic operators.

13. Functional Cell In an array, a group of cells which have been specified as a unit to perform some task or tasks.

14 General Function Cell A single rail cell which can be set to any of the 16 functions of two variables and the flip-flop function.

15 Head of a String The left end of a string.

16 Intermediate Function A function which is created in the process of creating another function.

17 Internal Function The output of a flip-flop.

18. Joinpoint A crossing of two busses which may be connected if desired.

19 Level-n Substring An argument of a level-n uniting operator. See: uniting operator.

20 Literal Any character in a string.

21 Monadic Operator Any operator requiring only one argument.

22 Operator String A string containing no variables.

23 Pluse The exclusive-or operator.

24 Polish Notation A system of notation in which the operators are written after (or before) their operands instead of between them.

25 P-Notation See Polish notation.

26 P-String A string written in P-notation.

27 P-Substring A string written in P-notation which is part of another string (including itself).

28 A Relationship This is defined in terms of the order of appearance of variables within a P-string. Define {< > ,} (a set) as follows. If a variable, x, appears or must appear to the left of a variable, y, in a P-string, write: x<y. If x must or does appear after y, write: x>y. (This will happen when the variables cannot be commuted). If x may appear after y or before y write: x,y. Thus x,y<z reads: x and y appear (must appear) to the left of z, and x and y may appear to the right or left of each other. This relationship is a partial ordering.

29 Simple P-String A P-string consisting of one variable or one variable followed by a prime.

30 Single Rail Cascade A cascade of cells in which the nearest neighbor cells are connected by one logical data path. (Each cell produces only one function at a time[21,22].)

31 String A sequence of variables and operators which constitutes a Boolean function.

32 Tail of a String The right end of a string.

33 Uniting Operator A level-n uniting operator has a level-n substring as one argument and a subfunction as the other. Its task is to unite the level-n term with the other level-n substrings which make up the subfunction which is its other argument. All level-one uniting operators can be found empirically as follows. Find the first literal from the tail which is a variable. Continuing left, find the next dyadic operator. The variables to the right of this operator are the variables of the rightmost level-one substring. These variables plus enough of the operators

from the immediate right of the first-found variable to make a well-formed string, constitute the rightmost level-one substring. All operators to the right of this substring are the level-one uniting operators. To find level-two uniting operators, apply this construction to each level-one substring. For example, if the literals in the function, abc++wx+y×ade××++, are numbered left to right as 1,2,...,16,17, then elements 16 and 17 are the level-one uniting operators. The level-one substrings are elements 1-5, 6-10 and 11-15. Level-two uniting operators are elements 4-5, 10 and 14-15. Level-two substrings are elements 1,2,3,6-7,9,11,12,13. The level-three uniting operator is element 8. Level-three substrings are elements 6 and 7. In practice the operators uniting elements of an elementary string are not thought of as uniting operators.

34 Variable String A string consisting entirely of variables and primes.

III.1.2 Notation

1 Operators may be represented by the letter f, either subscripted or not; by the Greek letters ϕ, θ and β, subscripted or not; by algebraic symbols and by numbers referring to Table 8. Multiplication in ordinary (not Polish) notation will sometimes be denoted by juxtaposition.

2 Functions may be represented by small letters or capital letters. They may be subscripted.

3 Strings may be represented by Greek letters other than ϕ, β and θ and by capital English letters. They may be subscripted.

4 Variables may be represented by single English letters at the end or the beginning of the alphabet. Monadic negation (prime) is represented for a variable x as x'.

5 Array dimensions are indicated by the number of cells in a row and column. The letters r and s, when not used for functions, denote respectively the row and column or the number of rows and columns.

6 Cell locations within an array are written as "cell(i,j)", where i is the row index and j is the column index. Numbering begins with cell(1,1) at the upper left hand corner of the array. Within a cascade cells are denoted by cell(j), where the numbering begins at the top of the cascade.

7 A mapping or transformation of one P-string into a functionally equivalent one will be designated where convenient by the single arrow: →. Mappings and transformations will also be denoted by NAME(PAR,f)=g, where NAME is the name of some procedure or transformation, PAR is a possibly empty vector of parameters and f is a vector of arguments.

8 APL vector notation may be applied to index strings. For example, if S is a string, S[i] is the ith element of S and S[⍳5] is the first 5 elements of the string[24].

III.2 Correspondence of P-Strings

In this section, justification is offered for using Polish notation, hereafter referred to as P-notation, when synthesizing cellular arrays. A unique correspondence is derived between arbitrary P-strings and certain cellular arrays.

With the advent of LSI and cellular arrays logic design moves decisively from a vector approach to a matrix approach, or said another way, from a one-dimensional to a two-dimensional circuit. Traditional two level logic problems consist typically of a set of AND gates with their outputs tied to an OR gate. For example, see Figure 15. This problem is almost one-dimensional. One could consider each product term as an item in one dimension and the number of levels as the second dimension. Conventional notation adequately describes the circuit. In contrast cellular arrays such as Figure 16 are definitely two-dimensional. If one writes an expression in infix for each column in an array, the function, g_i, of the ith column will appear as

$$g_i = ((wf_{1i}x)f_{2i}y)f_{3i}z.$$

For the related column at the bottom (the collector row)

$$f = (((1f_{44}g_4)f_{43}g_3)f_{42}g_2)f_{41}g_1.$$

Combining these, an expression for the array is

$$f = (((1f_{44}(((wf_{14}x)f_{24}y)f_{34}z))f_{43}(((wf_{13}x)f_{23}y)f_{33}z))\ldots$$
$$f_{41}(((wf_{11}x)f_{21}y)f_{31}z)$$

which is a rather cumbersome equation. The nesting of terms is a result of the fact that conventional notation needs parentheses to indicate the order of evaluation, which is independent of the particular function and dependent only upon position in the string.

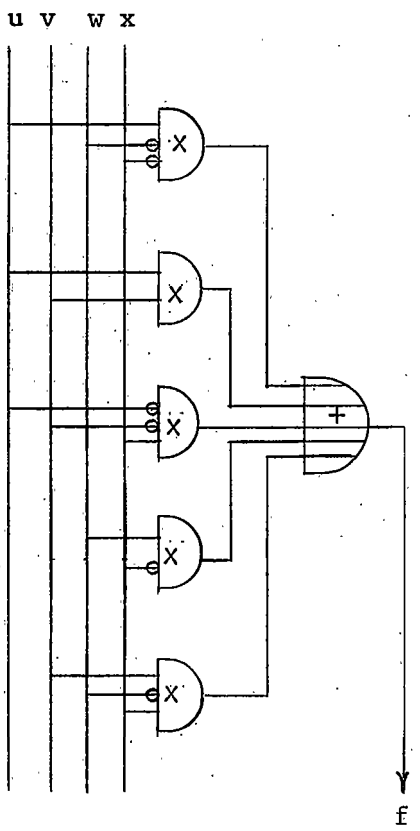


Figure 15. Two-Level Realization of f = uw'x'+uv+u'v'x+wx'+vw'x

In contrast, it will be shown that a P-string representation for this array is

$$f = zyxwf_{11}f_{21}f_{31}zyxwf_{12}f_{23}f_{32}\cdots f_{33}zyxwf_{14}f_{24}f_{34}1f_{44}f_{43}f_{42}f_{41}.$$

This expression is unencumbered with delimiters such as parentheses and it will be shown that it is immediately apparent what the array will be.

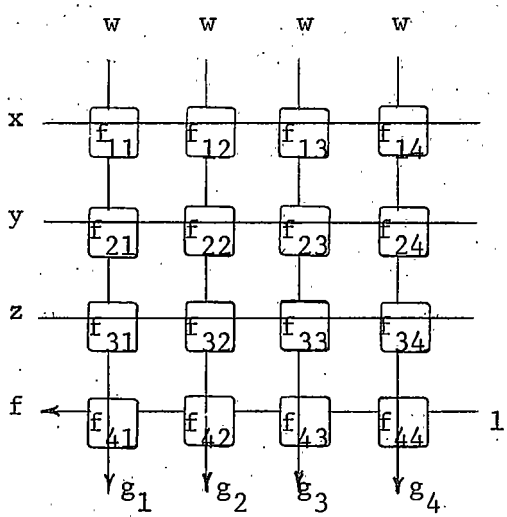


Figure 16. A Cellular Array

Minnick has observed that there is a natural correspondence between Polish strings of a particular type and Maitra cascades.*

*Private communication.

Consider the function, $f = (((vf_1w)f_2x)f_3y)f_4z$ which in P-notation is $f = zyxwvf_1f_2f_3f_4$. A Maitra cascade corresponding to this function is Figure 17. The cascade is determined by writing the variables of the elementary P-string, left to right, in order on the x or horizontal inputs of a cascade, beginning at the bottom and working up. The last variable is put into the y or vertical input of the top cell. The functions are the entered into the cells in order, the leftmost function specifying the top cell, and working down. The example generalizes by adding variables to the head of the P-string and appropriate operators to the tail. Thus any elementary P-string defines a cascade.

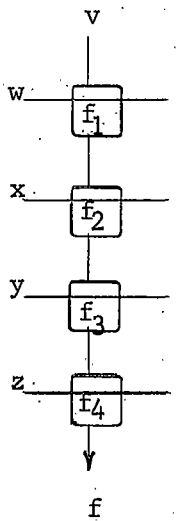


Figure 17. A Five Variable Example Cascade

The reverse of this is true. Any cascade defines an elementary P-string. One observes that beginning at the bottom of the cascade and proceding to the top writing the literals left to right and then appending the functions, left to right, yields the Polish representation of f. If the function of a cell is the degenerate function y (function 4 of Table 8), it is ignored as is the x input in writing the equation. The degenerate functions x, x', 1, and 0 are not considered since such a cascade with such a cell is equivalent to one made by deleting the portion above any of these functions. The degenerate function y' results in a complementation operator with one argument, the subfunction to the left in the P-string. The result is an elementary P-string. Thus one can see that there is a correspondence between elementary P-strings and Maitra cascades.* It is this property which makes P-notation useful for cellular synthesis.

If one restricts the operator set appropriately, one can similarly develop the cascades talked about by Levy Winder and Mott[12], Stone and Korenjak[30], Sklansky[28], Minnick[16] and Weiss[33].

To realize a function in one cascade one must merely write the function as a single elementary P-string. One can approach this using the heuristic method presented later in this chapter.

---

*The correspondence is not one to one since any number of cells with function 4 may be added to the cascade without changing the P-string associated with it.

It is now desired to show a correspondence between collector arrays and certain P-strings. First P-strings will be constructed from such an array. Applying the correspondence between P-strings and cascades to the collector row, the function of the array in Figure 16 can be written as

$$f = g_1 g_2 g_3 g_4 1 f_{44} f_{43} f_{42} f_{41}.$$

Substitution of the P-notation for the g_i's gives the original P-notation equation asserted as describing the array.

This equation can be described as five level-one substrings followed by four level-one uniting operators. Each level-one substring is an elementary string with variables appearing in the same order. The level-one substrings define the cascades while the uniting operators define the collector row. No restriction is placed on the size of the array. Thus any array of cascades with a collector row defines such a P-string with only level-one substrings, which are all elementary P-strings.

Reversing the procedure, it is desired to construct an array from a P-string with only level-one elementary strings and level-one uniting operators.

There are three steps to the procedure. To begin, a cascade for each elementary string is created and each output is put into a collector row. The leftmost elementary substring corresponds to the leftmost cascade, and so on. (Figure 18(a) illustrates this step for

f=wx'y++-vy'z+×w⊕+×. The function numbers correspond to Table 8.) Second, when primed variables occur on horizontal inputs, the cell function is altered to the one which complements that variable. This yields Figure 18(b). Starting with i equals 2, each cell of column i is checked, from the bottom cell to the top cell. Where the horizontal input of the cell of column i differs from any horizontal input to the left in the same row, a blank cell is inserted in that row in all columns to the left, pushing the former cells from the row and what lies above them up one row. New cells are set to function 4. After all columns are processed, cells are added on the top of short columns to make a rectangular array, the cells are set to function 4 and the vertical input is put into the top cell of the column. Figure 18(d) is the final cascade for the example. The buss lines are shown connected.

The third step is necessary only if a substring is primed. For example, consider

f=wx'y+'+vy'x+.

When the negation is encountered the function of the cell last specified must be changed to produce the complement of the function. Table 9 lists the function pairs for the interchange. The interchange is nothing more than an application of de Morgan's laws. Thus it

is shown that a collector array can be constructed from any P-string consisting of level-one terms which are all elementary.*

TABLE 8. FUNCTIONS OF TWO VARIABLES

Function Number	Function
0	0
1	1
2	x
3	x'
4	y
5	y'
6	x + y
7	x' + y
8	x + y'
9	x' + y'
10	xy
11	x'y
12	xy'
13	x'y'
14	x ⊕ y
15	x' ⊕ y
16	R-S flip-flop y = reset x = set

*In passing note that the correspondence which was just constructed between an array with Maitra cascades and a collector row and P-string with only elementary level-one substrings, is not one to one. If the array constructed from the last equation above were again reduced to an equation, wy'× would appear rather than w'y+', which are logically equivalent.

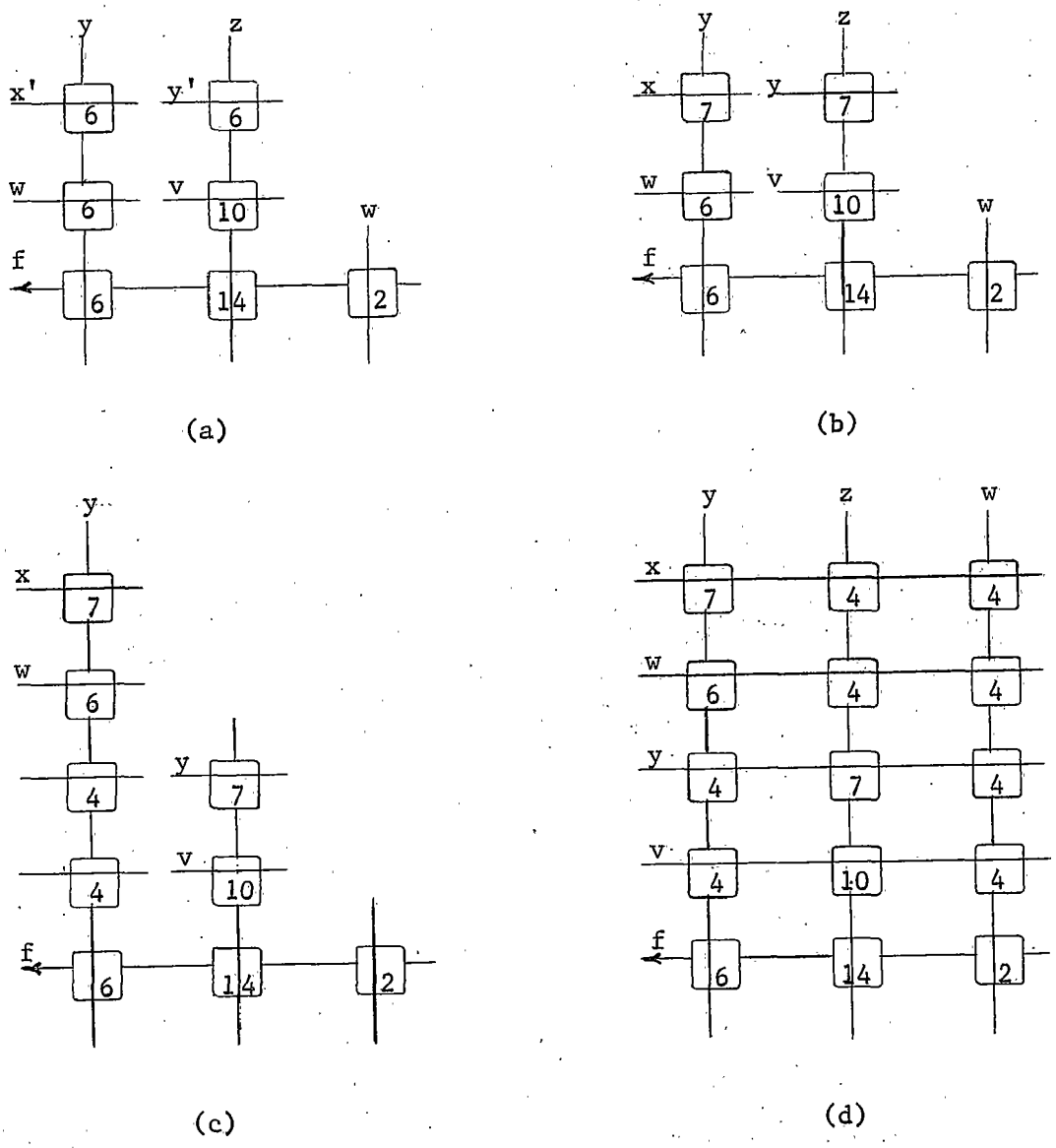


Figure 18. Embedding a P-String in a Collector Array

TABLE 9. OPERATOR PAIRS FOR COMPLEMENTATION OF PORTION OF CASCADE

Function	←Change to→ Number	Number	Function
0	0	1	1
x	2	3	x'
y	4	5	y'
x + y	6	13	x'y'
x' + y	7	12	xy'
x + y'	8	11	x'y
x' + y'	9	10	xy
x ⊕ y	14	15	x' ⊕ y

The above correspondence was derived assuming that the cells were general function cells. It is of interest to inquire whether a cutpoint cell could be used. The answer is that it can. The procedure is to construct the array as shown, and then transform the array to a cutpoint array. Minnick showed that this was possible by proving that any function realizable in a cascade having the same length and input and using cutpoint cells[16].

P-strings may also have level-n terms, where n is some integer greater than two. These strings do not correspond to cascade arrays with a collector row (collector arrays). However, a correspondence can be established between array styles like that of Figure 19 and arbitrary P-strings.

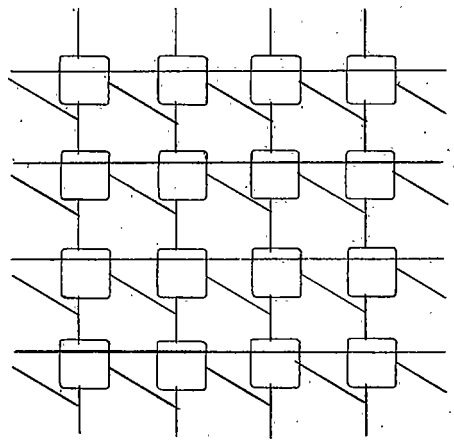


Figure 19. A Style 1 Array

Style 1 arrays, as presented in Figure 19, are defined as having general-function cells which can select as an x input either the horizontal buss or the diagonal input from the cell on the right. The y input is always from the cell above. A P-string can be constructed from the array using the algorithm which follows. In this algorithm,

if a diagonal input to a cell is used let this input data be denoted as g(i,j), where i and j signify the row and column of the cell from which the data comes.

1. Write the function for the cascade from which f emerges using the algorithm for writing elementary P-strings from cascades. Use g(i,j) as defined above whenever the x input is the diagonal input.*

2. Write functions for the cascades producing each g(i,j) using the algorithm for writing elementary P-strings from cascades.

3. If more g(i,j)'s are created in step 2, go to step 2 again and process the new g(i,j)'s.

4. Substitute in the expression for f the expression created for each g(i,j) which appears in the expression for f. If g(i,j) terms are contained in the expressions which were put into the function's representation, repeat this step until all g(i,j)'s are replaced by variables and operators.

The above algorithm will produce a P-string for the array in terms of the variables and the operators.

It is next desired to construct a Style 1 array from an arbitrary P-string. The construction is done in two stages. First, the function is embedded in a Style 2 array which differs from the Style 1 array in that the horizontal buss is broken, each cell having a connection to the outside world. See Figure 20.

---

*In the case where the x input is a g(i,j) and the cell function is one of the set {x,x'} a savings in effort can be realized by considering the cell from which the g(i,j) emerges as the next cell in the cascade. The result is the same since each g(i,j) is replaced in the algorithm.

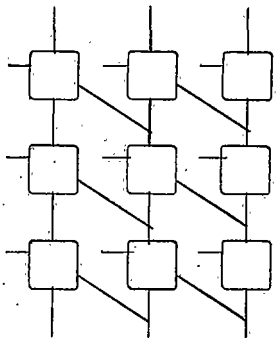


Figure 20. A Style 2 Array

The first stage is based upon the occurrence of elementary P-strings at some level. Each level-one substring is considered in turn. If it is complex, its substrings are considered in turn and collected together to form the level-one substring after they are realized. This procedure is iterated to a higher level if the level-two substrings are complex. The following algorithm describes the procedure in detail:

1. Let n=1. Let r= the number of rows in the array, numbering from top to bottom. Number each level-one substring from left to right beginning with number one. Number each level-two substring within a given level-one substring, from left to right, beginning with number one. Do this for each higher level. Define j_n as the number of the substring being considered at level n. Let j_n=0.

2. Increment j_n. If j_n-1 was the last substring in the current level-(n-1) substring, go to step 5. If substring j_n is elementary realize it using the algorithm for defining a cascade. Then go to step 2. If substring j_n is simple go to step 3. Otherwise, go to step 4.

3. Realize the simple substring as the vertical input to the cell in row r-(n-1) in the first vacant cascade of the array. Then go to step 2.

4. Increment n. Set j_n to 0. Go to step 2.

5. Form the collector row for the level-n terms realized and not previously collected. Use the previously presented algorithm for defining collector rows in collector arrays. Row n-(r-1) is the collector row. Go to step 6.

6. Decrement n by one. If n=0, go to step 7. Otherwise go to step 2.

7. Make all cascades of equal length by appending to the tops of shorter ones cells set to function 4. The variable that went into the top of the cascade is moved to the top of the modified cascade. Then, exit.

The function is now embedded in a style 2 array. If there are additional functions to be realized in the array, the realization of the next function is begun in the next vacant column of the array, and with step 1 of the algorithm.

An option is available to the designer at this point. There are additional style 2 arrays easily derived from this realization which may be more economical. For example, Figure 21 shows three realizations for the function

$$f = uvwx \times yz \times ts + + + +.$$

Figure 21(a) is formed by the above embedding algorithm. Figure 21(b) is a first alternative realization created by noting cascades with diagonal x inputs to the top cell that is not a function 4.*

To form this first alternate realization, the variable at the top of the cascade is moved to the x input of that cell. The subarray, consisting of cells with row numbers equal to or less than the chosen cell and column numbers greater than it, is moved one row up and one column left, leaving a row of empty cells to the right of the chosen cell. This subarray is shown enclosed by dashes in Figure 21. Figure 21(b) illustrates the result when this is done. A cascade has been eliminated and a row added. Stage two of the algorithm would have required such a row to be added, without the deletion of the column. Thus economy is served in this case by the modification.

A second alternative realization can now be formed. In this example, the extra row previously inserted can be eliminated (but it will be returned by stage two of the algorithm). Since cells(3,1) and (2,1) in Figure 21(b) are both function 6 which is symmetrical on its variables, it does not matter which one has the variable u for an input and which one has the diagonal input.** Figure 21(c)

---

*In Figure 21 and the sequel, diagonal x inputs are not shown except where they are actually used. This eliminates some clutter in the figures.

**If the function is not symmetrical, change the function appropriately. For example, $x' + y \rightarrow x + y'$.

shows the result when these are interchanged. This is done by sliding column one up one row, moving the variable and eliminating the vacant row. Figure 21(c) also corresponds to the style 2 array which would result from the form of f written as

$$f = uv+wx\times yz\times ts\times +++.$$

In some cases, stage two of the algorithm may not return the row eliminated, so the reduction is worth investigating.

Stage two of this algorithm is the transformation of the style 2 array into a style 1 array. To do this, it is necessary to line up the x input variables so that any particular row has the same variable (or none) as an x input for each column. This can be done by adding cells, set to function 4, in appropriate places. The following steps may be used to implement this algorithm:

1. For every cell, if the x input variable is primed, unprime it and change the function of the cell to the one which complements the input.

2. Let j=s-1. (s = the number of cascades in the array).

3. For each cell in column j, mark any cell which has a different x variable than the x variable input to that row in columns j + 1, j + 2, ..., s.

4. In column j, insert cells set to function 4 in the row of the bottommost conflict. Push the cascade, above and including the cell with the conflict, up until any conflicts caused by that cell are resolved. Do the same for each conflicting cell of column j, working from bottom to top.

5. Decrement j by one. If j=0, go to step 6. Otherwise, go to step 3.

6. Add cells set to function 4 to the top of short cascades, making all cascades equal in length. Move the y input from the top of the old cascades to the top of the lengthened ones.

7. For each row, buss all x inputs together. Then exit.

Thus an arbitrary Boolean function is embedded in a style 1 array. Figures 22 and 23 show several steps in the embedding of the function.

$$f = yx\times zvu\times+zv+0vw\times wzy\times\times++wvu00\times\times.$$

Figure 22(a) is the identification of the substrings by level. Figure 22(b) is the array produced for the first level-one substring. Figure 22(c) adds the realization of the first level-two substring contained in the second level-one substring. The actual realization of this substring takes place as two level-three substrings: uvz×+ and vz+. Note that these are realized upwards from row r-(n-1), in this case n being three. Figure 22(d) includes all of the second level-one substring. Figure 22(e) shows the entire realization. Note that an additional row has been added at the bottom to collect the level-one substrings.

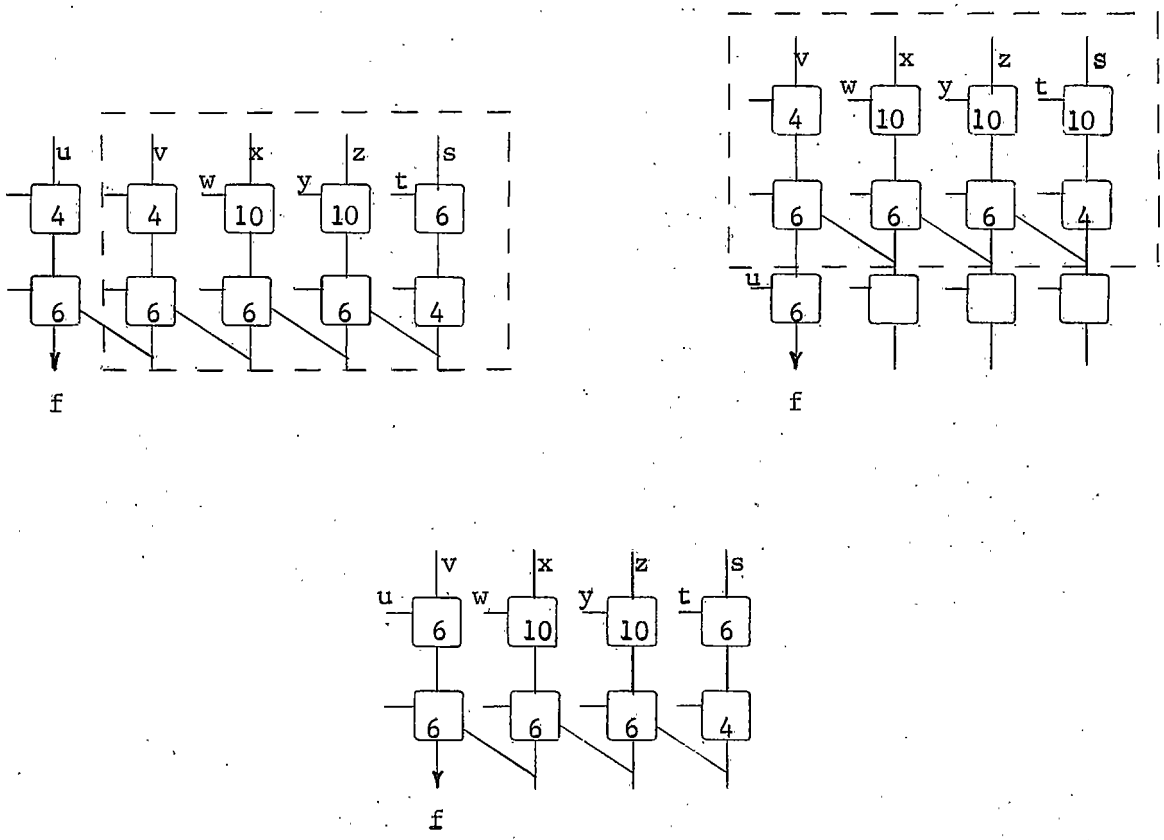


Figure 21. Embedding Example

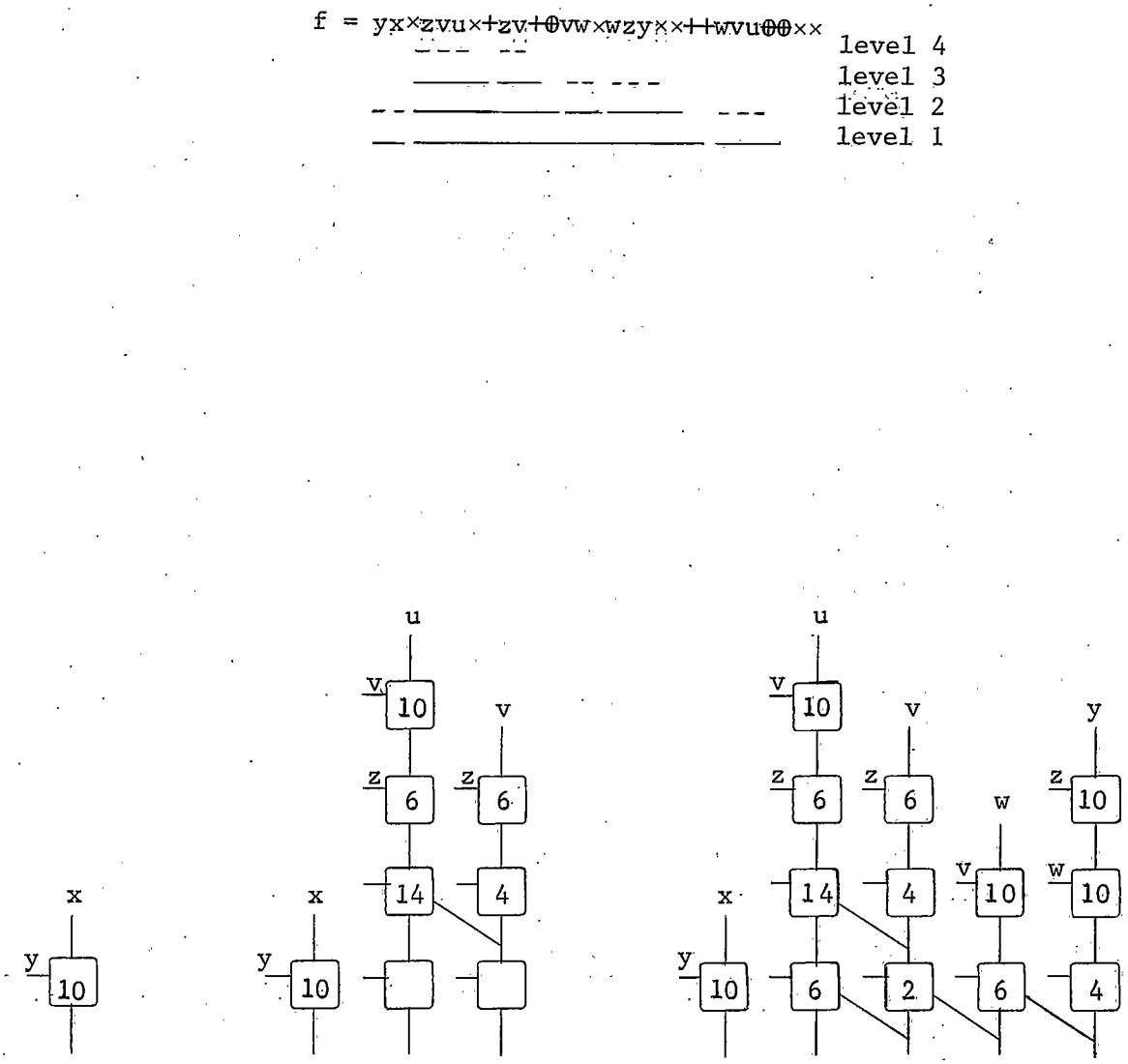


Figure 22. Embedding Example for a Style 2 Array

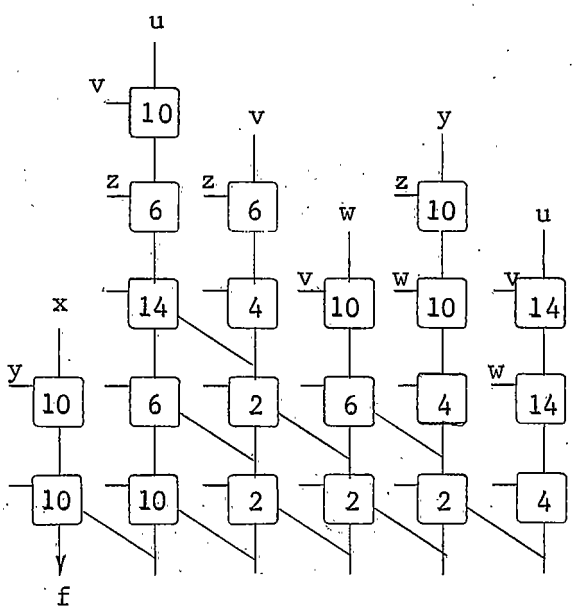
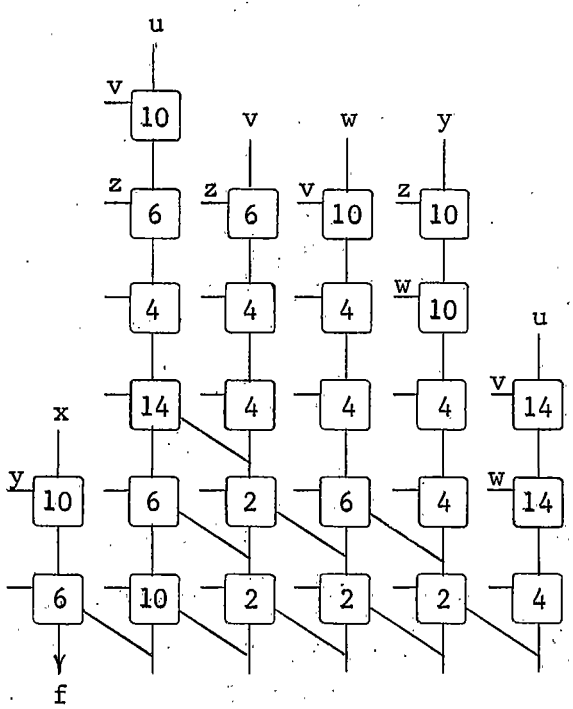


Figure 22. Continued.

Stage two of the process is shown in Figure 23. No primed variables are included in the function. The prime removal is illustrated in the example for collector arrays given before. Figure 23(a) shows the style 2 array after the columns have had blank cells added. Figure 23(b) shows the completed array, after all cells have been added to rows. Figure 23(c) shows the completed style 1 array having bussed x inputs.

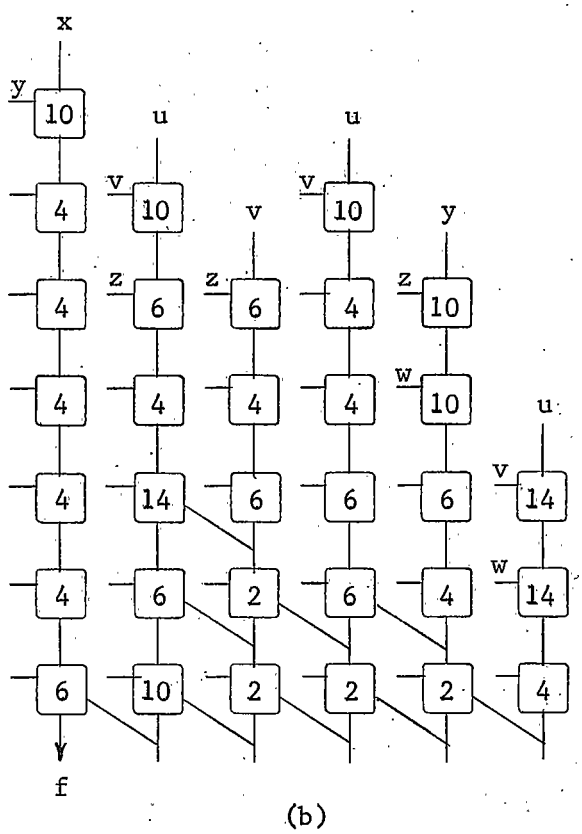
The array produced is not necessarily a minimal one. For example in Figure 23(c) if cell(5,6) is moved to cell(2,6) (with appropriate function 4's specified), cell(6,6) is moved to cell(4,6) and cell(1,1) moved to cell(6,1); then rows one and five can be eliminated yielding Figure 23(d). It may be possible to devise an algorithm to produce an array which is minimal, but the author is convinced that it would not be feasable. The multitude of possible cases covered would preclude a reasonably simple algorithm, and one could probably never be certain of covering all cases. As an alternative, as is done here, a simple workable algorithm is used, and the skill of the designer is relied upon to reduce the result. The end product is a reasonably efficient realization with moderate computational effort.

As mentioned for the cutpoint array, it is possible to restrict the general-function cell to the cutpoint functions (bringing a savings on hardware for each cell).



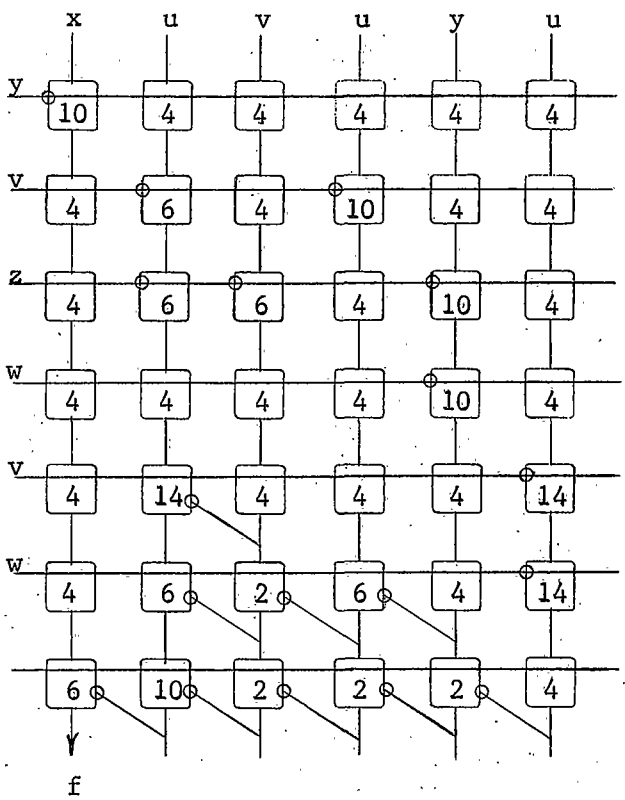
(a)

Figure 23. Synthesis Example for a Style 1 Array



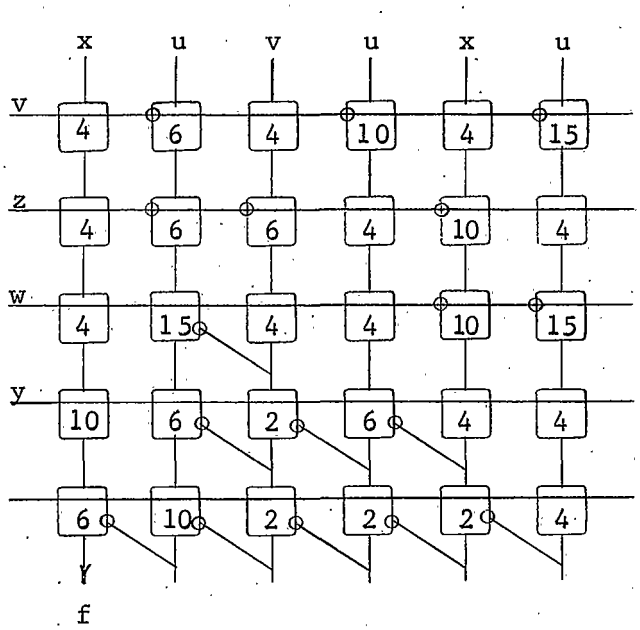
(b)

Figure 23. Continued.



(c)

Figure 23. Continued.



(d)

Figure 23. Continued.

In this section, it has been shown that any P-string can be embedded in a particular cellular array class based upon Maitra Cascades. This class of arrays includes Minnick's cobweb arrays. The simplest array of the class is the Style 1 array, shown in Figure 19. Conversely it is shown that any array has a P-string associated with and derived from it.

This realization of P-strings is reasonably efficient and forms a basis for the synthesis procedures of the last section in this chapter.

III.3 Manipulation of P-strings

If the conversion algorithm of Chapter II is combined with the embedding procedures of the previous section, one can realize an array for any Boolean function. However, such realizations may not be efficient. That is to say, perhaps the function can be realized with a smaller array having fewer rows or columns or both. If this can be done, one can say that he has a lower-cost realization.

In the system presented previously, each elementary P-string is realized in a cascade (column) of an array. Thus, reducing the number of cascades is equivalent to reducing the number of elementary P-strings. The number of rows is dependent upon three factors: the highest level number for substrings, the number of different variables in the function, and the ordering on the variables. Each increment of the level structure requires an extra row for collection of cascades. Thus, to reduce the highest level by one saves one row. In most cases, each distinct variable requires one or more row busses.* Thus elimination of a vacuous variable usually saves at least one row. As pointed out in Figure 23, variables appearing with different orderings within the substring may cause redundant entering

---

*An exception can occur if new variables appear as vertical inputs. The extreme case would be a function such as f=vw×vx⊕vy+vz×+|+ which has two rows and four cascades.

of variables. Ordering the variables within elementary P-strings will save rows, though some redundancies may remain.

An example illustrating these points is shown in Figure 24. One of the simplest functions of four variables not cascade realizable is

$$f_1 = \Sigma(3,6,7,11,12,13,14,15).$$

If f_1 is written in a sum of minterms and translated to P-notation, it can be written as

$$f_1 = v'w'yz\times\times\times v'wyz'\times\times\times v'wyz\times\times\times vw'yz\times\times\times vwy'z'\times\times\times vwy'z\times\times\times$$
$$vwyz'\times\times\times vwyz\times\times\times+++++++$$

which maps into eight cascades each producing a minterm as in Figure 24(a). If iterated concensus is performed, the prime implicants are all found to be essential and f_1 reduces to

$$f_1 = wv + wy + yz$$

which translates to P-notation as

$$f = wv\times wy\times yz\times++$$

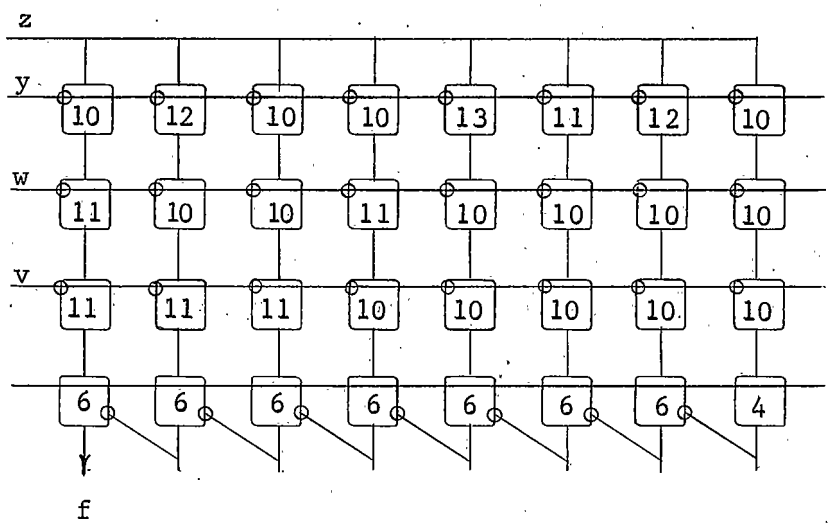
This form is realized in Figure 24(b). If the prime implicants are factored as

$$f_1 = yz + w(y + v)$$

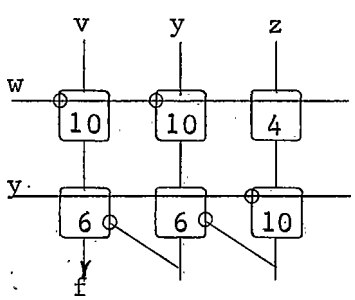
translation to P-notation yields

$$f_1 = yz\times wyv+\times+$$

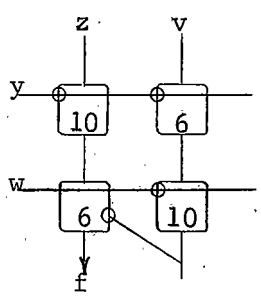
which maps into two cascades as in Figure 24(c).



(a)



(b)



(c)

Figure 24. Realizations of f = Σ(3,6,7,11,12,13,14,15)

Application of a realizability test from Chapter II will show the function not realizable in one cascade. Since no variables are entered twice, this realization seems to be as efficient as possible.

Several processes occurred in this example. Iterated concensus eliminates occurrences of variables as well as any vacuous ones. It also reduces the number of elementary strings and the highest level of substrings. The common methods for doing iterated concensus also result in an ordering of the variables. The factorization puts a substring realized in two cascades into a form realizable in one cascade.

III.3.1 Manipulative Procedures

It is desired, as pointed out above, to manipulate P-strings into more desirable forms. To this end, the author has developed a set of procedures which he has used successfully to produce efficient designs for many functions. Some of these procedures are presented as examples. The application of the procedures is heuristic in nature, calling upon the expertise of the user in the same manner as one would expect in reducing ordinary Boolean equations by hand. The procedures are divided into three categories: test, manipulate, and display, which are listed in Table 10 with brief explanations. In the paragraphs following each procedure is treated in detail. In Table 10,

the procedures are listed for the convenience of the user. For descriptive convenience, the procedures are divided in another way in Table 11, into basic procedures, procedures using other procedures and multiple procedures. The procedures are described in the order listed in Table 11.

TABLE 10. MANIPULATION PROCEDURES

Category	Name & Arguments	Description
Test	EQUAL(S,R)	Test substrings S and R for Boolean equality.
	COMP(S,R)	Determine if S = R'.
	CR(f)	Determine if f is cascade realizable. If it is, put it in that form.
Display	SEPARATE(N,f)	In P-string f insert $\sum_{i=1}^{n} i$ spaces between level-n substrings for n = 1,2,...N. N = 0 closes all spaces.
	POSITION(N,f)	In P-string f begin each level-n substring on a new line for n = 1,2,...,N. Indent substrings n spaces. Place uniting operators on line after last substring united with spaces between levels of uniting operators. N = 0 returns f to normal orientation.

TABLE 10. CONTINUED

Category	Name & Arguments	Description
Manipulate	COMMUTE(v,H,f)	Commute where possible variable v to the head (or tail if H is replaced by T) of each elementary P-string of f in which v or v' appears.
	CONVERT(f)	Translate the infix form of f into P-notation.
	CONC(f)	Do the iterated concensus algorithm on f.
	CR(f)	See entry under test category.
	DNF(f)	Put f into a disjunctive normal form.
	ELRULE(n,f)	Perform transformations n from Table 13 on f.
	EXCH(S,R)	Exchange the two substrings S and R.
	FACTOR(S,f)	Factor all forms of S out of f where S is a conjunction of variables. Use DNF first.
	MOVEOP(f)	Where possible move an operator towards the tail of f.
	NOPLUSE(f)	In f, eliminate all ⊕'s by WY⊕→ WY'×W'Y×+.
	ORDER(V,f)	Order variables and substrings in f, starting at the highest level, by the ordering of variables specified by the vector V. No V implies alphabetic ordering.

TABLE 10. CONTINUED

Category	Name & Arguments	Description
Manipulate (Cont.)	PLUSE(f)	Look for substrings in f of the form $WY\times Z_1Z_2\ldots Z_nW'Y'\times+\ldots+$. Transform them to $Z_1\ldots Z_nWY'\oplus+\ldots+$.
	PLUSET(f)	Look for substrings of the form $WYX\times\times Z_1\ldots Z_n\ W'Y'X\times\times Z_{n+1}\ldots Z_m+\ldots+$. Allow X, Y and Z to occur in any permutation during the search. Reform as $XWY'\oplus\times Z_1\ldots Z_m+\ldots+$.
	REED(v,f)	Decompose f into the Reed form for the variable v.
	SHANNON(V,f)	Decompose f into the Shannon form for the variables in V.
	SUB(n,S,R,f)	Substitute R for the nth (n = 0 implies all occurrences) of S in f.
	SUBSUME(f)	Remove all elementary P-string covered by others.

TABLE 11. DESCRIPTIVE CATEGORIZATION OF MANIPULATIVE PROCEDURES

Type	Name	Type	Name
Basic Procedures	EQUAL	Multiple Procedures	ELRULE
	COMP		
	SEPARATE	Procedures using	
	POSITION	other Procedures	PLUSE
	COMMUTE		PLUSET
	CONVERT		DNF
	CR		CONC
	EXCH		FACTOR
	MOVEOP		INF
	ORDER		SHANNON
	NOPLUSE		REED
	SUB		
	SUBSUME		

EQUAL and COMP respectively test their arguments for equality or Boolean complementation. This is useful if the user desires not to determine this by inspection. There are at least two methods for performing these tests. For EQUAL, the first is to use the elementary algebraic manipulation to transform the first argument into the second, or into a form easily recognizable as not equal to the second. A second method is that of an exhaustive search. Each expression is evaluated for all combinations of values for the variables. If the two arguments always have the same value, they are

equal. Similarly if they are always opposite in value, they are complements; otherwise they are neither. An algebraic manipulation like that of the first method could be used for complementation also. Figure 25 is a flow diagram for the exhaustion method. The actual evaluations may be done using the algorithm presented in Chapter II.

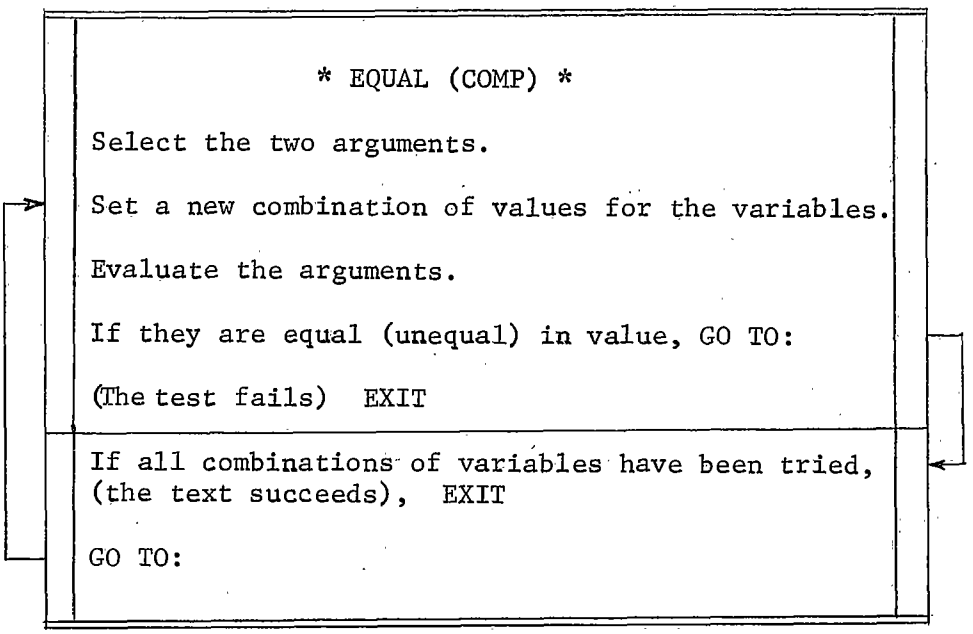


Figure 25. Flow Diagram for EQUAL and COMP

SEPARATE and POSITION are used exclusively to help the user view the function. It is possible that a different physical placement on paper in terms of the level structure will help the user identify useful manipulations. This is so because substrings at a

given level tend to be associated together before substrings at a lower level are considered. SEPARATE puts spaces between substrings which are correlated to the number of the level. The user may decide what the number of spaces should be. The author has found it convenient to separate substrings up to level N according to the sequence: 1, 3, 6, 10, ..., with one space between level-one substrings three between level-two substrings and so on.* POSITION starts each substring up to level N on a new line. Uniting operators for level-n are positioned just past the largest level-n term. Uniting operators at each lower level are placed just to the right of the last operator at the next higher level. A group of uniting operators is placed horizontally on the same level with the last term in the substring which it affects.

COMMUTE is the procedure of moving a variable v to the head of any elementary P-string in the argument. This can be done if all the operators in the P-string after and including the one for v are the same. The variable can then be simply moved to the head.

CONVERT is used to change a function written in the regular notational style into P-notation. An algorithm which can be used was presented in Section II.2.2.

---

*This sequence can be described by: $s_1 = 1$, $s_n = s_{n-1} + n$, where s_i is the ith term of the sequence.

CR is any cascade realizability test which the designer may wish to use. The weight synthesis test for cascade realizability was presented in Section II.1.5. The result of this procedure is either a FALSE or an elementary P-string which can be realized in a cascade, as done in Section III.2.

EXCH enables the user to alter the P-string by exchanging any two substrings. This is an arbitrary procedure which can change the value of the P-string, unless it is used where commutation is possible.

MOVEOP is a procedure to find every level-n operator which is the last operator of its level-n substring and which is identical to the level-(n-1) uniting operator associated with that substring. The operator is then moved to a point immediately after its uniting operator. Examples are given in Table 12 and can be seen to be an exercise in commutativity, valid for operators +, × and $\oplus$.

TABLE 12. EXAMPLES OF USE OF MOVEOP

No.	P-string	P-string after MOVEOP
1	ab+c+	abc++
2	ab×c+de×+	ab×cde×++
3	ab×ac×+cd×+de×+	ab×ac×cd×de×+++
4	ab×c×ef×+gb×+	abc××ef×gb×++

In Table 12, (1) shows the simplest use of MOVEOP, in which the commutativity of + is clearly invoked. This extends to (2) where the terms are more complex. Example (3) would result if a sum of product terms in regular notation was operated upon by CONVERT. The P-string after MOVEOP is easier to look at and consists entirely of elementary level-one substrings, (corresponding to a cutpoint array). Example (4) shows a simple case where MOVEOP causes changes at more than one level. In this case, it is satisfactory to treat the highest level first, the next highest second, and so on. In all cases, the number of higher level terms is decreased. In all cases but (3), the number of columns needed to realize the function is reduced.

One of the problems in realizing cellular array as mentioned in Chapter II is that of ordering the variables. For instance the P-string

$$f = vw\times wy\times yz\times++$$

is not realizable in a single style 1 array unless the variables w and y are introduced twice. This is illustrated in Figure 26. The algebraically equivalent P-string

$$f = wv\times wy\times yz\times++$$

is realizable with one less row and with only y entered redundantly as illustrated in Figure 24(b).

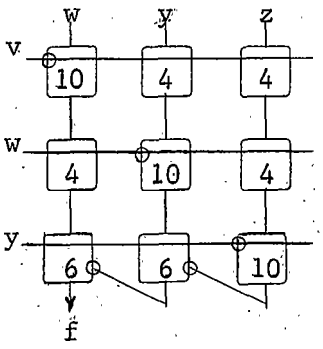


Figure 26. Example Realization for the Use of ORDER

The designer begins the ORDER procedure by defining an ordering of the variables. An ordering vector V is written as V = w,x,y,z. Where w,x,y and z are variables. The ordering means that V[s] is to appear in the P-string whenever possible, before V[t], when s < t. Starting with the highest level substrings, the order of variables is altered to conform with the selected order. This is done by commuting operands, making use of the commutative law. If there is a mixture of uniting operators for one substring, the ordering cannot be complete. For example, if the order is defined as a,b,c, then cba×× can be ordered as abc×× but cba×+ cannot.

In all but the highest level substrings, the ordering will be an ordering of substrings within a level-n substring. In these cases, the substring can be treated as a dictionary "word" and each

substring placed in its alphabetic order as defined by the ordering on the variables.

NOPLUSE is a procedure for removing all ⊕ operators. It is accomplished by locating the arguments of each ⊕ and rewriting that portion of the P-string. For example, XY⊕ becomes XY'×X'Y×+, which is easily recognized as the usual definition for the ⊕ operator. If a prime follows the ⊕, it is easiest to move the prime to one of the substrings before performing the transformation. Thus XY⊕' becomes XY'⊕ or X'Y⊕ which in turn becomes XY×X'Y'×+ or X'Y'×XY×+.

SUB is a procedure for arbitrarily changing a P-string. This allows the user to do anything he feels is worthwhile, including change the function of the P-string. The form of the procedure is defined in Table 10.

SUBSUME is used in many ways. It is part of the iterated concensus algorithm and may be useful as a separate algorithm. A substring is said to subsume another substring if whenever the second string has a true value, the first one does also. In this case, the first string can be removed without affecting the function produced. For example, in

$$f = wy\times wzv\times\times zv\times++$$

the substring wzv×× subsumes the substring zv×. Thus the function can be rewritten as

$$f = wy\times zv\times+,$$

saving one column in the realization. In particular, in a string of substrings +'d together where the substrings are elementary, with × operators, when a substring has all of the variables of another substring, the first one can be eliminated. The dual also holds (interchange × and + in the above.)

ELRULE is an expression of the basic Boolean laws as illustrated in Table 13, and is the one procedure which is made up of many procedures. The notation includes an argument which specifies the particular rule used. Thus

$$\text{ELRULE}(i,f) = g$$

implies that the ith transformation in Table 13 is applied to the P-string f yielding the P-string g.

Each of the transformations in Table 13 is well known in infix notation. The transformations in the fifth column are applied to the P-notation strings. Note that S denotes an entire P-string. The P-notation versions can be verified using the evaluation algorithm presented in Chapter II. For example, the first complementation rule says w plus w prime equals unity. The P-notation reads exactly the same way.

TABLE 13. FUNDAMENTAL ALGEBRAIC LAWS OF ELRULE IN P-NOTATION FOR THE OPERATOR SET {×,+,⊕,'}.

#	Name	Infix	P-Notation	Transformation
1	Commutation	wϕy=yϕw	wyϕ=ywϕ	S[ι3]→S[2,1,3]
2	Association	wϕ(yϕz)=(wϕy)ϕc	wyzϕϕ=zwyϕϕ	S[ι5]→S[3,1,2,4,5]
3	Distribution*	(wθy)ϕ(wθz)=wθ(yϕz)	wyθwzθϕ=wyzϕθ	S[ι7]→S[1,2,5,7,6]
4	Tautology**	wϕw=w	wwϕ=w	s[ι3]→S[1]
5	Absorbtion**	wϕ(wθy)=w	wwyθϕ=w	s[ι5]→S[1]
6	Complementation	w+w'=1 w⊕w'=1 ww' =0	ww'+=1 ww'⊕=1 ww'×=0	S→1 S→1 S→0
7	Pluse Definition	wy'+w'y=w⊕y	wy'×w'y×+=wy⊕†	S[ι7]→S[1,7],⊕
8	Counter Absorbtion*	wϕ(w'θy)=wϕy	ww'yθϕ=wyϕ	S[ι6]→S[1,4,6]
9	DeMorgan	(w+y)'=w'y' (wy)'=w'y'	wy+'=w'y'× wy×'=w'y'+	S[ι4]→S[1,4,2,4],× S[ι3]→S[1,4,2,4],+
10	Operations with 0 and 1	0w=0 1w=w 1+w=1 0+w=w 0⊕w=w 1⊕w=w'	0w×=0 1w×=w 1w+=1 0w+=w 0w⊕=w 1w⊕=w'	S→0 S→w S→1 S→w S→w S→w'

*Only holds for ϕ, θ ε {×,+} or ϕ=⊕, θ=×. **Only holds for ϕ ε {×,+}.

†Transformation depends upon the way primes are entered.

The remaining procedures of Tables 10 and 11 are those which use other procedures. This usage varies from the simple step of looking for complements (COMP) when creating $\oplus$ gates with PLUSE, to the relatively complex usage of several other procedures by CONC.

PLUSE is a procedure to look for a substring which has the form $wX\times Z_1Z_2\ldots Z_nw'Y\times++\ldots+$ where $X = Y'$, and rewrite the substring as $Z_1\ldots Z_nwX'\oplus+\ldots+$ (Z_i may all be vacuous). This procedure requires testing Y and X for the complementary property, as discussed earlier in this section for the procedure COMP. The procedure also requires locating at some level a sequence of substrings +'d together. PLUSE can fruitfully be generalized by allowing the variable w to be a substring W. A flow diagram for the generalized procedure is presented in Figure 27.

PLUSET is another procedure for creating $\oplus$ operators. Referring to Table 10, it will sometimes occur that a substring, X, can be factored out of a portion of a larger substring, leaving a set of substrings which can be shortened by introduction of a $\oplus$. If a substring meeting the criterion listed under PLUSET in Table 10 is found, the transformation takes place in three steps. First all the Z_i substrings are commuted to the tail of the P-string. This yields: $WYX\times\times W'Y'X\times\times Z_1\ldots Z_m+\ldots+$. Next X is factored out yielding: $XWY\times W'Y'\times+Z_1\ldots Z_m+\ldots+$.

Then the nine characters to the right of X reduce to XY'⊖, for the final form. Of course, in doing this one is making use of COMP, EQUAL and COMMUTE.

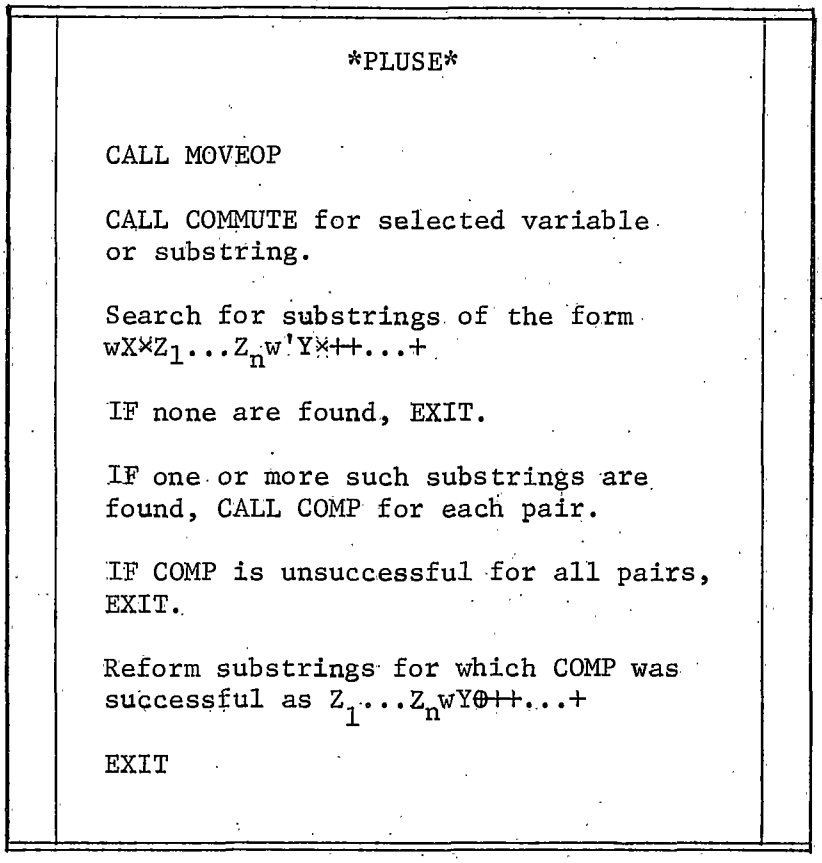


Figure 27. Flow Diagram for PLUSE

The object of the DNF algorithm is to put its argument string into disjunctive normal form. If all the +'s are placed at the end of the string, this results in a sequence of elementary level-one terms. This algorithm has several uses. It is a starting point for the iterated concensus procedure and for the Shannon and Reed Decomposition procedures. It may also be useful for visual inspection, as the result is a very uncomplicated form. The DNF algorithm makes repeated use of the transformations in Table 14, MOVEOP and ELRULE(4). Note that the arguments could be complex substrings.

TABLE 14. TRANSFORMATIONS USED IN DNF ALGORITHM

#	Infix	P-Notation
1	(w+y)z → wz+yz	wy+z× → wz×yz×+
2	z(w+y) → zw+zy	zwy+× → zw×zy×+
3	w⊕y → wy'+w'y	wy⊕ → wy'×w'y×+
4	(w⊕y)' → w⊕y'	wy⊕' → wy'⊕
5	(wy)' → w'+y'	wy×' → w'y'+
6	(w+y)' → w'y'	wy+' → w'y'×

Putting a function in disjunctive normal form is equivalent to manipulating the P-string of that function until it is a sequence of elementary level-one terms with all level-two operators as OR's and all level-one uniting operators as AND's. A method of producing a P-string in this form from any arbitrary P-string is described in Figure 28.

Note that SUBSUME is not strictly needed, but without it the function may be left needlessly complex, as indicated by the first example immediately following.

In Figure 29, two examples are given illustrating the use of the DNF algorithm as outlined above. In Figure 29, the affected substrings are underlined in each step, and the transformation numbers refer to Table 14.

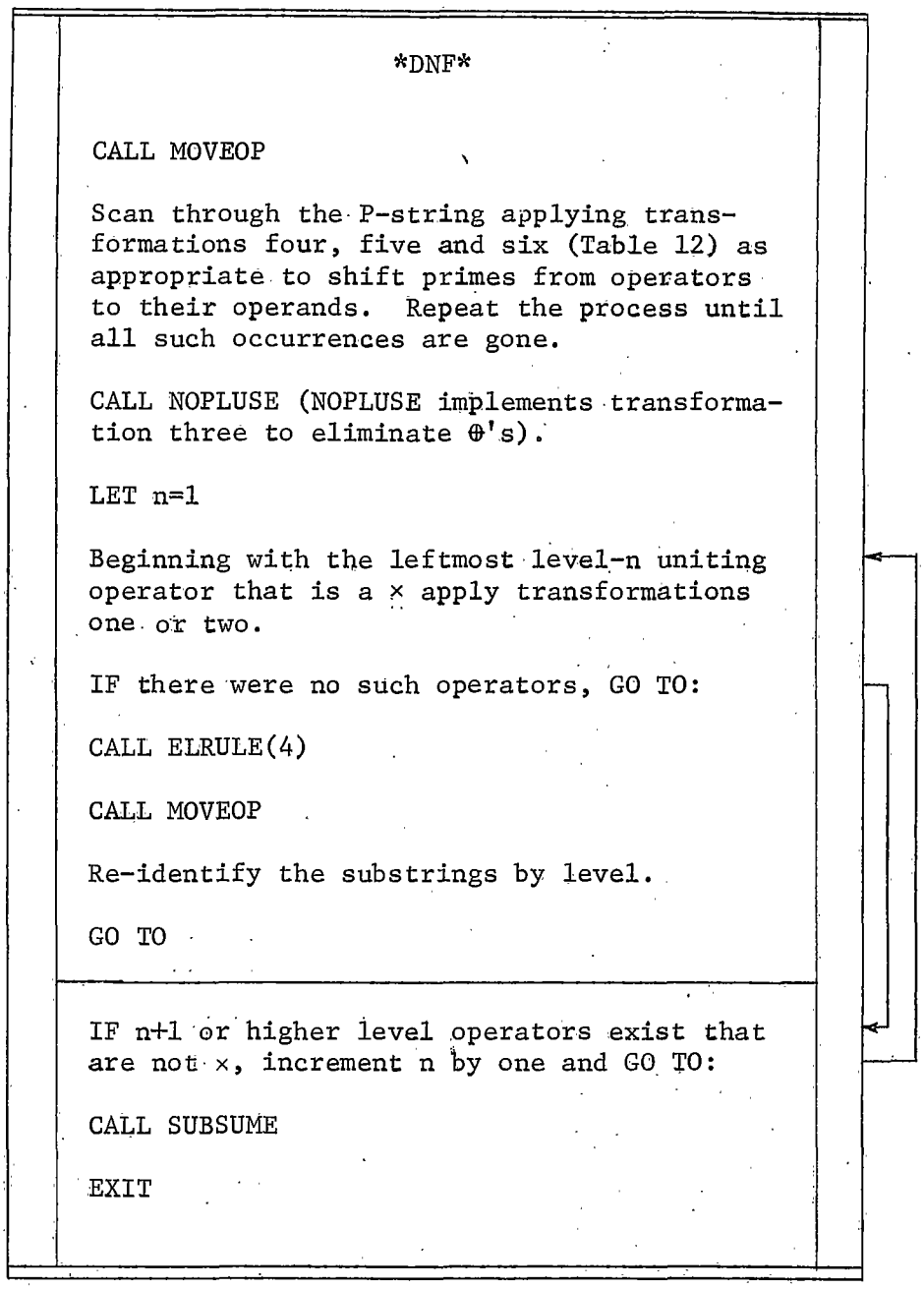


Figure 28. Flow Diagram for DNF

Trans-formation	P-string Before Transformation	Comment
MOVEOP	vy+uz+×zv××	No ⊕'s or primed operators.
1	vy+uz+zy×××	
ELRULE(4)	vy+uzv××zzv××+×	
2	vy+uzv××zv×+×	
1	vy+uzv×××vy+zv××+	
ELRULE(4)	vy+uzv×××vzv××yzv××++	
1	vy+uzv×××zv×yzv××++	
MOVEOP	vuzv×××yuzv×××+zv×yzu××++	
ELRULE(4)	vuzv×××yuzv×××zv×yzu××+++	
SUBSUME	uzv××yuzv×××zv×yzv××+++	
	zv×	
4	ab⊕'c'd×'+	New string with a ⊕.
5	ab'⊕c'd×'+	
NOPLUSE	ab'⊕cd'++	
MOVEOP	ab×a'b'×+cd'++	
	ab×a'b'×cd'+++	

Figure 29. Examples of DNF Procedure

The first step of the DNF algorithm is to perform the MOVEOP procedure. As shown in line one (Figure 29), there is one operator to be moved, producing line two. There are no primes to be shifted from operators to operands and no $\oplus$ operators to be eliminated in this example. In line two, transformation one is applied to the underlined substring. It is the only one applicable, (note that the first $\times$ is not considered a uniting operator since it is part of an elementary P-string), and the result is line three. The algorithm then calls for an elimination of any tautology produced in this step by ELRULE(4). The result is line four, which has no operators that can be moved by MOVEOP. The entire P-string is used in the next manipulation, and either transformation one or two can be used. Two is selected arbitrarily, yielding line five. ELRULE and MOVEOP produce no change. Only transformation one applies to line five, yielding line six, which does have a tautology. This is removed using ELRULE(4), producing line seven. At this point all level-one uniting operators are +'s, so the algorithm calls for consideration of the level-two uniting operators. The leftmost level-one substring is considered first. Only transformation one applies, yielding line eight. As will be the case with higher level uniting operators, MOVEOP can move the newly created operator to the tail of the string, reducing it to a level-one uniting operator.

Line 10 is the result after the tautology is removed from line nine. No level-two or higher uniting operators remain, and all level-one uniting operators are +'s. The algorithm now calls for SUBSUME and the reduction is complete. The function is in a disjunctive normal form.

The second example is similarly done, except that it illustrates the removal of $\oplus$ operators, and the removal of primes from operators.

Note that the result in example two does not conform to the criterion for completion if cd'+ is considered as a level-one term instead of two level-one terms with uniting operator. The definition of a level-n term implies that the first alternative is the correct one. For purposes of the DNF algorithm this is ignored. Such a substring ending in a + could appear elsewhere as an elementary string. The final MOVEOP will assure that the + is moved to the tail portion of the P-string.

The convergence of this algorithm is not immediately obvious. However, it can be argued as follows. After each pass through the level-one uniting operators, it is guaranteed that the last operator processed is a +. Transformations one and two add no new level-one operators, so that at most the number of passes at level one is the number of level-one uniting operators of the original string

modified by MOVEOP. Transformation one adds no new substrings so that if only transformation one is used, iteration up the level structure will terminate with the desired result. Transformation two creates an additional level-one substring. However, the new substrings are simpler than the old by the absence of one variable each. Thus using transformation two, one ultimately reaches elementary substrings and the procedure terminates.

Referring again to Tables 10 and 11, the iterated concensus algorithm, here called CONC, is well known. The result of this algorithm is the prime implicants of the function. For purposes here, the result of CONC will be written as though the result of the iterated concensus were written in infix and run through CONVERT and MOVEOP. The result is that all ×'s appear at ends of level-one terms and all +'s appear at the tail of the P-string. Additionally, published versions of iterated concensus assume that the function appears in disjunctive normal form with an ordering on the variables[14,15]. Thus the procedures DNF and ORDER are used first as shown in Figure 30.

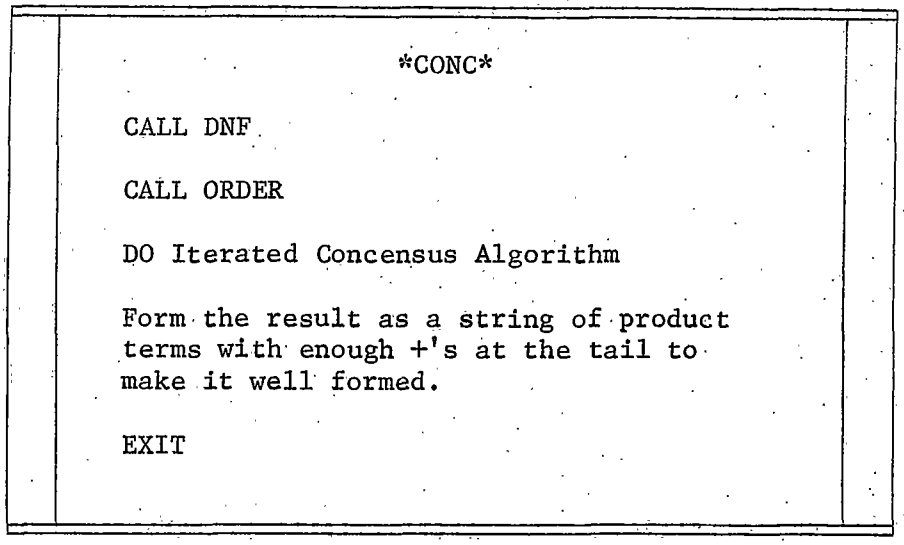


Figure 30. Flow Diagram for CONC

FACTOR is a decomposition procedure defined for a P-string in a disjunctive normal form. A n variable substring is chosen with its × operators. The string is rewritten as a sum of 2^n+1 substrings with the first 2^n having the chosen substring at its head with a particular priming structure on its variables. The last one consists of level-one terms not containing the chosen substring. For example, if

f = v'wx××v'zw'y×××v'y'w××v'w'zu×××wzux×××++++

then applying FACTOR to f with vwx for the argument yields

f = v'w'× zy×zu×+ ×

v'w× xy'+ ×

wzux××× ++.

The result is written with a line for each substring having a particular priming structure of the chosen string. Spaces are inserted between the factor and its substring and are used to set off the uniting operators for visual clarity. The POSITION procedure can be used to restore the P-string to one line.

INF results in an irredundant normal form for its argument. The function is assumed to be in disjunctive normal form and the iterated concensus algorithm is assumed to have been performed, leaving only prime implicants. An INF is defined as a disjunction of prime implicants, covering the true vertices of the function, which has no implicants that can be removed without uncovering some true vertex. Algorithms for this procedure are well known to logic designers and the reader is referred to[14] for two of them. The result will be assumed to be written as a string of P-notation product terms followed by enough +'s to make the P-string well-formed.

SHANNON and REED are respectively procedures to perform the well known SHANNON and REED decompositions on a P-string. Figure 31 is a flow chart for SHANNON. The algorithm can be explained as follows. Factor the selected variables out of all terms in an irredundant normal form of the P-string. Rewrite the string as a sum of substrings each of which is a sum of product terms, independent of the selected variables, multiplied by the variables.

Each product term in one sum results from factoring out the selected variables with a particular priming structure. This form can easily be seen to be equivalent to the well known form of the Shannon decomposition (expansion) theorem.

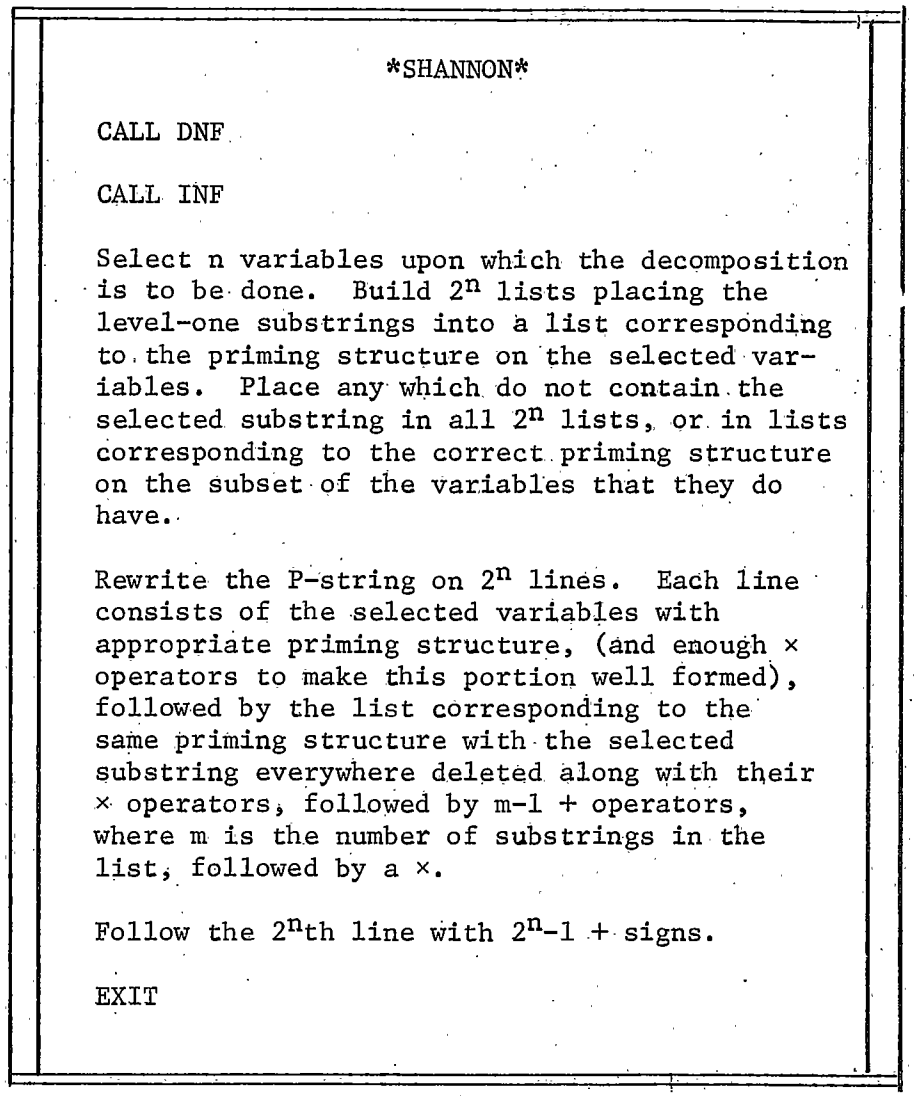


Figure 31. Flow Diagram for SHANNON

In the case where one variable is selected, it is useful to form an additional decomposition, attributed to Reed.* This form is well known to be

$$f = wH \oplus J$$

where H and J are substrings independent of w. Further, $H = G_1 \oplus G_0$ and $J = G_0$ where G_0 and G_1 are from the Shannon form, written as

$$f = wG_1 + w'G_0.$$

In P-notation, the Reed form appears as

$$f = JwH\times\oplus = wH\times J\oplus = wG_1G_0\oplus\times G_0\oplus.$$

REED is the name of the procedure which produces the function in this form. It is accomplished by rewriting the Shannon form as shown above.

The procedures detailed above are ones that the author has found to be useful in producing reasonably efficient cellular arrays. Included is a set sufficient for realizing all functions in style 1 arrays. Others are included but not presented as a closed set. It is expected that when large quantities of problems are solved by a user, he may develop additional procedures to add to the list. Certainly any computerized version of this system should be implemented with this in mind.

---

*The Reed form can be derived from the Shannon form as follows: $f = wG_1 + w'G_0 = wG_1 \oplus w'G_0 = wG_1 \oplus (1 \oplus w)G_0 = w(G_1 \oplus G_0) \oplus G_0 = f$.

III.3.2 Examples of P-string Manipulation

In this section, several synthesis examples are presented. In each example, a function is given as some algebraic string of literals and operators in standard notation. This form would probably result from word statements or logical design problems. If one wanted to consider some other form of representation as an input to the system, it should be a fairly simple matter to translate any form, for example Cauldwell notation, into an algebraic string.

The examples are produced using only the procedures of the previous section, with actual style 1 arrays, for each problem. Explanation of the reason for choosing a particular procedure is included. In the example arrays, numbers in cells refer to function numbers from Table 8. The x input connection, if any, is circled. Due to the limitation of 8½ by 11 paper, some P-string which would normally appear on one line are written on two or more.

Example 1: The Half Adder. A half adder and its table are illustrated in Figure 32. From the truth table the two functions are

$$f = xy$$

$$g = x'y + xy'.$$

Choosing f, the following manipulation is made.

$$\text{CONVERT}(f) = xy\times$$

This form takes one cell and can be called optimal since no fewer cells can be used. Choosing g, the procedures CONVERT and PLUSE are applied.

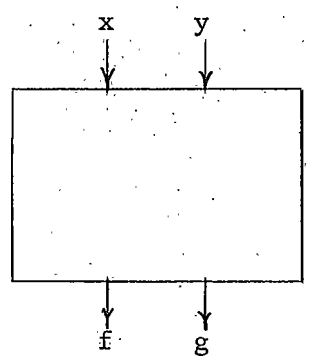
$$\text{CONVERT}(g) = x'y \times xy' \times + = g_1$$

$$\text{PLUSE}(g_1) = xy\oplus = g_2$$

This is also an optimal solution in that it takes only one cell. Figure 33 is the style 1 array for the half adder. The procedure CR would have yielded similar results for g. Thus, even in simple cases the user has a choice to make as to which procedures to apply.

x	y	f	g
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a)



(b)

Figure 32. The Half Adder

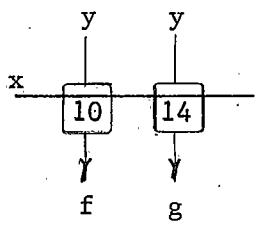


Figure 33. A Cellular Half Adder

Example 2: The Full Adder. Figure 34 illustrates the full adder and its truth table. From the truth table, the equations might be written as:

$$s = z'x'y+z'xy'+zx'y'+zxy$$

$$c = z'xy+zx'y+zxy'+zxy.$$

Select c and convert it.

CONVERT(c) = z'x×y×zx'×y×+zx×y'×+zx×y×+ = c_1

The algorithm produces a rather confused form with operators appearing as early as possible (as near the head as possible). Simplification results when they appear as late as possible. Thus, it is a good rule to apply MOVEOP early in the procedures.

MOVEOP(c_1) = z'xy××zx'y××zxy'××zxy××+++ = c_2

This results in a four cascade realization, whereas c_1 required eight. A substantial reduction in columns may often be gained

through CONC. Applying CONC to c_2 yields:

$$CONC(c_2) = xy\times zy\times zx\times ++ = c_3.$$

One might ask if the string is cascade realizable. Application of CR implies the use of one of the tests for cascade realizability. The author prefers the right synthesis test (see Section II.1.5). In this case, the result is:

$$CR(3) = False.$$

Thus, the best that can be hoped for is a two-cascade realization. Since c_3 is a disjunctive normal form, ORDER can be used to order completely the variables.

$$ORDER(z,x,y,c_3) = zx\times zy\times xy\times ++ = c_4$$

This particular order produces only a change in the order of occurrence of level-one terms and is as good as any other in this case. In more complex problems, the user may want to inspect several orders.

Next ELRULE(3) can be applied to factor out z.

$$ELRULE(3,c_4) = zxy+\times xy\times + = c_5$$

This produces a two-column realization. Alternately FACTOR could have been applied to c_4 for any variable. For instance.

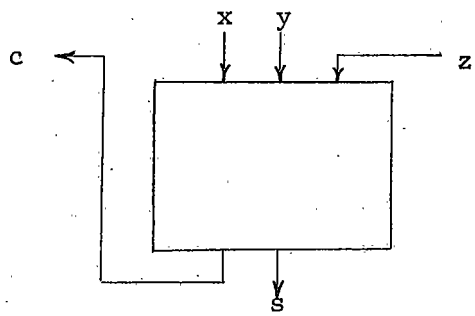
$$FACTOR(z,c_4) = z\ xy+\ \times$$
$$xy\times\quad + = c_6$$

After application of POSITION and EXCH for the two level-one substrings, c_7 can be seen to be the same as c_5 with the order of the level-one terms reversed.

$$xy\times zxy+\times + = c_7$$

z	x	y	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a)



(b)

Figure 34. The Full Adder.

Figure 35 shows style 1 arrays for c_5 and c_7, and points up a saving in cells which the user can effect. The embedding algorithm produces the array for c_7 shown in Figure 35(a). However, Figure 34(c) is a valid array for c_7 found by noting that the collection can be done in row 2 since cell(2,1) has no logical function in the first realization of c_7. This occurs when the two elementary strings have a different variable at their heads. This same saving is effected between Figures 35(b) and (d), where (b) is produced by the algorithm. In addition, Figures 35(c) and 35(d) point out another saving sometimes possible. In all four

realizations, the variable in row one is the same for both cascades. The variable in row two is used by only one cascade. If this variable is used in the right cascade then the cell in row two in the left cascade can be used to collect, saving an additional row. Thus if two adjacent elementary strings can be commuted, the one on the right should contain the variable (if any) at its head that is not used in the other. The user may use EXCH to transform c_7 into c_5. This appears to be an optimal realization for c_o since all cells in Figure 35(c) are used logically and there does not appear to be any way to write the function with fewer operators.

If s is taken from the truth table, it might be written as

$$s = z'x'y+z'xy'+zx'y'+zxy.$$

The CONVERT and MOVEOP and CONC procedures may then be applied.

$$\text{CONVERT}(s) = s_1$$

$$\text{MOVEOP}(s_1) = z'x'y\times\times z'xy'\times\times zx'y'\times\times zxy\times\times+++ = s_2$$

CONC produces no change. Thus the minterms are the prime implicants. One must then look for things to factor, try Θ reduction, or do a CR test. PLUSE will produce no change. FACTOR will produce some interesting results.

$$\text{FACTOR}(z,s_2) = z\ x'y'\times xy\times+\ \times$$

$$z'\ x'y\times xy'\times+\ \times\quad +\ = s_3$$

Now PLUSE will yield a reduction

$$\text{PLUSE}(s_3) = z\ \ xy\oplus' \times$$

$$z'\ xy\oplus\ \times\ + = s_4$$

The four-column realization is reduced to a two-column realization in s_4 and s_5.

$$\text{POSITION}(0, s_4) = zxy'\oplus \times z'xy\oplus \times + = s_5$$

One can note that $xy'\oplus$ is the complement of $xy\oplus$ or apply COMP to the two substrings.

$$\text{COMP}(xy'\oplus,\ xy\oplus) = \text{True}$$

This shows that the realization of s_5 is of the form $wX \times w'X' \times +$ which reduces by PLUSE

$$\text{PLUSE}(s_5) = zxy'\oplus'\oplus = s_6$$

The embedding algorithm will automatically remove the second prime, which in turn eliminates the first one. Thus s_6 has the same realization as s_7.

$$s_7 = zxy\oplus\oplus$$

The embedding algorithm states that the second function will be synthesized in the column to the right of the first function. Starting with Figure 35(c), a new column is added first to a style 2 array. The algorithm then produces the style 1 array of Figure 36. The embedding algorithm will find no conlficts in the buss line so no extra rows are added. This is a happy result obtained because z was

factored out of s_1. Any other variable could have been. The array would then have had more rows unless the operands had been commuted during a later stage using EXCH, COMMUTE, or SUB.

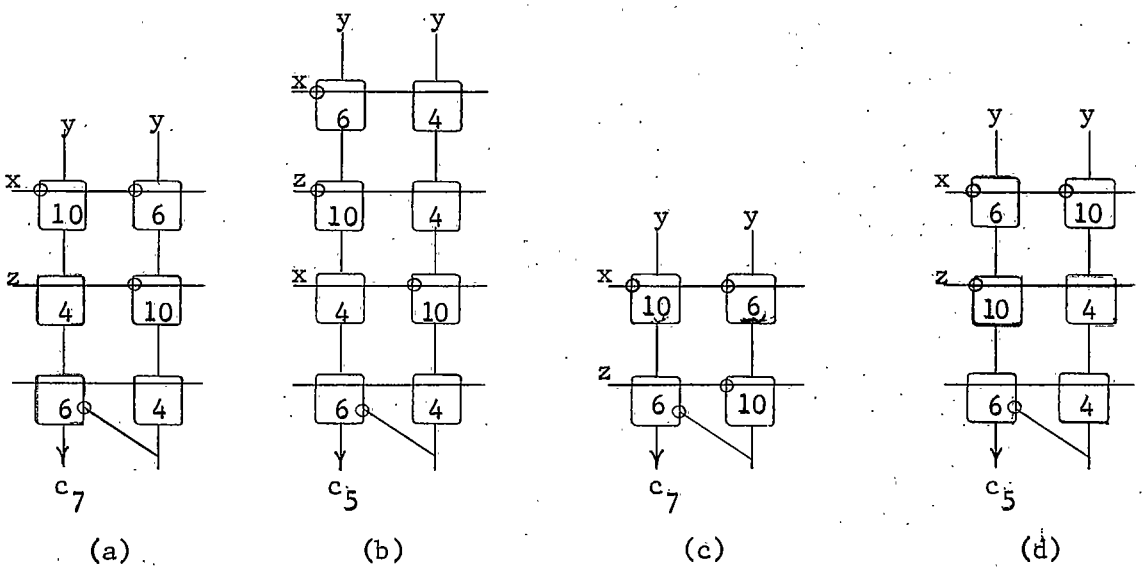


Figure 35. Style 1 Realizations of the Full Adder Carry Function

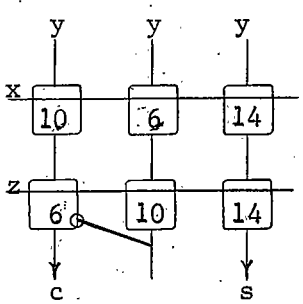


Figure 36. Style 1 Realization of the Full Adder

Example 3: A Double Adder. The double adder has two output digits, s and t, and a carry output, o. It has five inputs. These are u, v, w, x and c. Let u and w be the bits for the low order addition, v and x be the bits for the high order addition and c be the carry-in bit from a lower order stage. Then Figure 37 shows a truth table and block diagram for a double adder.

The function s will be treated first. Manipulations will not be discussed below where the reasoning is similar to that presented before.

CONVERT(s) = s_1

MOVEOP(s_1) = c'u'v'wx'××××c'u'v'wx××
c'u'vwx'××××c'u'vwx××××c'uv'w'x'××××
c'uv'w'x××××c'uvwx'××××c'uvw'x××××
cu'v'w'x'××××cu'v'w'x××××cu'vw'x'××××
cu'vw'x××××cuv'wx'××××cuv'wx××××
cuvwx'××××cuvwx××××+++++++++++++++ = s_2

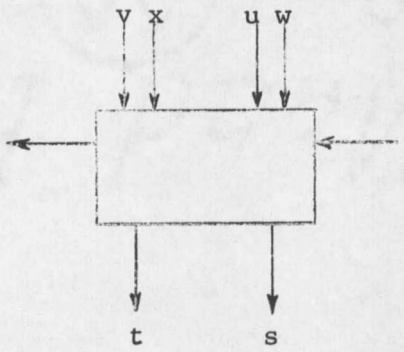
Many choices are available here. The most obvious would be to use CONC, CR or FACTOR. Certainly the answer is known to be a parity function, so CR would yield the result. It is illustrative though to go through a longer procedure. Concensus gives a considerable savings, but not a minimal form.

CONC(s_2) = c'u'w××c'uw'××cu'w'××cuw××+++ = s_3

FACTOR(c,s_3) = c u'w×uw×+ ×
c' uw'×u'w×+ × + = s_4

Inputs					Outputs		
c	u	v	w	x	s	t	o
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	0	0	1	0	1	0	1
0	0	0	1	1	1	1	0
0	0	1	0	0	0	1	0
0	0	1	0	1	0	0	1
0	0	1	1	0	1	1	0
0	0	1	1	1	1	0	1
0	1	0	0	0	1	0	0
0	1	0	0	1	1	1	0
0	1	0	1	0	0	1	0
0	1	0	1	1	0	0	0
0	1	1	0	0	1	1	0
0	1	1	0	1	1	0	1
0	1	1	1	0	0	0	1
0	1	1	1	1	0	1	1
1	0	0	0	0	1	0	0
1	0	0	0	1	1	1	0
1	0	0	1	0	0	0	0
1	0	0	1	1	0	1	0
1	0	1	0	0	1	1	0
1	0	1	0	1	1	0	1
1	0	1	1	0	0	1	0
1	0	1	1	1	0	0	1
1	1	0	0	0	0	1	0
1	1	0	0	1	0	0	0
1	1	0	1	0	1	1	0
1	1	0	1	1	1	0	0
1	1	1	0	0	0	0	1
1	1	1	0	1	0	1	1
1	1	1	1	0	1	0	1
1	1	1	1	1	1	1	1

(a)



(b)

Figure 37. The Double Adder

The form of s_4 is the same as that of s_3 in example 2. Thus the same manipulation could be used. To conveniently get the result CR is applied.

$$CR(s_4) = cuw'\oplus'\oplus = s_5$$

The manipulations have produced a simple representation of s. Figure 38 is the array.

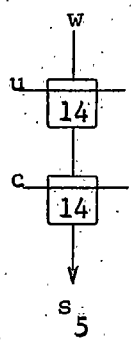


Figure 38. Cellular Array for the Low Order Sum of the Double Adder

Next consider the high order output, designated as t.

$$CONVERT(t) = t_1$$

$$MOVEOP(t_1) = c'u'v'w'x\times\times\times\times c'u'v'wx\times\times\times\times$$
$$c'u'vw'x'\times\times\times\times c'u'vwx'\times\times\times\times c'uv'w'x\times\times\times\times$$
$$c'uv'wx'\times\times\times\times c'uvw'x'\times\times\times\times c'uvwx\times\times\times\times$$
$$cu'v'w'x\times\times\times\times cu'v'wx\times\times\times\times cu'vw'x'\times\times\times\times$$
$$cu'vwx'\times\times\times\times cuv'w'x'\times\times\times\times cuv'wx'\times\times\times\times$$
$$cuvw'x\times\times\times\times cuvwx\times\times\times\times + + + + + + + + + + + + + + + = t_2$$

$$CONC(t_2) = t_3$$

Previous functions were parity functions or simple in nature so as to have an obvious resultant form. This one is not known to be a parity function or simple. There is a high probability that not all the prime implicants are essential. In this case, INF can be used to eliminate some non-essential prime implicants, yielding an irredundant normal form.

$$INF(t_3) = c'v'w'x\times\times\times c'vw'x'\times\times\times uv'wx'\times\times\times$$
$$uvwx\times\times\times cuv'x'\times\times\times cuvx\times\times\times u'v'\times\times\times$$
$$u'vx'\times\times+++++++ = t_4$$

Variable v appears in each term. By this virtue, it is a good candidate for FACTOR

$$FACTOR(v,t_4) = v\ c'w'x'\times\times uwx\times\times cvx\times\times u'x'\times+++\ \times$$
$$v'\ c'w'x\times\times uwx'\times\times cvx'\times\times u'x\times+++\ \times\quad +\ = t_5$$

Can the two substrings multiplying v and v' be simplified? Factoring out variables lends insight. One may ask if the two substrings are equal or complimentary. For simplification define:

$$t_5 = vR\times$$
$$v'S\times\ t.$$

Then test S and R

$$EQUAL(S,R) = False$$
$$COMP(S,R) = True$$

Since S = R', substitute vSΘ for the entire function. It is written below using the definition for S.

$$SUB(0,all,vS\Theta,t_5) = vc'w'x\times\times uwx''\times\times cux'\times\times u'x+++\Theta = t_6$$

Can S itself be simplified? Again, x appears in all terms. Factor it out.

$$FACTOR(x,S) = x\ c'w'\times u'+\ \times$$
$$x'\ uw\times cu\times+\ \times\quad + = S_1$$

Again, the substrings multiplying x and x' can be tested for equality and complimentation. The result is that they are complimentary. Performing the substitution again t can be written as:

$$t_6 = vxuw\times cu\times+\Theta\Theta.$$

An application of ELRULE completes the reduction.

$$ELRULE(3,t_6) = vxuwc+\ \Theta\Theta = t_7$$

Of course, CR would have given the result immediately. However, this example has illustrated the use of INF, COMP and EQUAL, which would be useful even if the function were not cascade realizable.

The third output is o. This is the carry out function.

$$CONVERT(o) = o_1$$

$$MOVEOP(o_1) = c'u'vw'x\times\times\times c'u'vwx\times\times\times c'uvw'x\times\times\times$$
$$c'uvwx'\times\times\times c'uvwx\times\times\times cu'vw'x\times\times\times+++++ = o_2$$

$$CONC(o_0) = o_3$$

$$INF(o_3) = uvw\times\times cuv\times\times vx\times ++ = o_4$$

$$FACTOR(v,o_4) = v\ uw\times cu\times x++\ \times = o_5$$

$$ORDER(v,x,u,w,o_5) = vxuw\times uc\times ++\times = o_6$$

Now u can be factored out of the inner two terms. Define a new argument for factor and substitute the result back into o_6.

$$FACTOR(u,uw\times uc\times +) = u\ wc+\ \times$$

$$SUB(0,uw\times uc\times,+,uwc+\times,o_6) = vxuwc+\times+\times = o_7$$

Thus the carry out function is also cascade realizable. Again CR could have been applied anywhere along the way. The method used illustrates the use of SUB.

A problem which confronts the designer at this stage is that of a different ordering for each of the functions to be realized in a single array. The three functions,

$$o_7 = vxuwc+\times+\times,$$

$$t_5 = vxucw\oplus\oplus\times+$$

$$s_5 = cuw'\oplus'\oplus,$$

do not have the same ordering. In this case t and s can be ordered to conform to the ordering in o. In the general case this cannot be done. The designer must determine a trade-off. He must live with the extra rows caused by the different orderings or he must introduce extra columns. By adding columns to the realization of one or more functions, a point will be reached where each column will fit into one ordering.

Using the ordering of function o_7, the functions are manipulated as below.

$$\text{ORDER}(v,x,u,w,c,s_5) = uw'c\oplus'\oplus = s_6$$

$$\text{ORDER}(v,x,u,w,c,t_5) = vxuwc\oplus\oplus\times+ = t_6$$

The realization implied by these equations and produced by the embedding algorithm as shown in Figure 39.

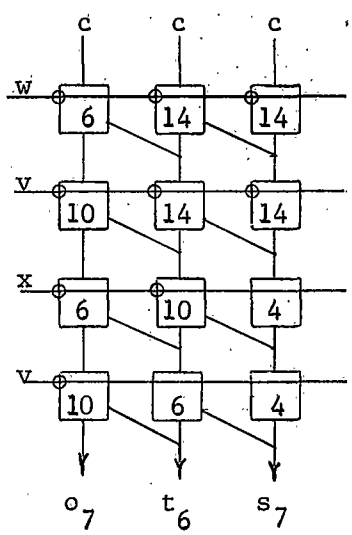


Figure 39. Cellular Realization of the Double Adder

Example 4. The following example was suggested by Dr. Kenneth J. Thurber*. It has six variables and a minimal realization is not known. The function was given to the author in the following form:

$$f = xyz'w + xyz'uv' + x'yz'w(u \oplus v') + zxw'(u' \oplus v) +$$
$$zyw'(u' \oplus v) + zx'(u \oplus v + w) + zy'(u \oplus v + w) +$$
$$z(y \oplus x) + xyz'u'v$$

After application of CONVERT, NOPLUSE and MOVEOP in that order, the function appears as follows.

f_1 = xyz'w×××xyz'uv'××××x'yz'wuv'×u'v×+××××
zxw'u'v'×uv×+×××zyw'u'v'×uv×+×××zx'uv'×u'v×w++××
zy'uv'×u'v×w++××zyx'×y'x×+×xyz'u'v××××+++++++

This is a fairly complex form with many characters, more than the eye can comprehend at once (for most people). Application of CR yields a negative result. If CONC is applied the following results.

CONC(f_1) = wxyz'×××uv'xyz'××××uvw'xz××××u'v'w'xz××××uvw'yz××××
u'v'w'yz××××uv'x'z×××u'vx'z×××wx'z××uv'y'z×××
u'vy'z×××wy'z××x'yz××xy'z××u'vxyz'××××uvwyz'××××
u'v'wyz'××××uvwx'y××××u'v'wx'y××××++++++++++++++++++ = f_2

This has nineteen elementary P-strings, more than f_1. However, not all of the prime implicants are needed. Applying INF will yield over 100

---

*Private Communication

irredundant normal forms, all having around 12 terms. One of these must be selected. The particular one selected appears to have some influence on the number of cells needed to realize the function, but not a radical one.* One irredundant normal form has been chosen for study here.

$$\begin{aligned} INF(f_2) = {} & wxyz'\times\times\times uvw'xz\times\times\times\times u'v'w'xz\times\times\times\times u'vx'z\times\times\times \\ & wx'z\times\times uv'y'z\times\times\times x'yy\times\times xy'z\times\times u'v'wyz'\times\times\times\times \\ & uv'xyz'\times\times\times\times u'vxyz'\times\times\times\times\!+\!+\!+\!+\!+\!+\!+\!+\!+ = f_3 \end{aligned}$$

Now since the CR test showed the function to require more than one column, two avenues may be pursued. First, the creation of $\oplus$ operators may be investigated. Second, variables may be factored out of the P-string, hoping that the resulting substrings will simplify. (An alternate to this step is to do a Shannon or Reed decomposition). Three sets of terms have structure that allow a $\oplus$ to be formed by PLUSET.

$$\begin{aligned} PLUSET(f_3) = {} & wxyz'\times\times\times wx'yuv'\oplus\times\times\times u'vx'z\times\times\times wx'y\times\times \\ & uv'y'z\times\times\times zx'y'\oplus\times u'v'wyz'\times\times\times\times \\ & xyz'uv\oplus\times\times\times\!+\!+\!+\!+\!+\!+\!+ = f_4 \end{aligned}$$

---

*One can see here the necessity of computerizing this method. The manipulations become involved with five or more variables, and the computer can quickly do the routine work such as the CONC and INF algorithms, allowing trial of several different irredundant normal forms.

Now u and v cannot be moved to the head of all substrings without producing extra rows. It may be helpful to order all substrings with u and v at the tail.

$$\text{COMMUTE}(u,T,f_4) = f_5$$

$$\text{COMMUTE}(v,T,f_5) = \text{'wxyz'}\times\times\times\text{w'xyuv}\oplus\times\times\times\text{x'zuv}\times\times\times$$
$$\text{wx'y}\times\times\text{y'zuv'}\times\times\times\text{zx'y'}\oplus\times\text{wyz'u'v'}\times\times\times\times\text{xyz'uv}\oplus$$
$$\times\times\times+++++++ = f_6$$

Note, y appears in most of the elementary P-strings. As before this may be a good variable to factor out.

$$\text{FACTOR}(y,f_6) = y\ \text{wxz'}\times\times\text{w'xuv}\oplus\times\times\text{wx'}\times\text{wz'u'v'}\times\times\times\text{xz'uv}\oplus\times\times\times+++ \times$$
$$y'\ \text{zuv'}\times\times \times$$
$$\text{x'zuv}\times\times\times\text{zx'y'}\oplus\times+ \quad ++ = f_7$$

The last substring contains a y which will not factor out. Now y appears at the tail and head of substrings. This will require a redundant entering of y. As an alternative, z is factored out of f_6.

$$\text{FACTOR}(z,f_6) = z\ \text{x'uv}\times\times\text{y'uv'}\times\times\text{x'y'}\oplus++ \times$$
$$z'\ \text{wxy}\times\times\text{wyu'v'}\times\times\times\text{xyuv}\oplus\times\times++ \times$$
$$\text{w'xyuv}\oplus\times\times\times\text{wx'y}\times\times+ \quad ++ = f_8$$

No simplifications are apparent here either. As a last alternative, w is factored out of f_6.

$$\text{FACTOR}(w,f_6) = w\ \text{xyz'}\times\times\text{x'y}\times\text{yz'u'v'}\times\times\times++ \times$$
$$w'\ \text{xyuv}\oplus\times\times \times$$
$$\text{x'zuv}\times\times\times\text{y'zuv'}\times\times\times\text{zx'y'}\oplus\times\text{xyz'uv}\oplus\times\times\times+++ \quad ++ = f_9$$

Now some simplifications can be made. First line one of f_9 is treated.

FACTOR(y,line 1 of f_9) = yw xz'××'z'u'v'××++ ×× = f_{10}

The counter absorption rule can be applied successfully here. Afterward an absorption or subsuming process will simplify the string.

ELRULE(8,f_{10}) = ywz'x'z'u'v'××++×× = f_{11}

SUBSUME(f_{11}) = ywz'x'+×× = f_{12}

SUB(0,line 1,f_{13},f_9) = ywz'x'+××

w' xyuv⊕×× ×

x'zuv×××y'zuv'×××zx'y'⊕×xyz'uv⊕×××+++ ++ = f_{13}

Line two of f_{13} is already a single string. Factoring variables out of line three does not reduce the column count. Likewise, if the ⊕'s are eliminated and u or v is factored, no improvement is gained. If one considered line three as the start of a new problem, CONC might be applied.

CONC(line 3 of f_{13}) = x'yz××xy'z××uv'xyz'××××u'vxyz'××××ux'z××

uy'z××+++++ = f_{14}.

An application of INF will show that only one irredundant normal form exits, consisting of all the prime implicants. PLUSET will produce two ⊕ gates.

PLUSET(f_{14}) = zx'y'⊕ xyz'uv⊕×××ux'z××uy'z××+++ = f_{15}.

Thus the column count is reduced to four in f_{15}. More can still be done.

$$\text{FACTOR}(z,f_{15}) = z\ x'y'\oplus ux'\times uy'\times ++\ \times$$
$$z'\ xyuv\oplus\times\times\ \times\quad +\ = f_{16}$$

$$\text{ELRULE}(3,f_{16}) = z\ x'y'\oplus ux'y'+\times +\ \times$$
$$z'\ xyuv\oplus\times\times\ \times\quad +\ = f_{17}$$

Line one of f_{17} has redundant entering of x and y. This is a candidate for reduction.

$$\text{CONC}(x'y'\oplus ux'y'+\times +) = x'y\times xy'\times ux'\times uy'\times +++ = f_{18}$$

$$\text{INF}(f_{19}) = ux'\times xy'\times x'y\times ++ = f_{19}$$

$$\text{PLUSE}(f_{20}) = ux'\times xy\oplus + = f_{20}$$

Substitute this back into f_{17}.

$$\text{SUB}(0,x'y'\oplus ux'y'+\times +,f_{20},f_{17}) = z\ ux'\times xy\oplus +\ \times$$
$$z'\ xyuv\oplus\times\times\ \times\quad +\ = f_{21}$$

Thus one more column has been eliminated from f_{15}. Substituting f_{17} back into the original function,

$$\text{SUB}(0,\text{line 3 of},f_{21},f_{13}) = ywz'x'+\times\times$$
$$w'\ xyuv\oplus\times\times\ \times$$
$$z\ ux'\times x'y'\oplus +\ \times$$
$$z'\ xyuv\oplus\times\times\ \times\quad +\quad ++\ = f_{22}.$$

As discussed before the variables should be ordered when possible to reduce the number of rows in the realization. For each P-string in f_{22}, a relationship can be defined. In this relationship $x < y$ if and only if x must appear nearer the head of the P-string than y. (See Section III.1.2) These are shown in Figure 40.

No.	Relationship	String	Arbitrarily Completed Relationships
1	y,w<z,x	ywz'x'+××	y<w<z<x
2	w,x,y<u,v	w'xyuvΘ×××	y<w<x<u<v
3	None	ux'×	u<v
4	None	xyΘ	y<x
5 and 6	z<u,x,y and u,x<x,y or x,y<u,x	zux'×xyΘ+×	z<y<x<u y<x<u<x
7	z<x,y<u,v	z'xyuvΘ×××	z<y<x<u<v
	(a)	(b)	(c)

Figure 40. Relationships for Variables in Synthesis Example

There are conflicting elements in the relationships expressed in Figure 40. For example, the first and fourth rows yield $y < z$ and $z < y$ respectively. One must choose an ordering to minimize such conflicts. This gives a corresponding savings in rows. Figure 40(c) shows a complete ordering which has several conflicts. (The ordering must be completed to define the array.) The sixth row conflicts with itself and others. This indicates that x will be entered into the array more than once. However, in this case, the second entry is on the extreme right which may be entered into the top of the array, thus not causing an additional row. The z in row 1 conflicts with other z's. A judgement must be made for the z conflict. Both orders could be tried. The least number of conflicts remain with the following ordering.

$$\text{POSITION}(0,f_{22}) = f_{23}$$

$$\text{ORDER}(y,w,z,x,u,v,f_{23}) = ywz'x'+\times\times yw'xuv\theta\times\times\times zyx\theta x'u\times+\times yz'xuv\theta$$
$$\times\times\times+++ = f_{24}$$

Figure 41(a) shows the style 2 array for f_{24} generated by the embedding algorithm. Figure 41(b) shows the resultant style 1 array from the embedding algorithm. Figure 41(c) shows a reduced array created by taking the array in (b) and sliding cells in columns up and down to achieve a better match between the variables on the horizontal inputs of a given row. Also the two inputs of cell(1,1)

have been interchanged. This is a place where the designer must use his intuition and a heuristic approach. No algorithm is known for performing this reduction, other than an exhaustive search. Such a search would have an exceedingly large number of branches. It took the author less than five minutes to effect the reduction, using his intuition. The per cent of cells used logically is 60. This seems reasonable when compared to other functions realized in related arrays. For example, Minnick has realized a cellular multiplier in cutpoint arrays, using 71 per cent of the cells logically, and in a cobweb array, using 55 per cent of the cells logically[18].

III.3.3 P-strings for Register Logic

Register logic may be logically embedded in a style 1 array using P-strings. There are two determining factors: (1) flip-flop cells, and (2) outputs from flip-flops used as inputs. Assume that the output from the flip-flop appears at the array edge. Then it is available to be put on the horizontal buss lines or into the top of the array as any variable is. Thus it can appear as an input where needed. P-string equations are written and manipulated as before, treating the flip-flop output as a variable. Such variables will be labeled as internal variables in the sequel. With the P-equations for the set and reset of a flip-flop in the desired form,

first realize the reset function in the array, then realize the set equation, using the embedding algorithm. This will cause the set function to appear to the right of the reset function. Locate the flip-flop cell in the column below the realization of the reset equation, in the first cell in that column into which the set function can be bussed using the diagonal input. Finally use a jumper wire to connect the output of the flip-flop to the horizontal busses of the rows where it is used as a variable.

Example 5: A Register Function. The function realized in this example is the general register cell used by Cannon in his parallel processing machine[3]. The functions, as is often true in register functions, are not complex. The two functions are R and S, the reset and set functions for a flip-flop. They are defined as:

$$R = cr + tY + id'$$

and

$$S = cs + tY + id.$$

Figure 42 is a diagram of the cell. Y and Y' are the outputs from the flip-flop. Only one of c, t and i are energized at one time. The respective functions of the cell in these cases are those of R-S, Trigger and D flip-flops. The activated data input sets for the three cases are respectively: {r,s},{t} and {d}. Figure 43 is a style 1 array realization for the cell. The form of the equations realized is:

R = cr×tY×id'×++

S = cs×tY'×id×++.

These are determined by applying CONVERT and MOVEOP to the equations written above. No reductions can be made on them. The only option open to the user is to re-arrange the order of the variables. With this arrangement, r and s come in the top while c, used by two cascades, comes in on a horizontal buss. This saves a row over the realization with rc× and sc× as the first elementary P-strings. The order of the other variables makes little difference.

III.3.4 Summary

In this section was set forth a method of manipulating P-strings into desirable forms for embedding in style 1 cellular arrays. The approach is heuristic in nature. Algorithms for manipulations were developed, but the method of applying them was left to the intuition of the designer. Several examples were included to indicate the method of application used by the author. The excellence of a final design will depend upon the expertise of the designer.

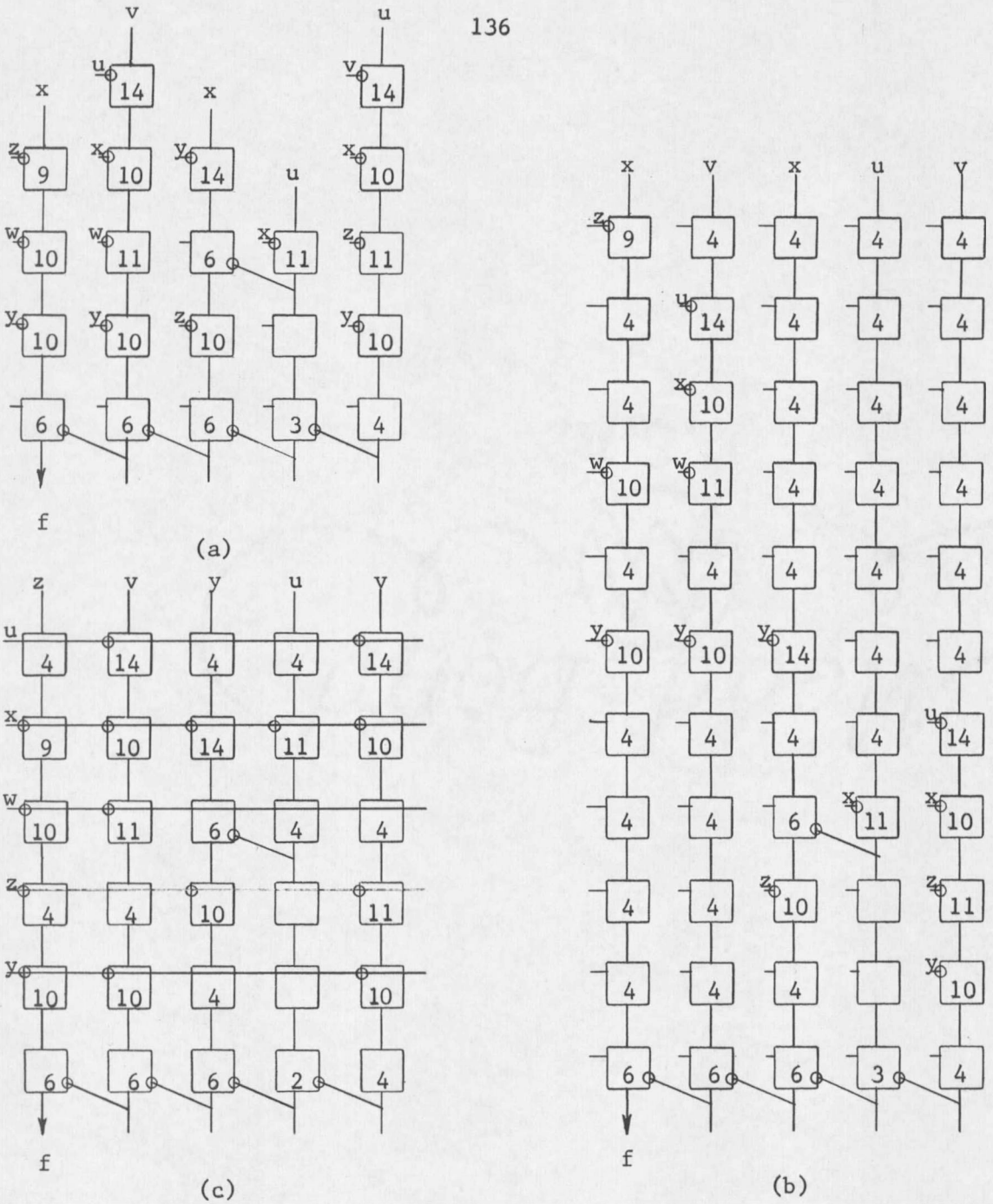


Figure 41. Realization for Example 4

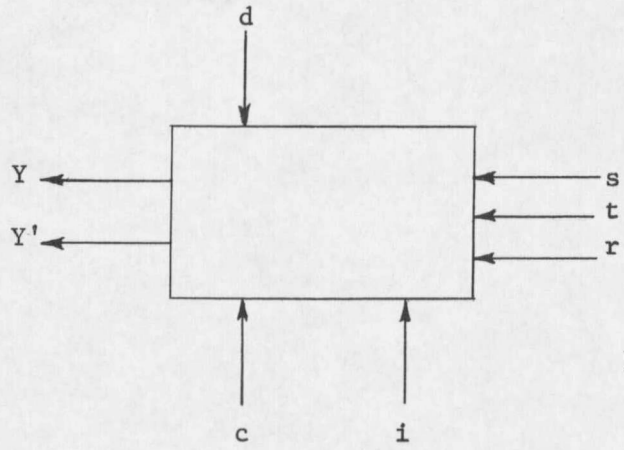


Figure 42. Cannon's General Register Cell

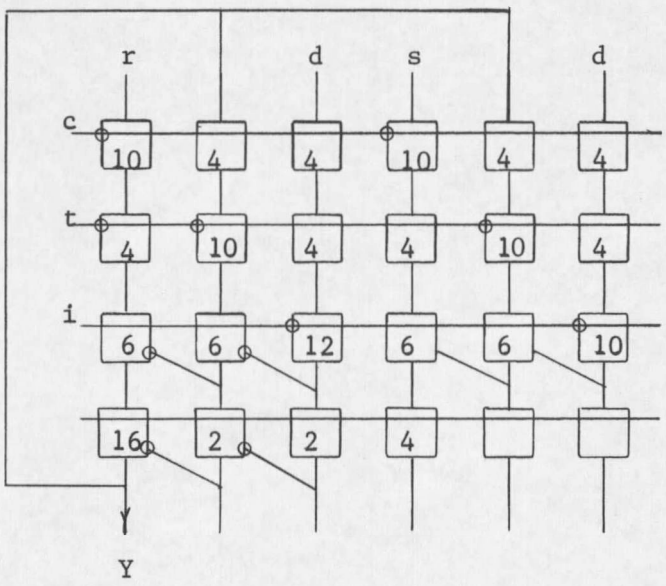


Figure 43. Style 1 Realization of Cannon's General Register Cell

III.4 Buss Synthesis

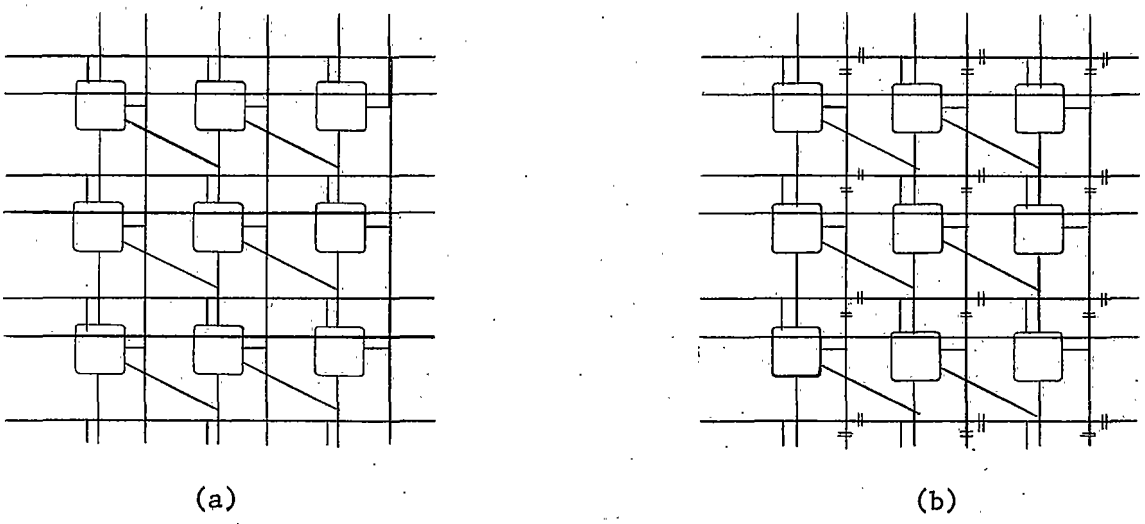
In this section, a new family of arrays is introduced. It is distinguished from previous arrays by a by a buss structure which can be generalized. A set of procedures is developed for embedding functions within a particular buss array called style 4. An algorithm is developed for using the procedures to embed functions, and a heuristic approach to using the procedures is discussed. An engineering example is produced and comparisons are made with realizations from the previous section.

III.4.1 Buss Arrays

Figure 44(a) shows one of the simplest forms of the buss array. It is a style 1 array with a horizontal and a vertical buss added between each row and column. The horizontal busses are connected as alternate y inputs to cells directly beneath them. Vertical busses are connected as alternative x inputs to the cells at their immediate left. It will be assumed that a cell contains the necessary logic to select one x and one y input. Henceforth, the original busses, vertical busses and new horizontal busses will be denoted respectively as Xbusses, Vbusses and Hbusses. Wherever an Hbuss crosses a Vbuss or a cell output, it is assumed that there is a joinpoint which can connect

the lines. When this connection is made, it will be specified in the figures by a small circle drawn over the crossing.

Figure 44(b) illustrates the style 4 array which has been chosen for use in this section. The short vertical and horizontal line pairs denote cutpoints in the busses. A closure of those cutpoints is indicated by a small circle enclosing the line pair.



(a) (b)

Figure 44. Two Buss Arrays.

Figure 45 is a more general array which indicates how more complicated arrays can be defined from the basic buss structure. There are four busses in a style 5 array for every one in a style 4 array. Additional cutpoints are placed where Vbusses cross Hbusses. One could

conceivably define any number of Vbusses, Hbusses and Xbusses. One could also select some or all of the buss and output intersections as joinpoints. The basic consideration must be cost versus efficiency of usage. Additional busses and joinpoints cost more, but the per cent of cells used logically rises also, allowing smaller arrays.

In the sequel the alternate x and y inputs will be shown only when used. Diagonal x inputs will also be shown only when used, and any x or y input used will be circled.

III.4.2 Comparison of Buss Arrays and Other Arrays.

The style 4 array is chosen in the usual engineering sense of compromise. The cost per cell is higher than that of a style 1 array but less than that of a style 5 array, or that of a full cobweb array. The measure of the cost suggested here is the number of bits it takes to specify a cell. The full cobweb array takes 12 bits[18]; the style 1 array requires five bits; the style 4 requires 11 bits; and the style 5 array requires 48. Style 4 arrays with cutpoint cells require 10 bits. Additionally, style 4 arrays alleviate problems encountered with style 1 arrays. Without the busses, there is no way to present information composed in one portion of the array to another without using cells for bussing. This is wasteful. Figure 46 is a realization of the function of Example 4 in a style 4 array. It requires 20 cells with 90 per cent

of the cells used logically. This compares to 30 cells required in the style 1 array with 60 per cent used logically. If the cost is considered as the number of specification bits times the number of cells, the cost of the style 1 realization is 150 and that of the style 4 is 220. In so far as the cost is realistic, this example would not justify the use of the more complex array. However, the cost criterion may not be accurate and indeed is not the only consideration. In the author's opinion, the cost of a cell is more complex than the sum of its cutpoints and joinpoints. If the weightings were shifted to weight the cost of a joinpoint or cutpoint bit by one fourth, the style 1 cost would be five per cell and the style 4 cost would be eight instead of respectively five and eleven as before. This may be a reasonable cost criterion, since if burn-out fuses are used to set joinpoints and cutpoints in the busses, the cost per bit should be less than the cost per bit of the switches to set the cells and certainly less than the cost of fabrication of the cells themselves. This new cost measure still yields a higher cost for the style 4 realization of the example. However, additional justification for the buss arrays is given below.

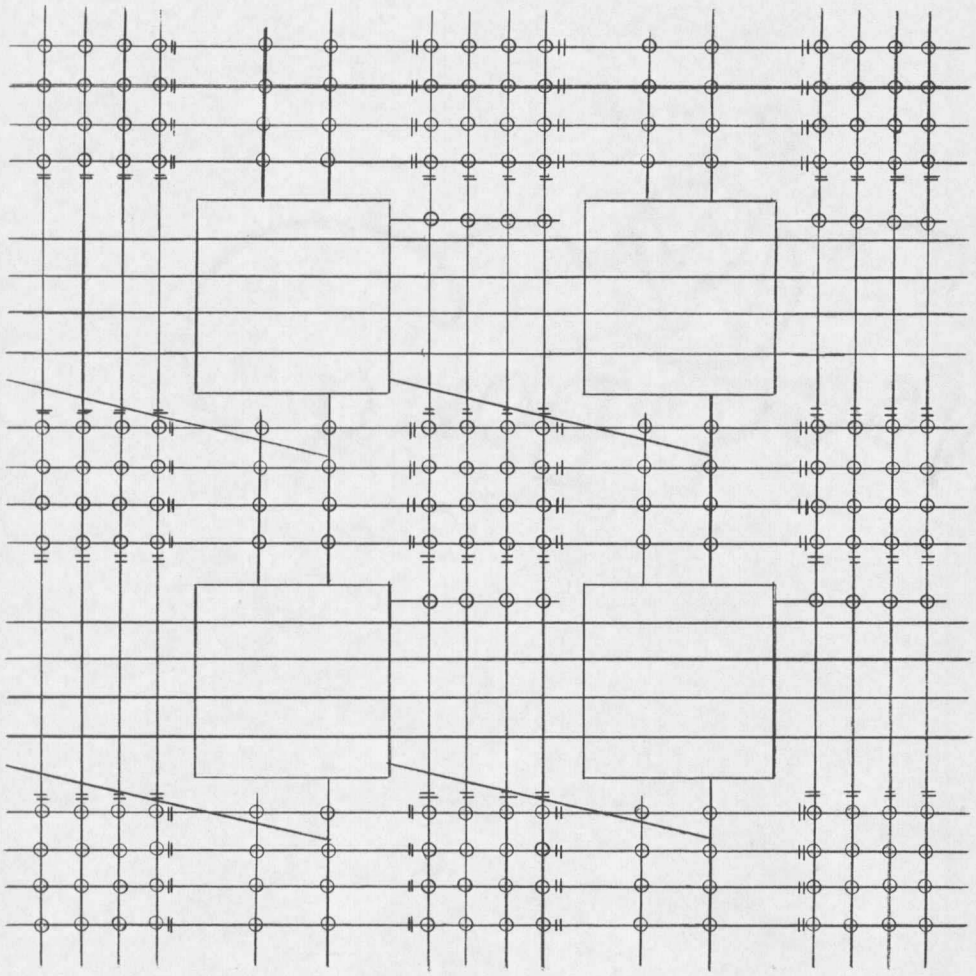


Figure 45. A Style 5 Buss Array.

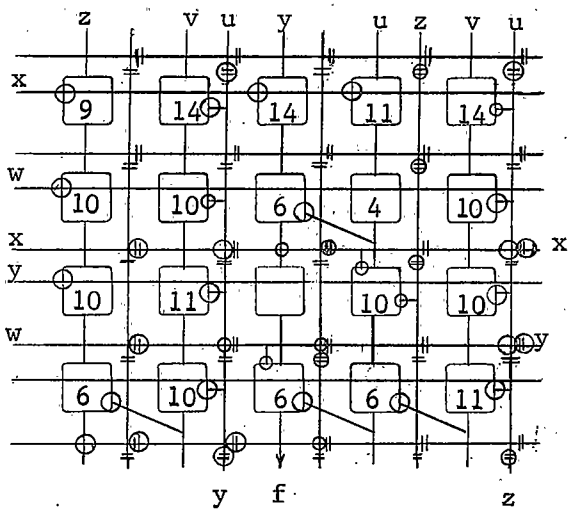


Figure 46. Style 4 Realization of Example 4.

III.4.3 Solution of Iterated Problems with Buss Arrays.

An important factor affecting the choice of arrays is the system within which the functions realized in an array are embedded. That is to say, it is one thing to realize a function and quite another to realize it within the constraints imposed by an overall system. For example, one may wish to realize a full adder, as was done in Section III.3.2. In addition, one may desire that this be one cell in an 18 bit adder. Such a system is pictured in Figure 47. When speaking of hundreds of cells on a chip, it is conceivable that more than one full adder (perhaps all) will be realized in one chip. This is the iterated approach to synthesis. A functional cell is designed and identical

ones are iterated across the array. In this case, the relative location of the full adders and the location relationship between outputs from one full adder and inputs to the next becomes important.

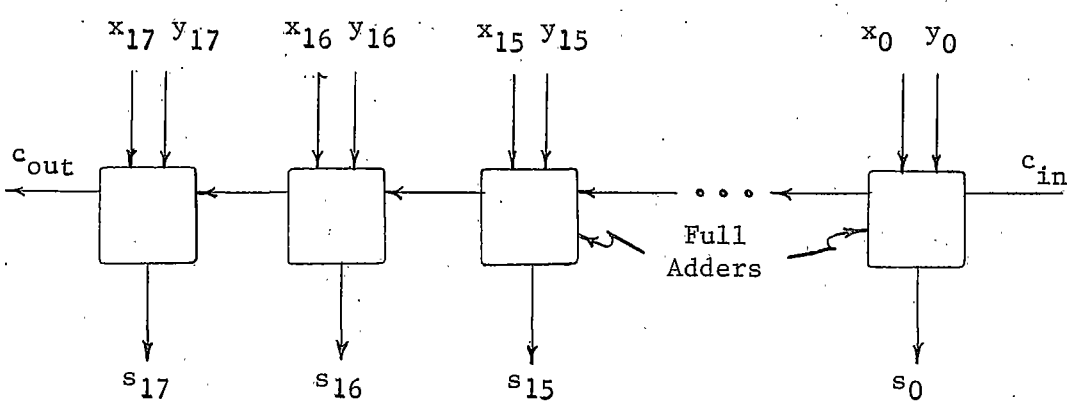


Figure 47. An 18 Bit Adder.

In particular, one can envision realizing the adders side by side in the array. In this case, the carry out from one full adder must connect with the carry in of the next higher one. If one is to avoid external wiring, with its associated wastage in space and low reliability factor, a method must be devised to buss the output from where

it is created to the point of input, usually higher in the array. With the style 1 array, this cannot be done. With cobweb and buss arrays, it can be done by routing the output of functions through buss segments to the input of the appropriate cell in the next stage.

A more complex problem is Cannon's Processor Cell[3], pictured in Figure 48. In Figure 48 various functional blocks are represented with letters. The logical interconnections between blocks are shown with lines. Thus if one assumes that the functional blocks are realized with the relative locations indicated, then some constraints are imposed on the location of outputs from and inputs to each block. Additionally, one must make provision for bussing some data through or around a block to the next one. For example, several lines pass through block R and are not connected to it.

The functional blocks are in turn made up of cells which are iterated. In particular, block M is a multiplier with parallel I/O transfer, made up of four registers and associated logical control. Figure 49 shows the internal structure of the block containing this multiplier. The cells which have a one and zero within them are general register cells. Such a cell was embedded in a style 1 array in

section III.3.1. The cells marked with a + are full adders. Such a cell was also synthesized in section III.3.1. The interconnection of these cells places constraints upon locations of inputs and outputs of the cells, similar to those described above.

With style 1 arrays there is no provision to interconnect the sub-arrays in the complex fashion indicated by this system. With buss arrays it is possible. A function realized at any location can be bussed, through use of Vbusses, Hbusses, cutpoints and joinpoints, to any location within the array (not already in use by another function or variable).

This is the principal advantage of the buss array: the capability of relocating data arbitrarily within the array. This property will, for many problems, override the comparative cost of buss arrays.

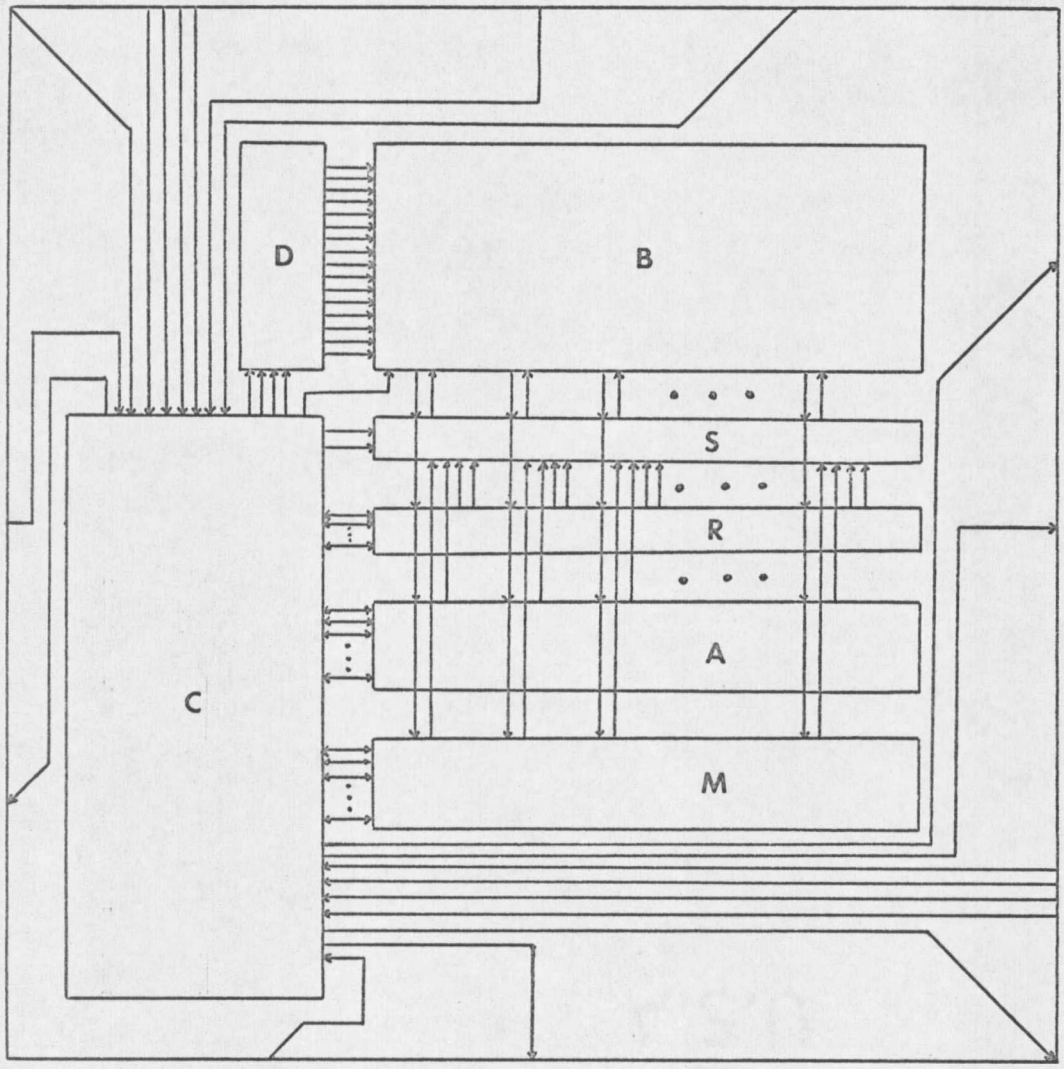


Figure 48. Cannon's Processor Cell

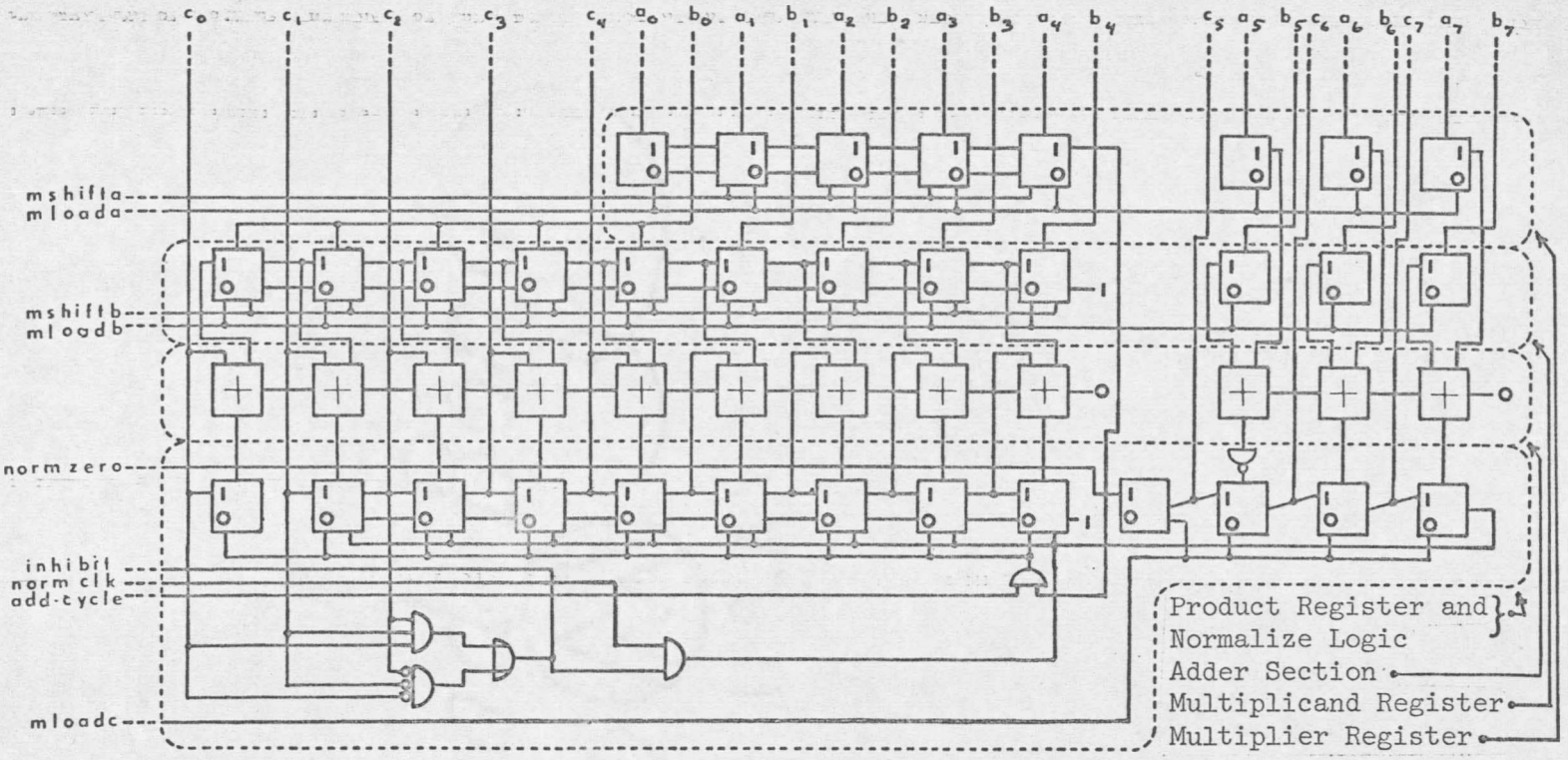


Figure 49. Cannon's Multiplier

III.4.4 Constraints Imposed by Iterated Problems.

The iterated approach to synthesis is to break the problem down into functional cells which are then joined together with identical or similar cells. This can be generalized somewhat. One can conceive of a complex structure broken down into dissimilar cells which are then joined together. The constraints imposed by either type of problem are the same. Cannon's processor cell illustrates both the original approach and the generalization. The embedding of the functional blocks is a generalized iteration while the blocks themselves are formed through the original more specific iteration.

Some of the constraint types imposed by iteration have already been mentioned. These are constraints: (1) on output and input locations, and (2) causing a datum to be bussed from one side of a cell to another. Additional constraints are listed along with these in Figure 50. The author has found these to be sufficient to allow synthesis of buss arrays for all problems investigated. However, they are not presented as a closed set. Any user may wish to add or subtract to them as he sees fit.

Class	Type	Options*	Parameter	Comment
RESERVE	CELL		i,j	i=row j=column
	VBUSS	{SEGMENTS / ALL}	n-m	n is leftmost or top segment to be reserved, m is the rightmost or bottom one
	HBUSS	{SEGMENTS / ALL}	n-m	
	XBUSS		i	i=row number
DEDICATE	COLUMN**		i,v	v=variable or internal function i=column number
	CELL		v	v=internal function identifying a flip-flop

Figure 50. Constraints for Buss Array Synthesis

*Choose one option from each set of brackets. The dash indicates a selection of "no option". Some combinations have no meaning. For example, an XBUSS has no SEGMENTS. Also, a VBUSS has no left or right.

**Column here refers the input to the top cascade cell which connects to the outside world.

Class	Type	Options*	Parameter	Comment
DEDICATE (Cont.)	VBUSS		i,n-m,v	i=BUSS number† n-m segments n through m. v= variable or interval function.
	HBUSS		i,n-m,v	Same as VBUSS
	XBUSS		i,v	Same as VBUSS
TOPOLOGY	CELLS	{ MAXIMUM EXACT }	n	n=number specified
	ROWS	{ MAXIMUM EXACT }	n	
	COLUMN	{ MAXIMUM EXACT }	n	
DEMAND	FUNCTION OUTPUT	{ VBUSS HBUSS COLUMN } { TOP BOTTOM RIGHT LEFT }	i	i=buss or column number

Figure 50. Continued

†Busses are numbered top to bottom and left to right.

Class	Type	Options*	Parameter	Comment
DEMAND	VARIABLE OR INTERNAL FUNCTION OR INTERMEDIATE FUNCTION	{INPUT, OUTPUT, -} {ONLY, OTHERS, -}	v_1,v_2,i,n-m	i=buss or column number n-m=first and last segments used
		{VBUSS, HBUSS, COLUMN, XBUSS, -} {TOP, BOTTOM, RIGHT, LEFT, SEGMENTS, -}††		
	BELOW		x,y	y must appear as an output directly opposite (below or beside) of anywhere that x is an input. x and y can be variables or functions or internal functions
	BESIDE		x,y	

Figure 50. Continued

††ONLY is implied here.

The RESERVE class of constraints is used when it is desirable to reserve a cell or buss segment for use at some other task. The class would more properly be labeled "declare as unavailable for this problem", but such a title is unwieldy. Reservations can be used to avoid faulty cells and busses. They can be used if a particular cell or buss is being saved for use by a neighboring functional block. For Vbusses and Hbusses one has the option of reserving segments of a buss or all of it.

A second type of constraint is the DEDICATE constraint. To dedicate a buss or column to a variable or internal function is to reserve that buss or column for use by that variable or internal function. Use of this is varied and may overlap that of the DEMAND constraints. For example, one may desire a variable to be bussed through an array being designed, say from top to bottom. One then may dedicate an appropriate Vbuss to this variable. Note that this specifies the entire path of the variable from input to output. (The DEMAND constraint can be used for the bussing function, but normally would specify only the end points, not the path.) Cells may be dedicated to flip-flops. Such a cell is set to function 16.

TOPOLOGY constraints have to do with the size of the array. One can specify the maximum or exact size of the array and its shape.

DEMAND constraints are perhaps the most used in iteration problems. When one wishes to juxtapose two or more functional arrays, as suggested before, outputs from one must match inputs to the other. If the two arrays perform identical functions such as a register of full adders then it is sufficient to require that the output of one functional cell (full adder) appear directly opposite the corresponding input of the next. For example, if the carry out of a full adder is from column three and the arrays are to be placed one below the other as in Figure 51, it is sufficient to require that the carry in input be made only to column three, that the inputs come in on the left side, and that the output emerge on the right side. This can be assured by specifying the five demands:

DEMAND c_o OUTPUT ONLY COLUMN 3

DEMAND c_i INPUT ONLY COLUMN 3

DEMAND x INPUT ONLY LEFT

DEMAND y INPUT ONLY LEFT

DEMAND s OUTPUT ONLY RIGHT

This approach is also useful if one is trying to match outputs and inputs with a dissimilar array.

A more general approach for iterating identical arrays is use of the BELOW and BESIDE constraints. It is sufficient in the example of

Figure 51 to require that the carry out appear below the carry in. One need not specify the actual column or Vbuss. The demand replacing the two demands on c_o and c_i might take the form:

DEMAND c_o BELOW c_i.

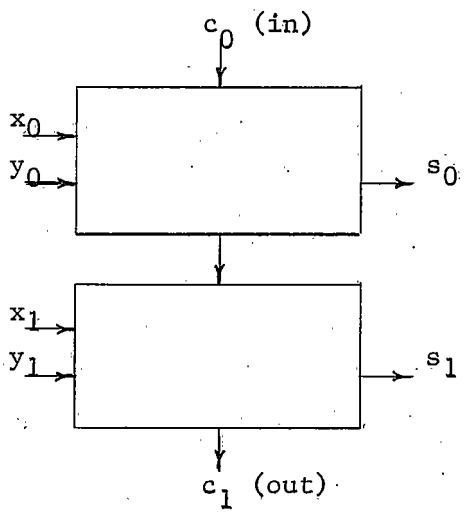


Figure 51. Iterated Two Bit Adder.

Some of the options need to be clarified. The {ONLY, OTHERS, -} option allows the user to restrict the function or variable to appear as an input or output only at the designated places, or allow it to appear at additional inputs or outputs as dictated by the algorithm or heuristics used to produce the array. As mentioned in an earlier foot-

note (page 150), some combinations of options have no meaning. For example, an internal function is never an input, a Vbuss has no right or left segments and an Xbuss has no segments at all. Column inputs and outputs refer to the y inputs connected between the outside world and top cells of the array, to the outputs from cells in the bottom row of the array, and to the vertical output from the bottom Hbuss (if any).

III.4.5 An Approach to Embedding Functions in Buss Arrays.

The embedding algorithm of section III.2 can be directly applied to buss arrays. To effect OUTPUT constraints, one need only route the outputs from where they are created to the required location. This can be done visually or with the ROUTE algorithm developed in the next section. Some hand modification of the array must be done if the desired output location is already in use. (This will occur only if the output location is a column output).

INPUT constraints can be handled in the same manner. The array is first synthesized. Then the busses are used to route the variable from the constrained input location to the cell input required by the embedding algorithm. Again adjustments may have to be made to the array if conflicts in bussing arise.

BELOW and BESIDE constraints can be handled similarly. A joinpoint is used to connect the required data to a buss line and then the data is bussed to the desired location.

RESERVE constraints can be handled for Xbusses by inserting an extra row in the array above the row with a reserved Xbuss, pushing the old row down. This new row will physically occupy the space in the array which belonged to the old row. The cells in the new row are set to function 4. This should be done before bussing inputs and outputs to required locations because the row numbers will change below the row with the reserved buss. The treatment for a reserved cell is the same except that the reserved cell is bypassed by using the busses to buss the output of the cell above to the y input of the cell below. For Vbusses and Hbusses, the treatment of a reserved segment is to avoid it when bussing inputs or outputs.

DEDICATE constraints for Vbusses and Hbusses can be treated after the embedding of the array by just specifying the variable or function to be applied and setting joinpoints and cutpoints to access the data. It makes no sense to dedicate an Xbuss since use of Xbusses is controlled by the embedding algorithm. Likewise the author can see no use for dedication of a column, since the embedding algorithm determines the use of the cascades. For the same reason, topology constraints are not usable with this algorithm.

The designer must make some decisions when routing the various data items around the array. He must route them in a fashion that has no two pieces of data requiring the same buss segment. This may at times require routing through a maze of paths which "wander" through the array. If this is not possible, the constraints cannot all be satisfied. Engineering choices of compromise must then be made.

One can see that the procedures of this section are not powerful enough to solve all constrained problems. Also, full advantage is not taken of the arrays' bussing capabilities. A cell may be sitting unused or for transmission only, when a more powerful use of the bussing structure would allow it to be used logically, opening the possibility for reduction in the size of the array. For these reasons an algorithm is developed in the next section which uses a different approach.

III.4.6 An Algorithm for Embedding Functions in Buss Arrays.

The algorithm of this section provides a method for synthesizing buss arrays. In particular, the style 4 is chosen but the algorithm could be modified to take into account any buss array, as is done for the example in section III.4.7.

The arrays produced by this algorithm are guided but not governed by the P-notation representation of functions. That is to say, the basic order in which variables and operators are embedded in the array is determined by the P-string (using the portion of the embedding algorithm which picks out the characters), but not the precise location.

The basic idea of the algorithm is to make use of the buss structure instead of cells to buss information. P-strings are embedded, as before, in a more or less left to right fashion. Variables are entered into the array, subject to the constraints, as close as possible to the cells that are to receive them. Such cells will be in turn close to the last cells used. This process begins at the upper left hand corner of the array and proceeds generally down and to the right.

Synthesis is effected through a set of procedures which may be applied algorithmically, or heuristically in an interactive process. In this section the set of procedures is set forth. In section III.4.8 modifications of these procedures are introduced for an interactive, heuristic environment.

Figure 52 sets out the general flow of the algorithm. First the constraints are specified. The various types of constraints were set forth in the previous section. This procedure of specifying the con-

straints is called CONSTRAINT. Second, the equation or equations of the functions to be realized are specified. Each equation is written in P-notation and should be written in the best form possible, governed by the principles of section III.3. This is done in the procedure INEQ. The algorithm continues with the procedure GETLIT which selects literals from the equation in a manner specified later. If the literal is a variable, a new cell is located by the LOCCEL procedure as close as possible to the last cell used by the partially realized function. All possible sources for this variable, subject to the constraints, are tabulated by GETSRC and the closest one to the cell is selected. The variable is then bussed using ROUTE from this source to an input of the cell.

The subfunction to which the variable is to be attached must then be routed from the output of the cell, where it exists, and into the remaining input of the cell into which the variable was put. This is also done using ROUTE. Another literal is then selected. If the selected literal is an operator then the cell that it specifies can be determined and this cell is then set to the desired function. If the operator is a uniting operator, specifying two cascades to be collected, additional cells or buss lines between the two cascades may have to be specified to transmit the information. This is done with

SETCEL. When all literals have been processed, the functions are embedded within the array.

If any flip-flop (storage) elements are specified, the equations realized include their set and reset equations which are denoted by capital letters. At this time, each flip-flop is located in turn and the set and reset equations are routed through the buss structure to appear at the x and y inputs of the flip-flop cell. The output of the cell is then routed through the buss structure to any points at which it is needed in the set and reset equations. This is done in the FLFL procedure.

The final step is to satisfy any output constraints. If there are none, the convention is to bring all functions to the outside of the array by the shortest path (subject to use of busses by other data). If any output demands are made, they are satisfied before any unconstrained outputs are brought out. This is called the OUTPUT procedure. The algorithm then terminates.

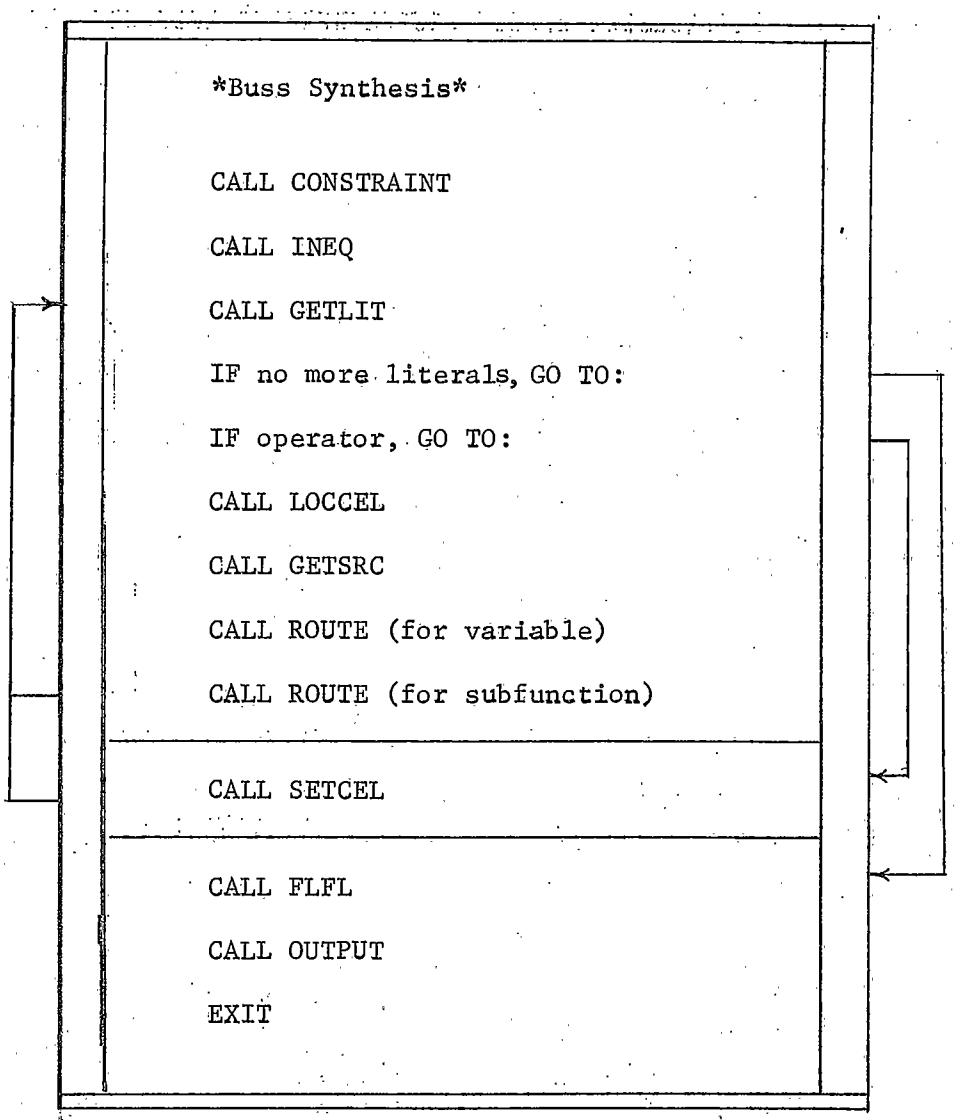


Figure 52. Flow Diagram for Buss Synthesis

At some point in the procedure a failure may be encountered. That is to say, with the function partially embedded in the array, there may be no way within the algorithm to complete the next step. For example, it may be impossible to enter the next variable into the selected cell because all busses in the neightborhood are already in use as well as neighboring cells through which the variable might enter indirectly.* In each case of failure, a step backward is taken and an alternative realization of the previous stage of the algorithm is produced. For example, if a variable cannot reach a cell from one source and a new source for that variable may enable it to reach the cell, or a new cell may be selected. If all alternatives are tried at the previous stage without success, another step backward is taken and the process of choosing alternatives repeated. If a successful alternative is found, the algorithm then proceeds as if the next stages were not solved before. The procedures are defined so as to always try a particular type of solution first at each new stage.

---

*An indirect entry of a variable occurs when a variable comes to a cell through the diagonal input from the cell to the right, and the cell to the right is used only for transmission of that variable. (I.e., the cell on the right is set to function 2,3,4 or 5).

The attempt at realization fails completely only after all alternatives have been exhausted. In this case the array must be enlarged, the constraints modified or the form of the equations modified. How the modification is made is up to the designer.

When failures are encountered, the algorithm reverses the flow shown in Figure 52. This necessitates that the user be able to identify what was done at each stage of the solution. He may keep a list of busses used, cells specified and sources found at each stage. Alternately, by looking at the P-strings and constraints he may "peel off" one stage at a time. For example, if a cell cannot be located for stage n, the algorithm backs up to stage n-1 and produces an alternate solution for routing the subfunction. If one cannot be found, the original subfunction route is cleared and an alternative variable route is searched for. Thus a stage is peeled off and the previous stage solved for an alternative answer.

The next paragraphs contain descriptions of the procedures in Figure 52. These descriptions along with the comments above and Figure 52 define the buss synthesis algorithm.

The CONSTRAINT procedure performs the tabulation of the constraints. There is no specific mathematical form of doing this, but the acceptable constraints are listed in Figure 50. The author has

made use of a semi-sentence structure in writing the constraints. (See section III.4.7). Immediately after this, the reserved and dedicated items as well as input demands and output demands should be indicated on a picture of an array.

INEQ requires that the equation or equations be written in a P-notation form. The form should be reduced as much as possible using the techniques of section III.3. Such forms are advantageous because they tend to minimize the number of occurrences of variables and the number of elementary P-strings. The basic process of embedding elementary P-strings in cascades remains, with additional flexibility as pointed out below.

The process GETLIT is the first of the more complex procedures in this algorithm. Its charge is to locate successive elementary P-strings and within each one to point out the next literal or literals to be embedded in the array. GETLIT is diagrammed in Figure 53.

The order of picking literals for processing needs explanation. It is referred to as the "next" literal in Figure 53. The first cell located is in the upper left corner of the array. Therefore, the first two literals must be the two which go with that cell. Referring to the cascade creating algorithm of section III.2.1, one can see that these are the two variables nearest the tail of the elementary P-string.

Furthermore, the one nearest the tail goes into the y input if possible. The second cell requires the next variable to the left, and so on. Therefore, after each new elementary P-string is secured from the function, one begins by choosing the rightmost variable, and its prime if any, for embedding. One proceeds to the left picking the variables in order from the P-string.

After all variables of the elementary P-string are processed, referral once again to the cascade creating algorithm shows that the first dyadic operator to the right of the last variable determines the first (topmost) cell of the cascade. Any prime following immediately after is associated with that cell also. The cell is set to the function of that operator and then modified according to the primes associated with it and the variable(s) (and subfunction) which are inputs to that cell. The function is changed to allow all inputs to come in unprimed. For example, function 10 becomes function 11 if the x input variable or subfunction has a prime associated with it. It becomes function 12 if the y input has a prime associated with it, and it becomes function 13 if both inputs have primes associated with them. Then the modified function is transformed using Table 9 (de Morgan's law) if the operator

itself has a prime following it in the P-string. The next cell is specified by the next dyadic operator (and prime, if any) in the P-string, and so on until all operators are processed.

Each elementary substring is taken from the P-string in order from left to right and each equation is taken in turn. When this is completed all equations are realized and the algorithm proceeds to the FLFL and OUTPUT sections.

LOCCEL is the process of locating the next cell to be used in realizing the P-string. Remaining as close as possible to the cascade realization algorithm, the next cell selected is normally the next cell in the column of the array below the last cell used. If this cell is unavailable the next lower one is selected, and so on down to the bottom row of the array. If another cell is required, the first cell in the next column is selected and this column is treated as an extension of the first. Each time a new elementary P-string is selected a new cascade is begun at the top of the array in the first column of the array that is vacant.

GETSCR is the procedure charged with identifying all possible sources of a variable. If an ONLY type demand has been made in the constraints, the sources will include only the input locations demanded. If not, all unused (undedicated, unreserved and not previously

used by another variable) edge connections to the array are included. If the variable has been previously used in the array, any buss segments to which it is attached are also included as sources. Sources are then ordered by distance from the destination cell input with the closest one being the one tried first.

ROUTE finds a path through the buss structure from the source to the destination input. For a variable the first tries are to the x input of the desired cell. If no path is found, an indirect input is tried. If the cell where the input is desired is cell(i,j), the first indirect input cell is cell(i,j+1). Other indirect input cells allowed are cell(i-1, j+1), cell(i,j+2) and cell(i-1,j+2). The algorithm treats each of these in turn as a destination cell and all buss routes are tried first to the x input and then to the y input. If successful, the indirect cell(s) are set to functions 2 or 4 as necessary to buss the variable into the x diagonal input of the original cell. If this fails then the y input of the original cell is selected as the destination and a path is attempted from the source to the y input. This last step is not allowed if the cell being specified is directly below the last one used. (This is because the output from the previous cell would have little chance of getting to the x input).

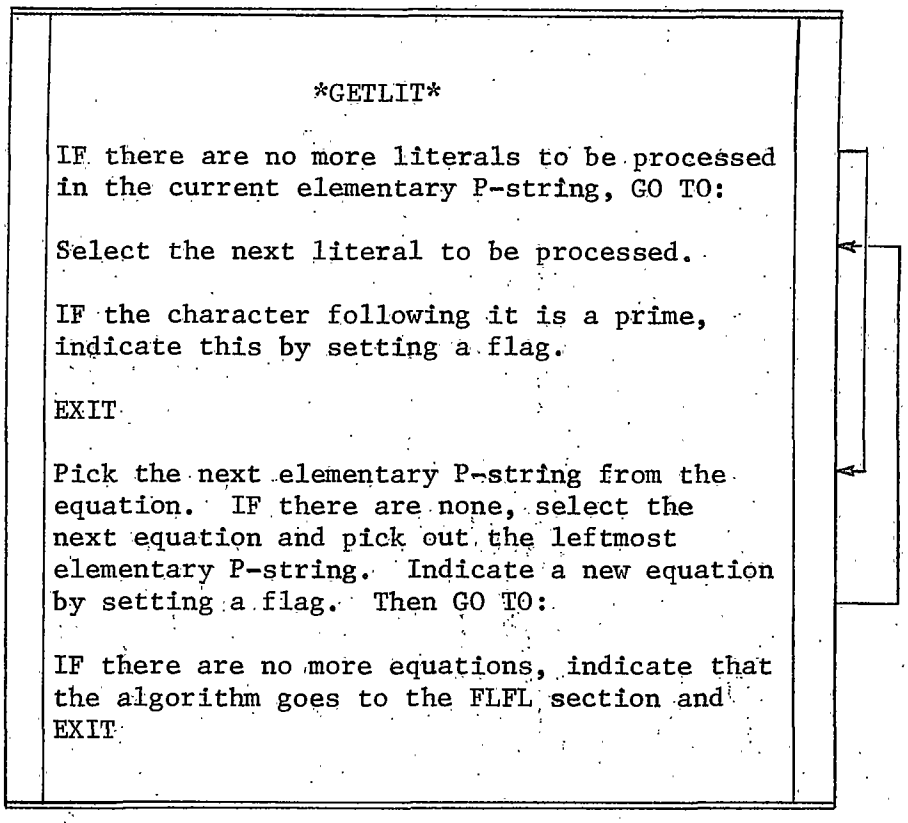


Figure 53. Flow Diagram for GETLIT.

The input for the subfunction to the current cell is either the y or x input. It is the one left after the variable is entered. If it is an x input, the same procedure is followed as is used for putting a variable in the x input. If it is a y input, the direct buss is used as above for y inputs, except that the case is allowed where the cell is directly below the last one. If the cell is in the same column as the one from which the subfunction originates, the route is through cells between the two cells, setting the intervening cells to function 4.

The actual bussing or routing procedure is illustrated in Figure 54. Consider the group of nodes shown as follows. Large circles are cells. Dots above, below, right and left of these large circles stand for the Vbuss segments, Hbuss segments and Xbuss surrounding the cell. The +'s diagonal from the circles represent joinpoints on the busses--corners where data paths may change direction. Figure 55 further classifies the nodes by the odd-even characteristic of their indices.

For each application of the routing algorithm there is a source and a destination. Sources can be Vbuss or Hbuss segments, Xbusses or columns. If the data is taken from the output of the cell, the source is assumed to be the Hbuss segment below the cell (unless a direct connection to the y input of the cell immediately below is desired).

If an Xbuss is the source it can only be used as a source for the cells in its own row, unless that variable has an ONLY demand upon it. Then a cell in that row may be set to function 4 and its output considered as a source. Likewise, if a variable is input ONLY to a column and the Hbuss above the first cell in that row is reserved, the cell in row one may be set to function 4 and its output considered as a source.

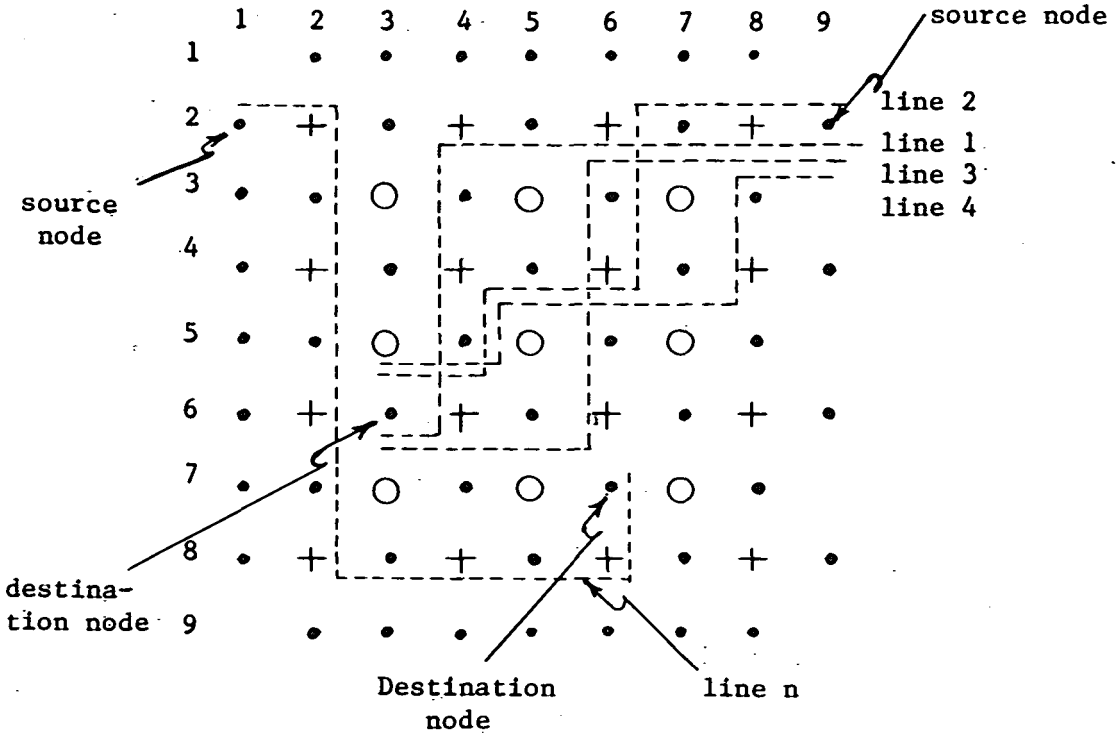


Figure 54. Illustration of the Routing Algorithm.

Characteristic of indices i	j	Condition	Meaning of node
odd	odd	1<i<9, 1<j<9	cell
		i=1	column input
		i=9	column output
		j=1	Xbuss
even	even	---	joinpoint (corner)
odd	even	---	Vbuss segment
even	odd	---	Hbuss segment

Figure 55. Classification of Nodes in Figure 53.

Destinations are cells, buss segments and column outputs. If the source makes a direct connection to a cell, (from a neighboring cell or column input), no bussing is necessary. Otherwise, if a cell is the destination, the Vbuss or Hbuss respectively to the right of or above the cell is used as the destination (unless a direct connection is made from the cell above.)

In all cases where a direct connection is not used, the routing algorithm is used. First a direction set is assigned from source to destination. The four direction sets are: (1) left and up, (2) left and down, (3) right and up, and (4) right and down. A path is then traced from source to destination with each segment of the path related to the previous segment by one of the two directions in the assigned direction set. No other paths are allowed. The first path traced is found by moving vertically to the first joinpoint encountered, (if necessary), moving horizontally to the column of the destination, or the column just before in the case of y inputs to cells and Hbuss destinations, and then moving vertically to the destination or, in the case of y inputs to cells and Hbuss destinations, to the joinpoint nearest the destination. In these cases, one horizontal segment is added to finish the buss. This is illustrated in Figure 54 by line one, which is the principal path from node(2,9) to the y input of cell(3,3). In case that a segment is used previously, alternate paths are generated. A corner in the path nearest the destination which fits the criterion expressed in Figure 56 is searched for, and reflected about the dashed line. The path is then traced horizontally as if the new corner vertex were the source. Lines two through four in Figure 54 illustrate how the algorithm walks through all possible paths which do not violate, except in some cases for the last path segment, the direction con-

straints. Line n in Figure 54 is such a case. It illustrates the last path to be tried from node(2,1) to node(7,6).

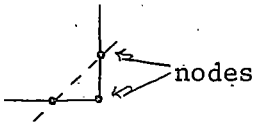
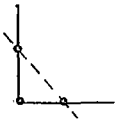
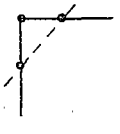
Direction criterion	Corner orientation
up, right	
up, left	
down, right	
down, left	

Figure 56. Corner Criterion for a New Path in Routing Algorithm.

After a path has been found, one should make some designation showing joinpoints and buss segments used. The author has found it convenient to draw an array, to circle joinpoints and cutpoints in pencil as used, and to circle the inputs connected to the cells. Pencil allows erasure during the iterative process.

The procedure SETCEL serves two purposes. First, if the operator is to specify a cell into which two variables or a variable and a subfunction have been put, then the cell is set to the function specified by the operator. The cell function must be modified if there are primes involved with it or the inputs to the cells, as explained on page

The second function of SETCEL is to unite cascades when uniting operators are encountered. A uniting operator must unite two cascades which are not necessarily adjacent. Let cell(i,j) be the cell in which the cascade on the left is completed. Let cell(s,t) be the cell in which the cascade on the right is completed. Note that $t \geq j + 1$. Three cases can occur. For case one let $s = i + 1$. Then cell(i + 1, j) is set to the uniting operator and cells in row i + 1 between columns j and t are set to function 2. For case two, let $s > i + 1$. In this case cells(i + 1,j), (i + 2,j)... (i + r,j), where $i + r = s - 1$, are specified as function 4, and i + r replaces i in a case one solution. For case three, $s < i + 1$ is left. Here cells(s + 1,j), (s + 2, j)... (s + r,j), where $s + r = i + 1$, are specified to function 4, and s + r replaces s in a case one solution. In other words, cells in one cascade are specified to buss the y input until the output from the cascade on the right is available to the diagonal x inputs of the row

below the left cascade. The cells between the two cascades are set to buss the information. Of course, primes associated with either argument or the uniting operator cause modification of the uniting cell. (See pages 62, 166).

The procedure FLFL has three functions: locating of a cell for the flip-flop, bussing of the set and reset functions to that cell, and bussing of the output from this cell (the internal function) to cell inputs where it is required. To find this cell, unused cells in the array are ranked according to their distance from the point of appearance of the reset equation for that flip-flop. The distance criterion is the number of buss segments that it would take by the most direct route to reach the y input of the flip-flop cell with the reset function. The cell directly below the cell producing the reset equation would be selected first since it uses zero busses. The reset equation is then bussed into the y input, using the routing algorithm if it is not directly below the cell producing the reset function. The set function is then bussed into the x input. If it appears one cascade to the right, the path is taken through cells in that column. Otherwise, the first try is through cells. Enough cells in the cascade producing the set function are set to function 4 to bring it to the row of the flip-flop. Diagonal inputs are then used and cells of the row

set to function 2. This method works only if the reset equation is realized in a column to the right of the flip-flop cell. If this method fails the routing algorithm is used, allowing both direct and indirect entry. If the selected cell cannot be used for the flip-flop, cells below it in that cascade are tried in turn. (They all have a distance of zero). The cells between the reset equation and the cell selected in the cascade are set to function 4. If none work, then cells of distance one are tried, using the routing algorithm for the set function, and so on.

Once the flip-flop cell has been located and the input equations brought in, the output of the cell (the internal variable) is bussed to locations at which it is required as an input to cells. This is done using the routing algorithm, treating the internal function as a variable appearing on the Hbuss below the flip-flop cell.

The final procedure in the realization algorithm is the OUTPUT procedure. It is charged with responsibility for satisfying BESIDE, BELOW, and OUTPUT demands and routing unconstrained outputs to the array edge. When it is invoked, all functions are available in some location within the array, including outputs from flip-flops (internal functions). The demands are satisfied in the following order: (1) OUTPUT, (2) BELOW, (3) BESIDE. If some functions have no demands upon

them they are brought to the edge of the array, after all demands are satisfied, at the closest point to their appearance, using the distance criterion set forth in the previous paragraph describing SETCEL.

BELOW demands are of two types. One type is a demand for a function to appear below a particular input, wherever that input appears. The second is a demand for a variable to appear as an output below any occurrence of an input of that variable or another. In the first case, the function is bussed to the required location using the routing algorithm unless it already appears in the right cascade. In that case it is brought out by setting cells below it to function 4. In the second case, the routing algorithm is used allowing all buss segments upon which the variable appears as sources.

BESIDE demands are treated in exactly the same way as BELOW demands. OUTPUT demands are treated similarly. A direct output through cells of the same cascade is used for a function or variable if possible. If not, the routing algorithm is used.

If an intermediate function is specified with an OUTPUT demand, then this intermediate function is treated just like a function. Intermediate function demands are satisfied after function demands. If a function, intermediate function, or internal function is demanded to be complemented and brought out, first a search is made to see if it

exists in that form. If so, the demand is satisfied like any other OUTPUT demand. If not, a cell must be specified to complement it. The first choice for this cell is the one nearest the output location. First a check is made to see if the output of this cell can be bussed to the desired output location directly or by route. If not, another cell as close as possible is tried until one is found which has an output that can be bussed to the desired location. Then the function is brought into the x input directly or indirectly or if that fails, into the y input, using ROUTE. If these all fail, another cell is selected.

This completes the description of the procedures used in the realization algorithm. The possibilities for alternate solutions in each part are the only ones allowed. However, nothing was said as to what is to be done when a failure is encountered in a procedure. Table 15 corrects this omission. It is an exhaustive listing of what is to be done within a procedure when a failure is encountered. The phrase "back up one step" indicates that one is to back up to the last procedure before the failure and find the next solution as specified by that procedure.

TABLE 15. FAILURE MODES AND RECOVERY PROCEDURE FOR THE BUSS SYNTHESIS ALGORITHM.

Process where failure occurs	Conditions at Failure	Remedy
Constraint definition	Conflicting constraints	Modify constraints
Equation definition	Equation not well formed	Correct the equation
Locate cell	Process backed up to this step	Locate next cell
	The last available cell has been tried	Back up one step
Find source	Process backed up to this step	Try next source
	Source failed to buss to destination	Try next source
	No more sources	Back up one step
Route variable	Process backed up to here	Try next route
	Route to x input fails	Try next route to x input
	No more routes to x input	Try indirect route
	Indirect route to x input fails	Try next indirect route
	No more indirect routes	Try first route to next indirect cell
	No more indirect cells	Try route to y input if cell is not immediately below last used cell. If not, back up one step.

TABLE 15. Continued.

Process where failure occurs	Conditions at failure	Remedy
Route subfunction	Process backed up to here	Try next route
	Route to y input fails	Try new route to y input
	No more routes to y input	Back up one step
	Route to x input fails	Try next route to x input
	No more routes to x input	Try indirect route
	Indirect route to x input fails	Try next indirect route
	No more routes to indirect cell	Try first route to next indirect cell
	No more indirect cells	Back up one step
Set cell	Process backed up to here	Back up one step
	Failure in joining cascades	Back up one step
Locate flip-flop	Fail to locate cell	Back up one step
	Fail to route x or y inputs to cell	Locate new cell
	Fail to route internal function to array location	Locate new cell
Buss outputs	Fail to route output to satisfy OUTPUT demand	Back up one step
	Fail to route variable or function for BELOW or BESIDE	Try new route for OUTPUT demand

In summary, in this section an algorithm has been described which will realize arbitrary Boolean functions, including functions for flip-flop (storage). The algorithm allows the use of constraints sufficient for realization of iterated problems within one array. The next section is an example synthesis using this algorithm. In it several different functional cells are realized and tied together in one array. The entire array is a realization of Cannon's multiplier.

III.4.7 An Engineering Example--A Cellular Parallel Multiplier

Figure 49 illustrates Cannon's multiplier. This multiplier is considered by the author to be a significant engineering example problem. In this section, a realization is given for this multiplier in terms of style 6 arrays. The realization exercises the constraints and the various procedures of the buss synthesis algorithm.

Figure 57 shows a style 6 array. It differs from a style 4 array in that the Vbusses and Hbusses have cutpoints only after every other joinpoint of a Vbuss with an Hbuss, instead of after every one as the style 4 array does. This arrangement saves two cutpoints for every four cells, realizing a savings of about ten per cent in cutpoints. There are, of course, cases where extra cells will be needed because of the inability to separate each buss segment.

This array requires only a simple modification of the buss synthesis algorithm. When an Hbuss or Vbuss segment is specified, one of the two in juxtaposition with it is also specified. The original segment and the segment lying next to it and not separated by a cutpoint are specified. The other neighbor remains free.

Figure 57 illustrates the functional decomposition of the multiplier. Except for the block labeled logic, each block consists of several functionally identical cells of the indicated type. The number of cells in each block is indicated by the subscripts on the vertical inputs. Except for the logic block, each vertical input affects only one cell while the horizontal inputs affect all cells of the block. The notations conform to that of Figure 49, and lines which cross the dashed line surrounding the blocks are connections to the outside world.

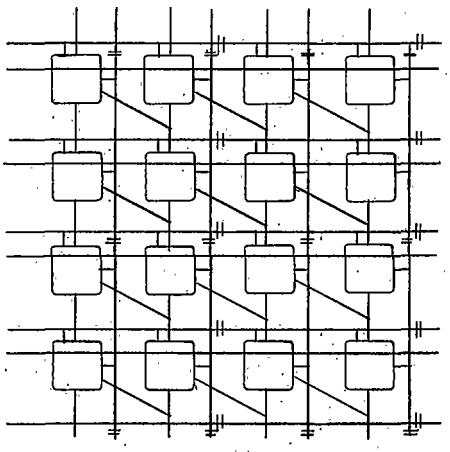


Figure 57. A Style 6 Array.

The author realized this multiplier by an iterative process beginning with the exponent portion of the multiplier register. A beginning set of constraints was imposed including array size and BELOW requirements upon the outputs of the flip-flop cells. At various stages of the solution requirements imposed upon the synthesis by new functional cells, caused earlier functional cells to be redesigned. For example, the requirement that b inputs be from the left side of the array requires reservation of a buss through the multiplier

register. Which buss is needed can be determined only after a cell is realized for the multiplicand register. To minimize the iterations, the designer should look as far ahead as possible. He must take into account such things as busses needed to transfer data, location of input to cells realized in other parts of the array and the size needed to realize other cells when they must be placed in juxtaposition with earlier ones. This ability is highly intuitive and gained by experience.

In general, the constraints used in the example required that the inputs only be from the top and left, while the outputs could appear only at the right and bottom. The specific constraints presented evolved out of the iterative process of redesigning functional cells to satisfy I/O constraints of functional cells placed in juxtaposition with them. Only the final designs of the cells are presented here.

Figure 59 shows the block diagram for the cellular realization, divided into blocks labeled with alphanumeric names. The alphabetic characters correspond to the cell types in Figure 58. The numbers indicate how many versions of the cell were tried during the iterative process. For example, DSR11 indicates that the cell was the eleventh different D-SR flip-flop synthesized. Each of the square labeled blocks contains a four by four style 6 array. A space is shown be-

tween blocks, though in the actual array there is none. Input and output lines which extend slightly within the blocks are Xbuss or column connections. Vbuss and Hbuss lines terminate at the edge of the blocks. Busses which are used to carry information through a block are indicated by drawing a solid line through that block. Small letters and capital Y's appearing just to the right or above the I/0 lines they label correspond to the internal functions and variables used to produce the blocks. Labels in parentheses correspond to the notation of Figures 49 and 58. Other capital letters indicate externally wired connections.

The realization of each of the blocks is illustrated below. The constraints are indicated and in some cases the steps followed by the buss synthesis algorithm are included. The reader can follow the alternatives by use of Figures 52-56 and Table 15.

Figure 60 is the Realization of DFF2. The constraints used are:

1 USE EXACTLY 4 ROWS
2 USE EXACTLY 4 COLUMNS
3 DEMAND i INPUT ONLY COLUMNS 2,4
4 DEMAND i BELOW i
5 RESERVE ALL HBUSS 1
6 DEMAND INTERNAL FUNCTION Y OUTPUT RIGHT HBUSS 4
7 DEMAND d INPUT ONLY XBUSS 1

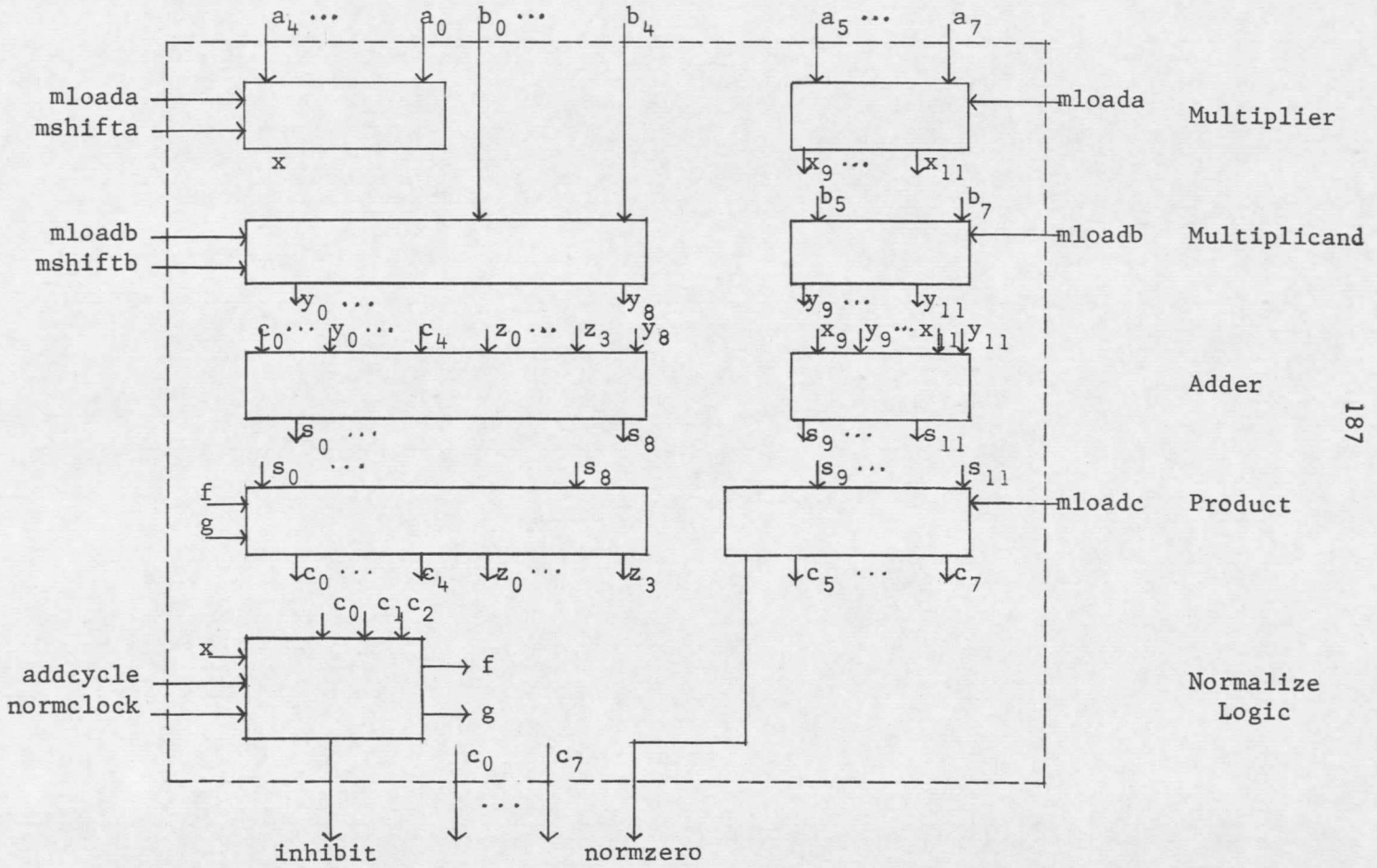


Figure 58. Functional Decomposition of Cannon's Multiplier

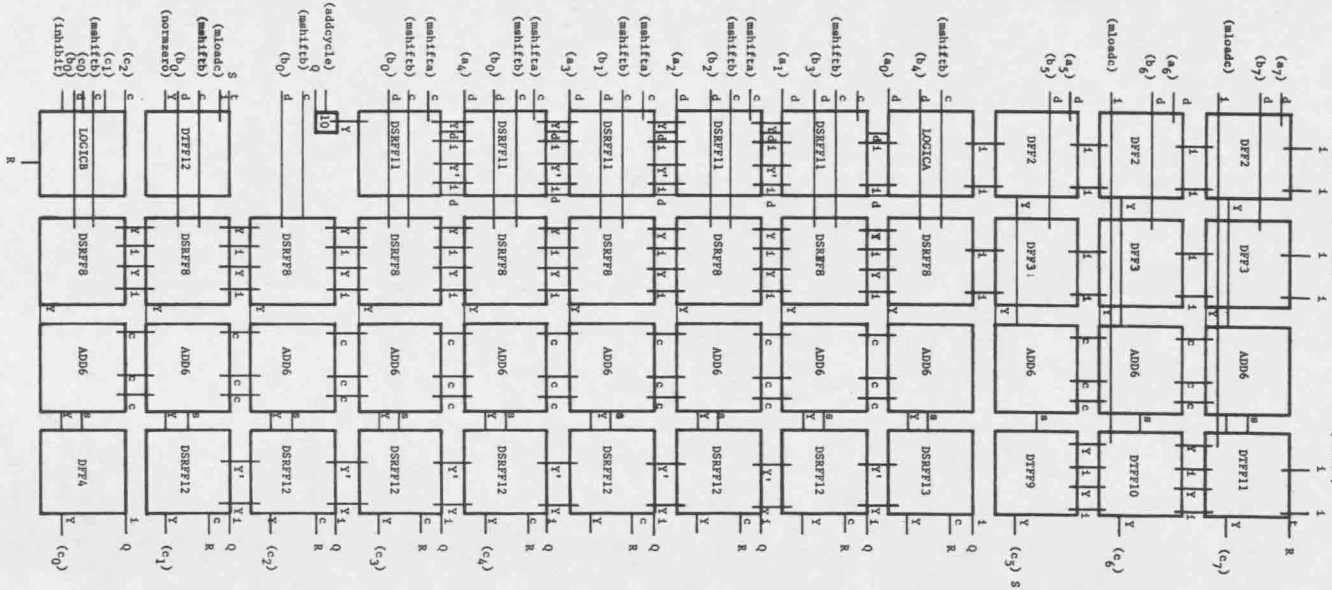


Figure 59. Block Diagram of the Cellular Realization of Cannon's Multiplier

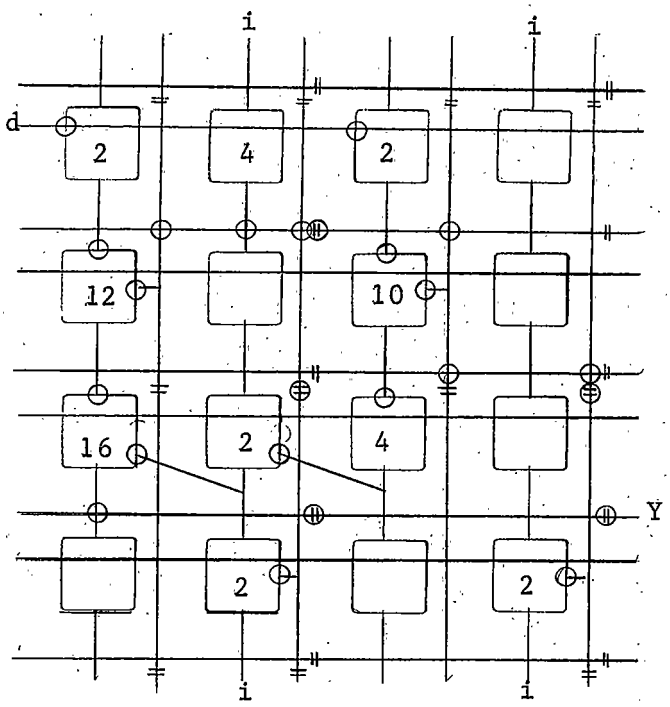


Figure 60. Realization of DFF2.

The equations for a D flip-flop may be written in P-notation as:

r = d'ix

s = dix

The realization proceeds as follows: Cell(1,1) is chosen as the starting point for r. The first variable is i. However, i cannot reach the y input to this cell while d can only enter the x input. Therefore, the algorithm selects the next cell, cell(2,1). Since d and i are restricted by ONLY demands to an XBUSS and COL inputs, a cell is allowed to get them onto the buss lines. Thus d can go in the x input. How-

ever, this uses up the Hbuss necessary to get i into the y input (even entering d in the indirect mode, which is tried after the direct one), so that a try is made to put d in the y input. This succeeds by the direct path from cell(1,1). The next character in the P-string is an operator which specifies cell(2,1) as Function 10, which is modified to Function 12 because of the prime on d in the equation. This procedure essentially repeats itself beginning in column three for the s equation, except that i is now present on Hbuss two and this becomes the source since the cell to get the i into column four is used only as a last resort. The flip-flop is located next. The first choice is cell(2,1) since it is the first under the reset function. The set equation goes into the x input through two additional cells, as was detailed when FLFL was described. The output demands are next satisfied. First the internal function Y is bussed, using the routing algorithm, to its required location. Then i is bussed to its BELOW demand location. The routing algorithm chooses the path shown to the x input of cell(2,4). It must also route i to cell(4,4). Moving horizontally then vertically yields the path through Hbuss three and Vbuss four. (The closest source was chosen).

Three of DFF2 cells make up the exponent portion of the multiplier register. The Y outputs correspond to $x_9 - x_{11}$ in Figure 58.

Consider next DSRFF11. Figure 61 is the realization. The constraints are:

```
 1  USE EXACTLY 4 COLUMNS
 2  USE EXACTLY 4 ROWS
 3  RESERVE HBUSS 1
 4  RESERVE XBUSSES 2,3
 5  DEMAND INTERNAL FUNCTION Y' OUTPUT COLUMN 1
 6  DEMAND INTERNAL FUNCTION Y OUTPUT COLUMN 3
 7  DEMAND INTERNAL FUNCTION Y OUTPUT HBUSS 4 RIGHT
 8  DEMAND i INPUT ONLY COLUMNS 2,4
 9  DEMAND r INPUT ONLY COLUMN 1
10  DEMAND s INPUT ONLY COLUMN 3
11  DENABD j INPUT ONLY HBUSS 5
12  DEMAND j OUTPUT ONLY VBUSSES 2,4 BOTTOM
13  DEMAND d INPUT ONLY VBUSSES 2,4 TOP
```

The equations are:

```
R = rc×id'×+

S = sc×id×+
```

The realization proceeds as follows. Cell(1,1) is located as a first choice. Variable s is selected to try for the x input. It succeeds via the Xbuss (a first choice). The next variable is r. It is demanded in column 1 and goes directly to cell(1,1). Next, the first

× operator specifies cell(1,1). The next elementary string is id'×. This is begun in column two. The first variable is d. It is routed into the x input of cell(2,1). The second variable, i, goes directly to the y input. The next character is the second ×. It specifies cell (2,1) as function 10 which is modified to function 11 because of the prime on d. Next, the operator + is processed. It unites the two subfunctions as shown in SETCEL, through cell(2,2). Thus r emerges from cell(2,1). The procedure is identical for the s equation. FLFL then locates the flip-flop under the reset function in cell(3,1). FLFL also busses s to cell(3,1) through cells(3,3) and (3,2). OUTPUT must then produce Y' in column 1. This is done directly through cell(4,1). The function is modified from 4 by the prime on Y. Additionally the constraints require a Y output from column three. The first try is to use cell(4,3) as a function 2. This is done by routing Y from cell (3,1). This leaves j which is straightforwardly routed to the Vbusses two and four.

Five of the DSRF11 cells make up the multipler mantissa. They are placed as shown in Figure 59. The functional cell above these labeled LOGICA will serve to illustrate the iterative process. All that is required of it now is that it buss i and d. Later it will be redesigned. The constraints for the LOGICA cell are:

```
1  USE EXACTLY 4 ROWS
2  USE EXACTLY 4 COLUMNS
3  RESERVE ALL HBUSS 1
4  DEMAND i OUTPUT ONLY COLUMNS 2,4
5  DEMAND i OUTPUT BELOW i INPUT
6  DEMAND d INPUT HBUSS 5
7  DEMAND d OUTPUT VBUSSES 2,4 BOTTOM
```

The realization consists of bussing i and d. First i is bussed. Cell (2,1) is used to get it into the buss structure and from there the routing algorithm is used. The routing algorithm busses d also. Note that a failure is encountered in doing this. See Figure 62. Any attempt to buss i directly into the x input of cell(4,2) causes a conflict when the attempt to buss d is made. The first alternate is to route i indirectly through cell(4,3). This is done successfully and the algorithm goes on to the second output of i and the bussing of d. A second conflict occurs with putting i into the x input of cell(4,4). Vbuss four will be needed for d. The alternate of indirectly bussing i to the x input cannot be done, (no cells to the right), so the y input is tried. This is successful and d busses successfully, as shown in Figure 63.

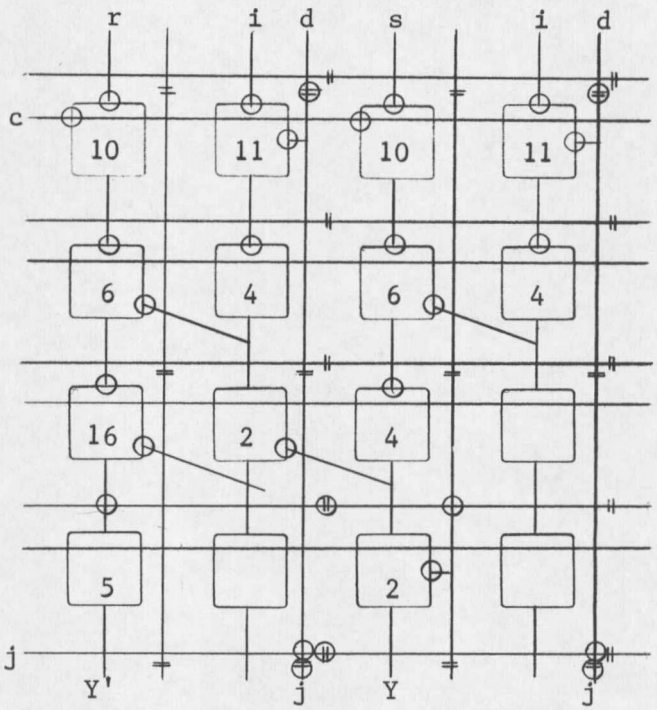


Figure 61. Realization of DSRFF11.

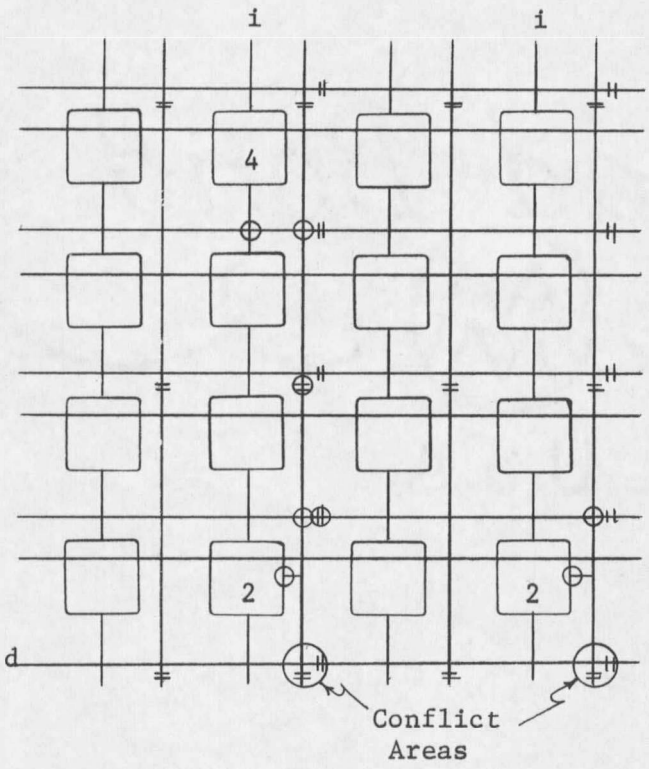


Figure 62. Unsuccessful Attempt at Realization of LOGICA.

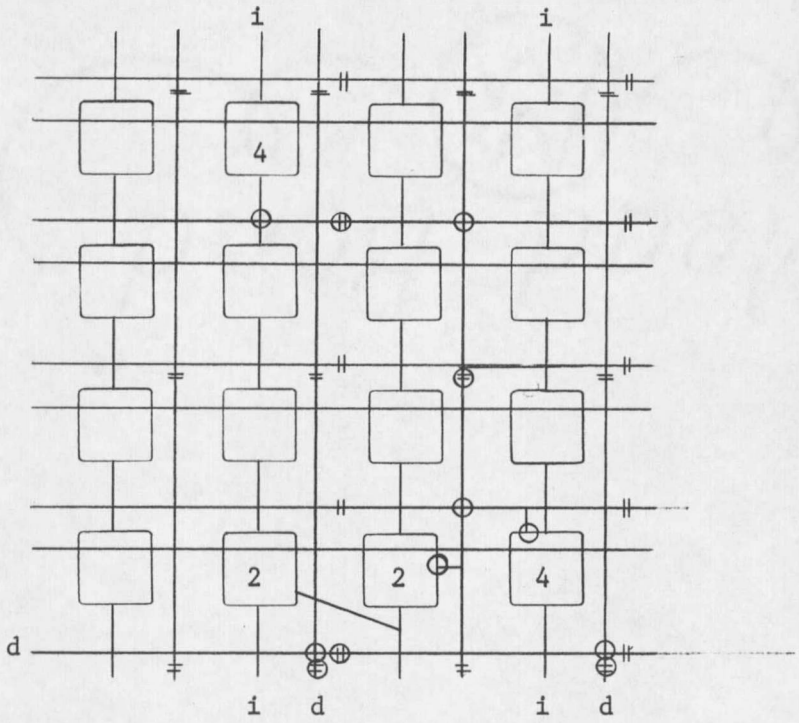


Figure 63. Realization of LOGICA.

Consider next DFF3. Figure 64 is the attempted realization. The constraints are:

1 USE EXACTLY 4 COLUMNS
2 USE EXACTLY 4 ROWS
3 RESERVE XBUSS 1
4 RESERVE ALL HBUSS 1
5 RESERVE ALL HBUSS 4
6 DEMAND i INPUT ONLY COLUMNS 2,4
7 DEMAND i OUTPUT BELOW i
8 DEMAND INTERNAL FUNCTION Y OUTPUT HBUSS 5 RIGHT
9 DEMAND d INPUT ONLY XBUSS 2

The equations are written as for DFF2.

r = d'i×

s = di×

Cell(1,1) cannot receive both d and i. Thus the first alternate cell (2,1) is selected. The x input is d and i comes out of cell(1,2) to get to the y input. Function 11 is placed in the cell because of the prime (and the × operator). Column three is used to begin function s. The rest of the procedure goes as explained before down to FLFL. Cell(3,1) is chosen first for the flip-flop, but Hbuss four is reserved and the algorithm allows no way for y to get to Hbuss five. Therefore, the first alternative is chosen for the flip-flop cell. This is cell(4,1). FLFL then busses s and r into cell(4,1). However, now there is no way to buss i to column two as required. Running

through all available cells for the placement of the flip-flop, none alleviate the problem. Backing up a step the creation of s is moved down the column and through the next column without success.

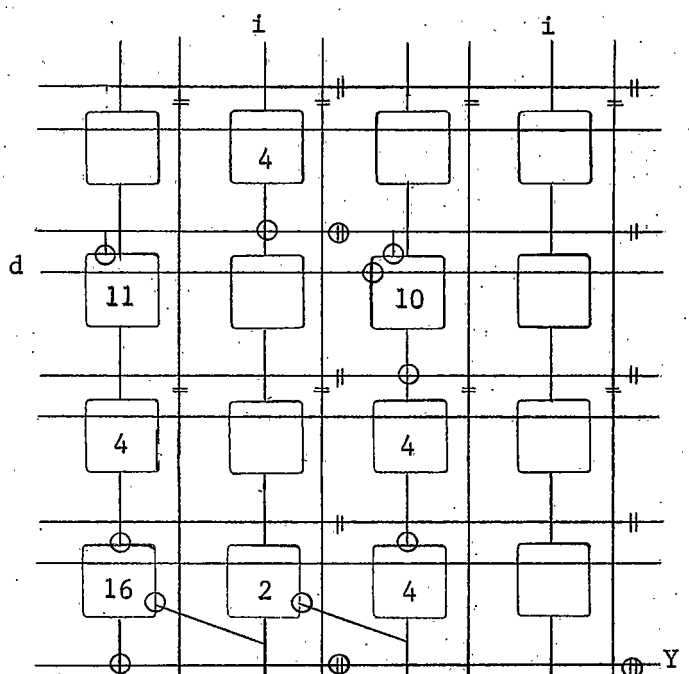


Figure 64. Unsuccessful Attempt at Realization of DFF3.

The designer may wish to continue with the algorithm at this point. However, a solution is easily obtained by noting that one can realize s and r in the opposite order. This leads to use of cell(4,3) as the flip-flop cell by the algorithm. Cells (4,2) and (4,4) can now be reached by OUTPUT after Y is put out on Hbuss five as required. Thus the functional cell can be realized as in Figure 65.

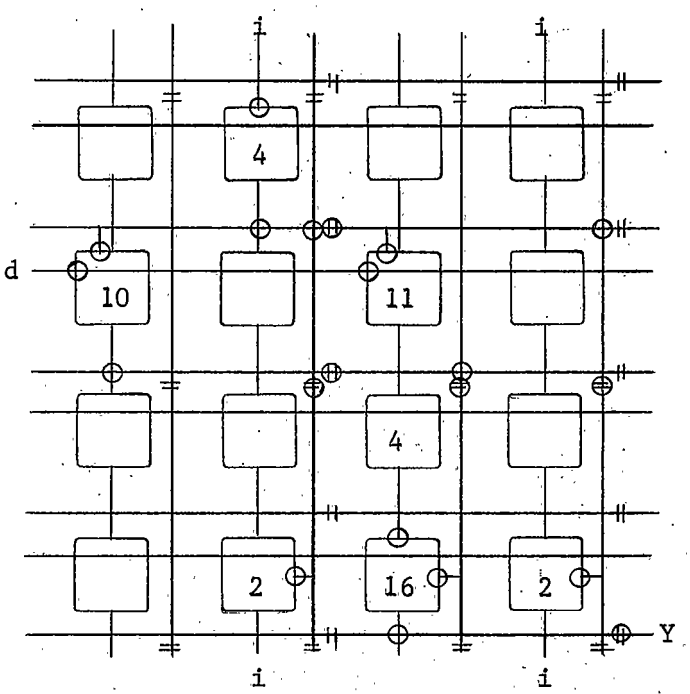


Figure 65. Realization of DFF3.

Next consider DSRFF8. It is similar to DSRFF11. The constraints are:

1 USE EXACTLY 4 ROWS
2 RESERVE ALL HBUSS 1
3 USE EXACTLY 4 COLUMNS
4 DEMAND c INPUT ONLY XBUSS 2
5 DEMAND d INPUT ONLY XBUSS 3
6 DEMAND r INPUT ONLY COLUMN 1
7 DEMAND INTERNAL FUNCTION Y BELOW s
8 DEMAND INTERNAL FUNCTION Y' BELOW r
9 DEMAND i INPUT ONLY COLUMNS 2,4

The equations are written as for DSRFF11. The realization is straightforward using the algorithm and the result is Figure 66. One different alternative is called for that has not been used previously. This is the use of the routing algorithm to bring s to the flip-flop cell. This alternative is reached by the algorithm because of unsuccessful attempts to satisfy the i OUTPUT demand.

The next group of cells is the ADD6 group. Figure 67 shows the realization of one cell. Historically, the adder cells caused the redesign of the multiplier and multiplicand register cells. This was caused by the necessity of data from the multiplicand registers to the left appearing in this cell. (Each add cell is a full adder, and such a cell was realized in a style 1 array in Figure 36). The equations are written as:

s = wzj⊕⊕

c = zj×wzj+×+.

The sum equation is s and the carry function is c. The equations are ordered so that j, which is the carry in, will most naturally fit into a vertical input. The inputs w and z come from the multiplicand and product registers, respectively. Thus z must come from the right and w from the left. The input w is fixed in location by DSRFF8 at Hbuss five. The z location required for the z input is not known at this point, but it will turn out that Hbuss four will work. The carry in may enter as many vertical inputs as necessary. One must only require that the carry out appear below each carry in. The constraints are:

```
1  USE EXACTLY 4 ROWS
2  USE EXACTLY 4 COLUMNS
3  RESERVE HBUSS 1
4  DEMAND FUNCTION c BELOW j
5  DEMAND z INPUT ONLY HBUSS 5 LEFT
6  DEMAND w INPUT ONLY HBUSS 4 RIGHT
7  DEMAND j INPUT ONLY TOP
8  DEMAND FUNCTION c OUTPUT HBUSS 3 RIGHT
```

The realization is straightforward down to satisfying BELOW demands. Here for the first time, an indirect bussing of c is used to get this function to column one.

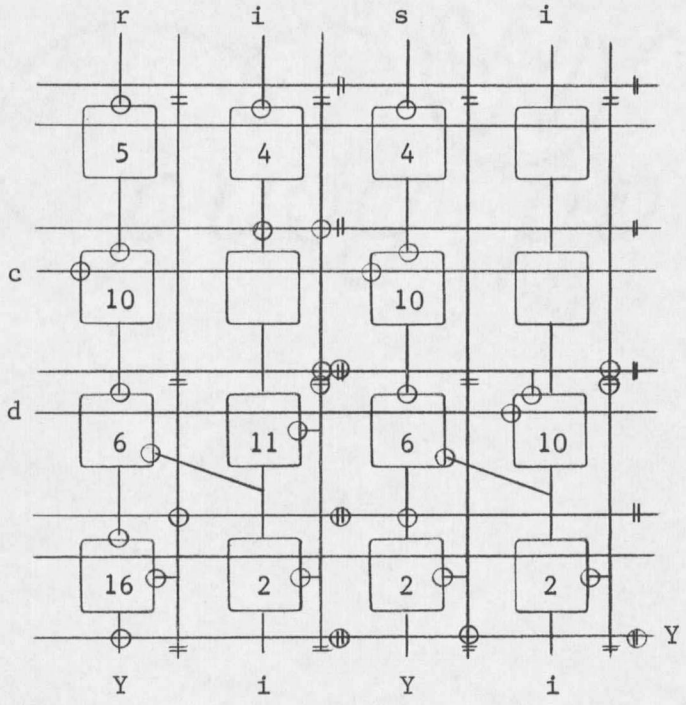


Figure 66. Realization of DSRFF8.

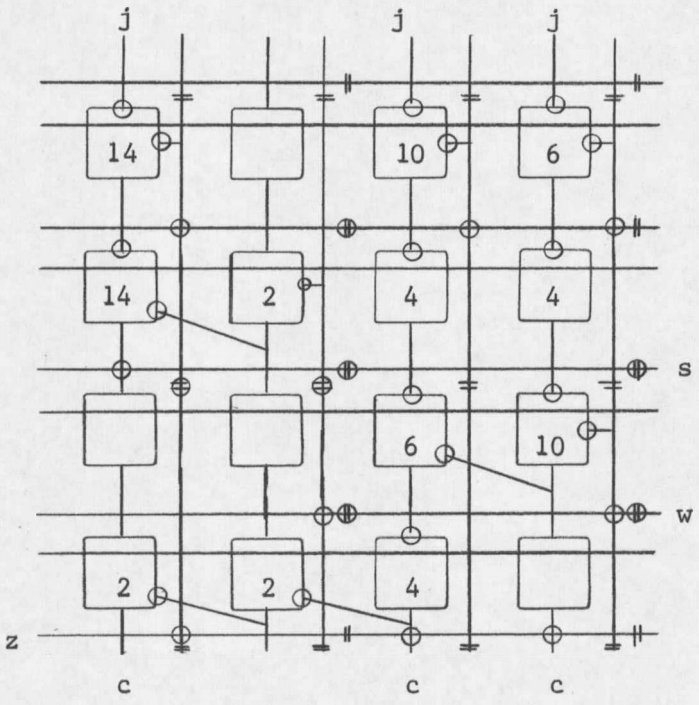


Figure 67. Realization of ADD6.

DTFF9 is shown in Figure 68. The constraints are:

1 USE EXACTLY 4 ROWS

2 USE EXACTLY 4 COLUMNS

3 RESERVE ALL HBUSS 5

4 DEMAND INTERNAL FUNCTION Y OUTPUT HBUSS RIGHT

5 DEMAND t INPUT ONLY TOP

6 DEMAND i INPUT ONLY TOP

7 DEMAND INTERMEDIATE FUNCTION Y'tx OUTPUT HBUSS RIGHT

8 DEMAND d INPUT ONLY HBUSS 3 LEFT

The equations are:

r = Yt di×+

s = Y't×d'i×+

It is convenient to write the equations as shown since t and i are constrained to be top inputs and they are the first variable encountered, causing them to be placed first in the y input to a top row cell. The d input is primed with respect to normal equations for the set and reset equations because of the inverter to this functional cell, shown in Figure 49.

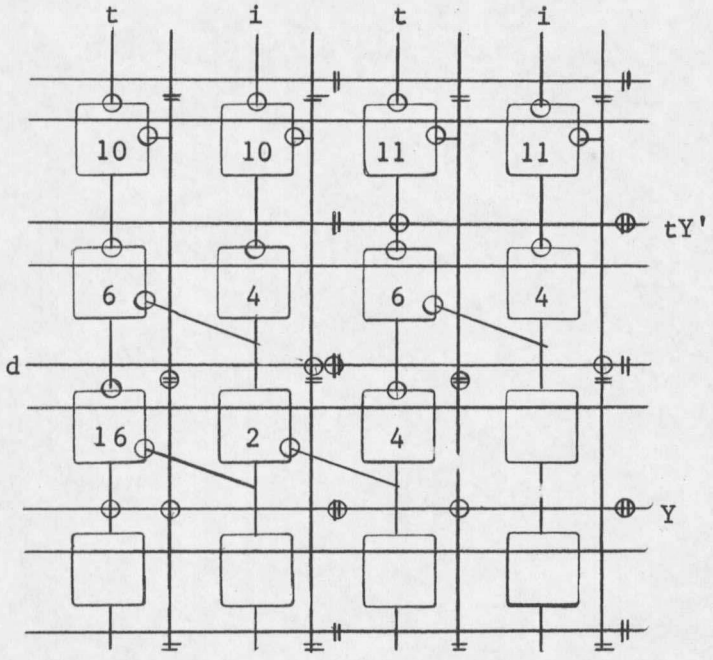


Figure 68. Realization of DTFF9.

An intermediate function is required as output in this example. It is treated like any other function and routed out as required in the OUTPUT algorithm.

DTFF10 is another element of the product term exponent register. Its equations are the same as DTFF9. Figure 69 is its realization. It is constrained, of course, by the previously realized cells. It is desired to bring in i and d from the left to this register. If Xbuss four is used in DTFF10, its fourth row can be used as a column output to DTFF9. Xbuss four is unused in other functional cells to the left, so this will allow i to be on input from the left side of the array. Y'tx is not required as an intermediate function for this cell. Since the Y output is the trigger (t) for DTFF9, it must be the output of columns one and three. The constraints are:

1 USE EXACTLY 4 ROWS

2 USE EXACTLY 4 COLUMNS

3 DEMAND j INPUT ONLY XBUSS 4

4 DEMAND j OUTPUT COLUMNS 2,4

5 DEMAND INTERNAL FUNCTION Y OUTPUT COLUMNS 1,3

6 DEMAND d INPUT ONLY HBUSS 3 LEFT

7 DEMAND INTERNAL FUNCTION Y OUTPUT HBUSS RIGHT

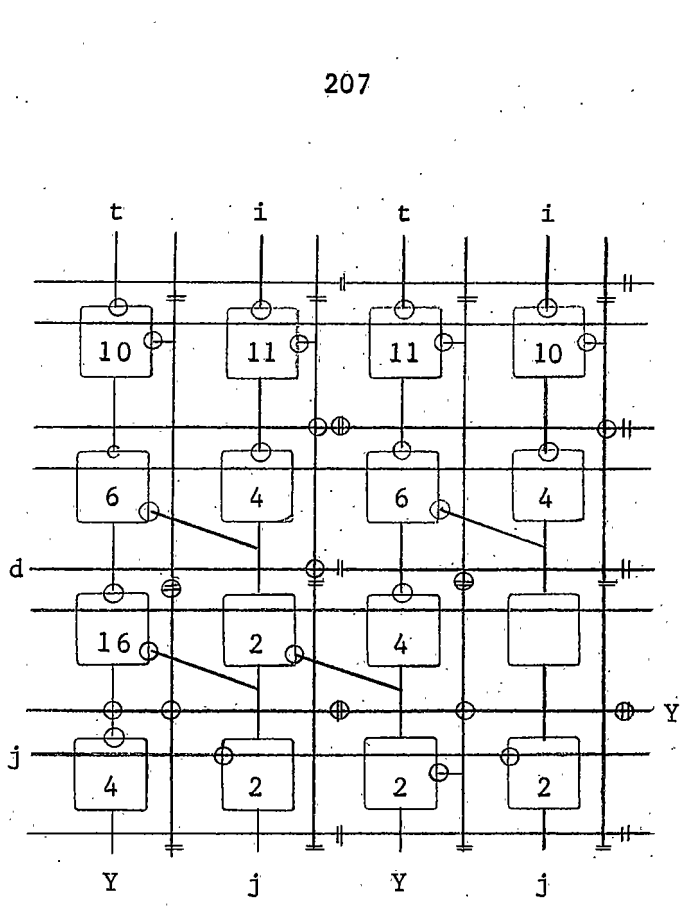


Figure 69. Realization of DTFF10.

DTFF11 differs from DTFF10 only in that t is restricted to come in from the right on Hbuss one. This is compatible with the algorithm. It first puts t in the y input, from the Hbuss, and then it proceeds as above. Figure 70 is the realization of DTFF11.

The mantissa of the product register is made up of seven DSRFF12 cells. Figure 71 is a realization for the cell. The equations are written:

$$R = d'i\times cr\times +$$

$$S = di\times cs\times +$$

The constraints are:

1 USE EXACTLY 4 ROWS

2 USE EXACTLY 4 COLUMNS

3 RESERVE HBUSS 5

4 DEMAND INTERNAL FUNCTION Y BELOW s

5 DEMAND INTERNAL FUNCTION Y' BELOW r

6 DEMAND INTERNAL FUNCTION Y OUTPUT HBUSS 3 LEFT

7 DEMAND INTERNAL FUNCTION Y OUTPUT HBUSS RIGHT

8 DEMAND r,s INPUT ONLY TOP

9 DEMAND c INPUT ONLY RIGHT

10 DEMAND i INPUT ONLY RIGHT

In this case, Hbuss one is available since DTFF9 doesn't use it. If it is used, Hbuss five must be reserved since they overlap.

DSRFF13 differs from DSRFF12 only in that cell(1,2) is set to function 2 in order to have the r input always 1, as shown in Figure 49.

DFF4 is a D flip-flop. The realization is Figure 72. The constraints are:

1 USE EXACTLY 4 ROWS

2 USE EXACTLY 4 COLUMNS

3 DEMAND d INPUT HBUSS 3 LEFT

4 DEMAND INTERNAL FUNCTION Y OUTPUT HBUSS 4 LEFT AND RIGHT

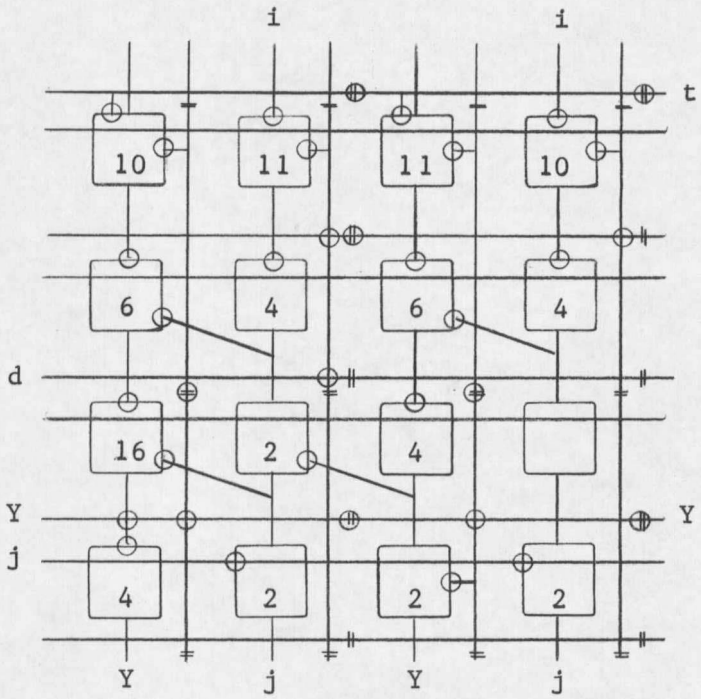


Figure 70. Realization of DTFF11.

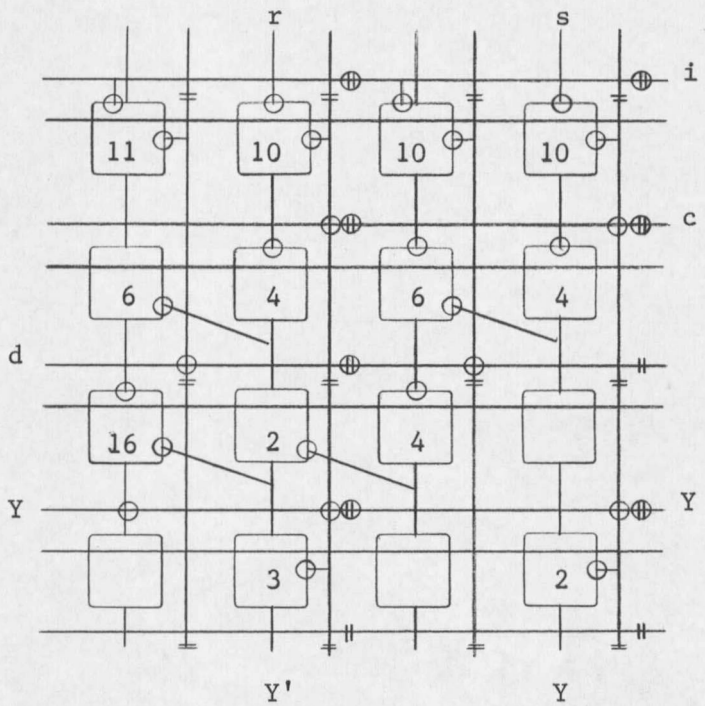


Figure 71. Realization of DSRFF12.

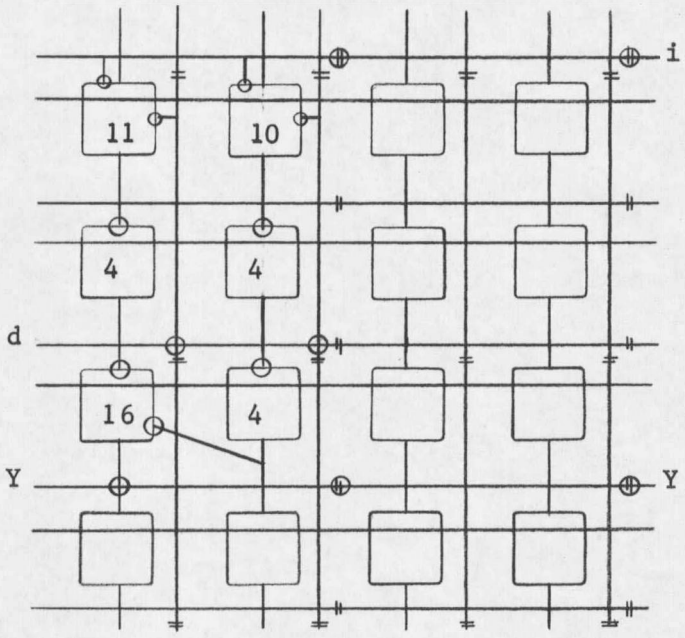


Figure 72. Realization of DFF4.

Next, it is well to return to LOGICA and see what demands the array on the right places upon it. Xbusses two and three are the only additions from Figure 63 and these were left free. Thus LOGICA can be redesigned by constraining Xbusses two and four to carry the required variables.

There are two considerations left. The fifth cell from the right in the product register of Figure 49 has not been synthesized. There are also some normalize logic control functions to be synthesized. A decision must be made at this point. One could synthesize the product register cell which will be called DTFF12 in the two columns to the right of DTFF9-11. This would add at least 24 cells to the array. The remaining cells could be used to synthesize the normalize logic. Alternately the group of unused cells in the lower left corner of the array (Figure 59) could be used for both tasks. In this case, external wires could be used to make the appropriate connections, or a row or two and a column or two could be added to do the bussing. To illustrate, the external wiring method will be chosen here.

The block DTFF12 is set aside for realization of the product register cell. The cell is not really a D-T flip-flop but can be characterized by:

s = Y't

r = Yt+i

Sufficient constraints are:

1 USE 4 COLUMNS

2 USE 4 ROWS

3 DEMAND i,t INPUT LEFT

4 DEMAND INTERNAL FUNCTION Y OUTPUT LEFT

The realization is Figure 73. The points marked S in Figure 59 are wired externally together to supply the trigger input.

For the normalizing logic two blocks are used. First, a one-cell block is used to form Q which is then externally wired to the other Q points. Second, Figure 74 shows a realization of LOGICB based on the equations:

$$f_1 = c_2c_1c_0{\times\times}c_2'c_1'c_0'{\times\times}+$$

$$f_2 = f_1{\times\times},$$

where the c_i are Cannon's c_i which appear in Figure 49. The constraints are:

1 USE EXACTLY 4 COLUMNS

2 USE EXACTLY 4 ROWS

3 RESERVE XBUSSES 2,3

4 DEMAND c_0, c_1, c_2, x INPUT ONLY LEFT

5 DEMAND f_1 OUTPUT LEFT

6 DEMAND f_2 OUTPUT BOTTOM

The c_i and R are connected externally.

The final cellular realization of the Cannon multiplier has 768 cells. It has 16 columns and 48 rows. Logically used cells number 282 for 37 per cent. Cells used for bussing number 257 for 34 per cent. Thus 29 per cent of the cells are unused. Figure 75 illustrates the array. The only problem available for comparison which is anywhere equal in magnitude is Minnick's serial multiplier[18]. He used 432 cells for a 12 bit adder with no constraints upon location of inputs and outputs. Logical use was 55 per cent, jumper use (bussing) was 26 per cent and 19 per cent were not used. It appears that the constraints imposed by parallel I/O and locations of I/O lines cause the efficiency of the realization of Cannon's multiplier to be less than that of the serial multiplier. No comparison is available on the amount of man hours needed to solve the problems.

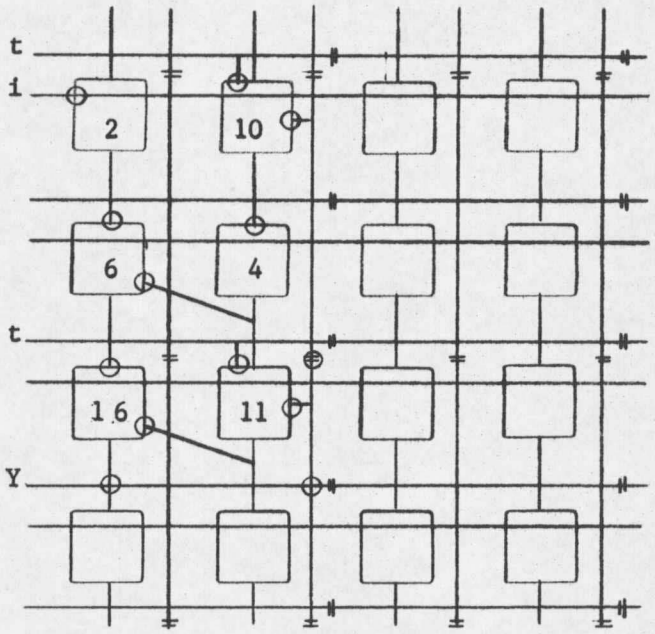


Figure 73. Realization of DTFF12.

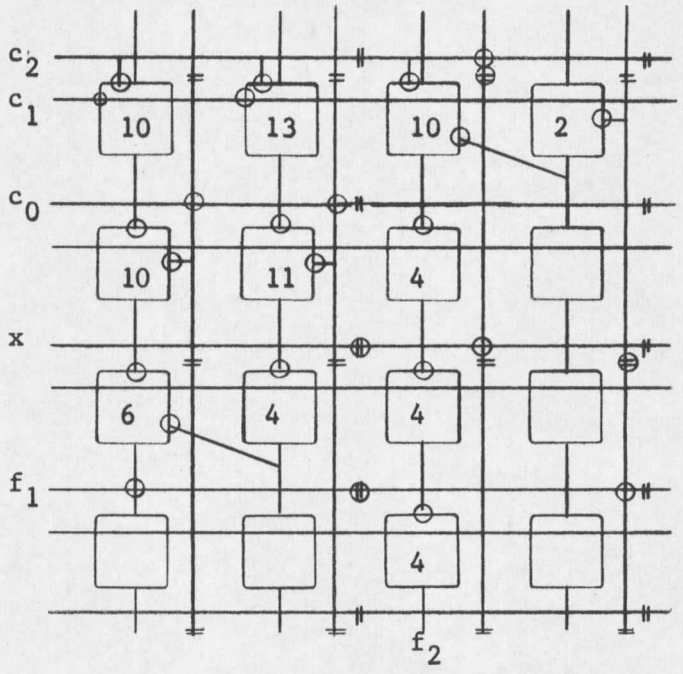


Figure 74. Realization of LOGICB.

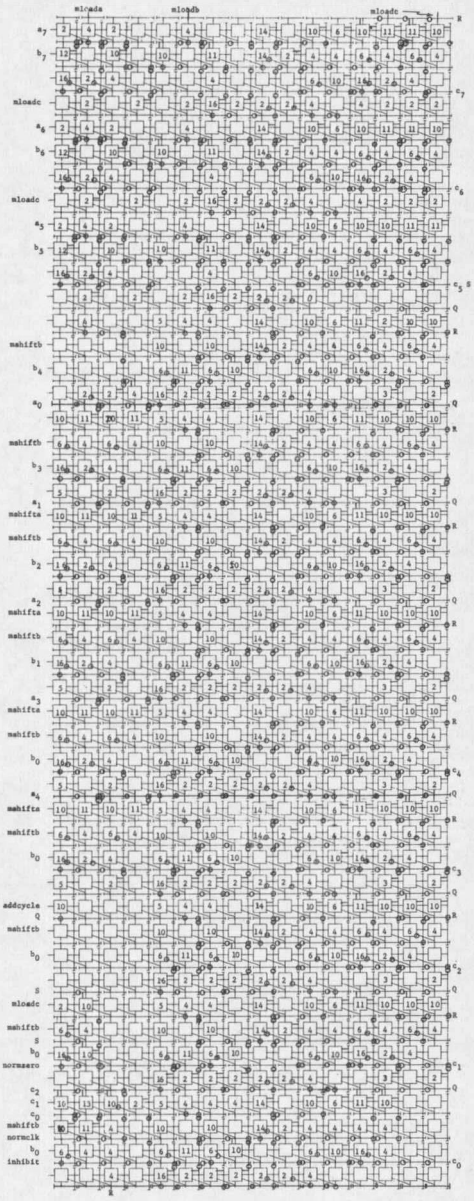


Figure 75. Cellular Realization of Cannon's Parallel Multiplier.

III.4.8 An Interactive, Heuristic Approach to Buss Array Synthesis.

The algorithm presented in the last sections for synthesizing buss arrays makes no uses of the designer's intuition and skill at designing arrays. There is good reason to believe that in many cases a better realization can be had by combining the computation and searching ability of an algorithm with shortcuts and alternate paths that can be produced by the designer.

For example, Figure 76 illustrates the style 4 algorithmic realization of the function of Example 4 (section III.3.1). The realization has five rows and six columns, for a total of 30 cells. The cells used logically number 18, those used as busses number 7, and 5 are unused. Figure 45 has already shown a four by five realization, again with 18 logically used cells, for a savings of one third on the number of cells in the array.

The algorithm may be modified to effect savings in cells. Figure 77 illustrates a possible realization gained by altering the SETCEL portion of the algorithm. Specifically, the collection of cascades has been modified. The first two cascades are collected conveniently in cell(4,1) and the output is bussed over Hbuss five to the lower right corner of the array. There it is collected along with the other two level-one substrings. The cells are specified as 18 logical, three buss, and three unused--for a total of 24 cells.

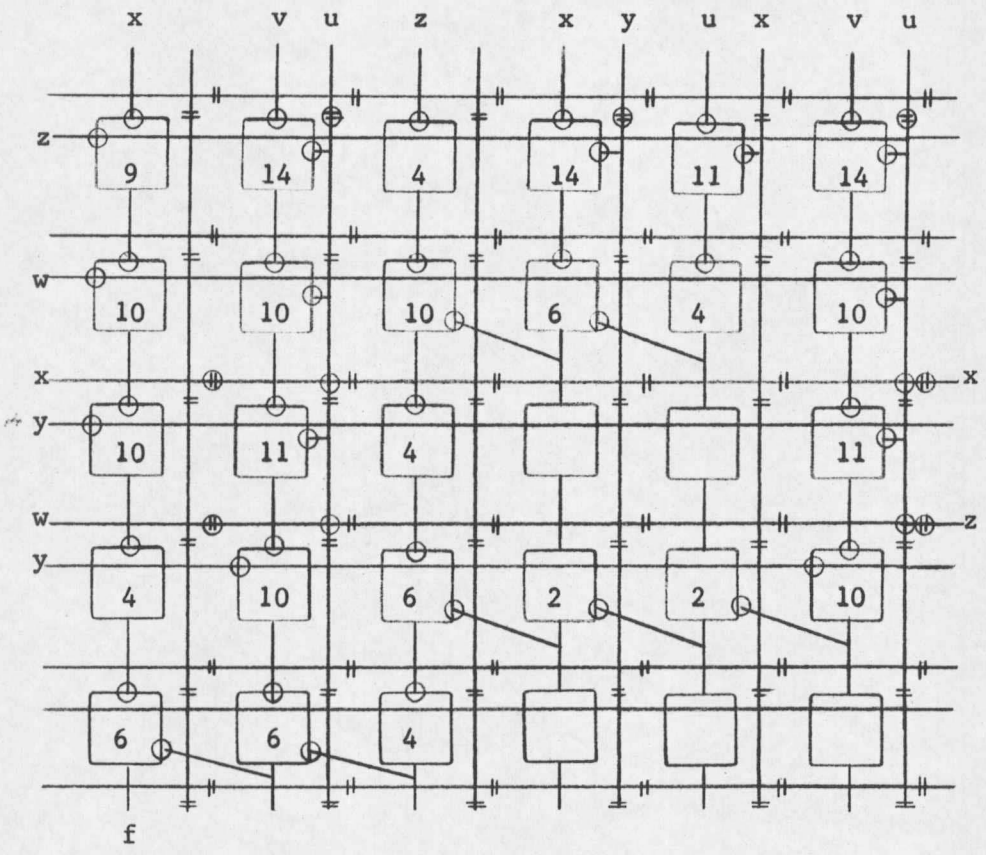


Figure 76. Algorithmic Realization of Example 4.

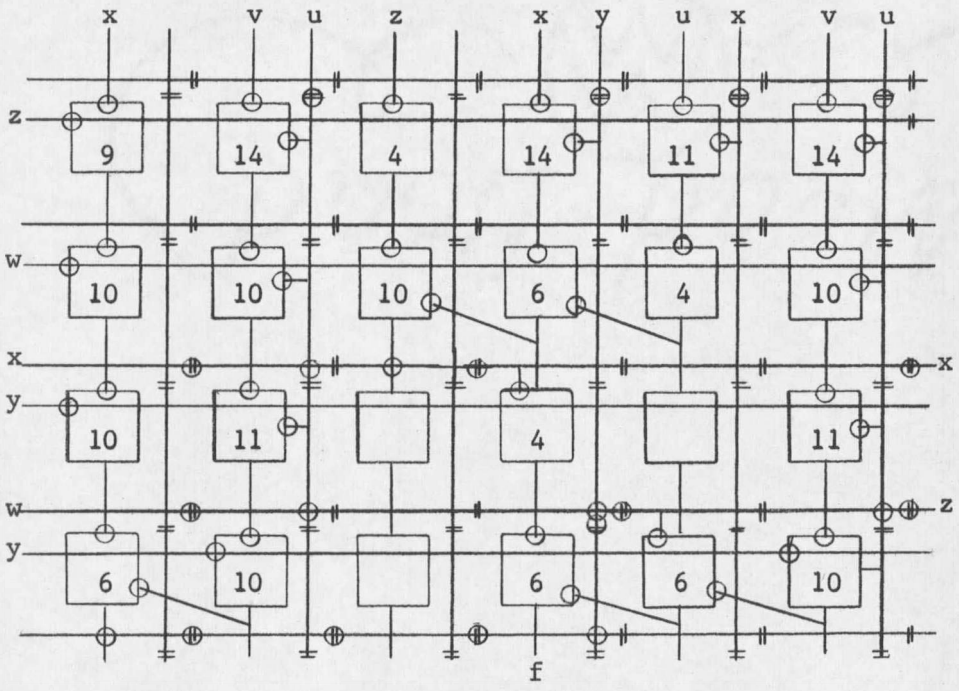


Figure 77. A Fourth Realization of Example 4.

Figure 78 shows yet another realization in which the location of cells used for embedding the substrings (LOCCEL), as well as the method of collection has been altered. The cell in which the last variable of the last substring is embedded has been altered from cell(4,6) to cell(3,5). This produces no saving of cells, but the remaining cells are clustered together in the lower right corner of the array and might be used for some additional function.

Another realization of Example 4 is contained in Figure 79. It has 25 cells with 18 used logically, 4 used as bussed, and 3 left unused. This realization is obtained by altering the order in which the substrings are realized, and where they are realized. In section III.2 it was mentioned that when a simple string was encountered it sometimes was efficient to realize it not in a separate column, but to attach it to another column. In this algorithm, a similar option is available. In this case, the substring zyxθx'u$\times$+$\times$ was embedded by creating cascades in new columns for yxθ and x'u$\times$, uniting them (as in the normal procedure) and then attaching z. This differs in that the original algorithm requires a separate cascade (hence another column in the array) for z. This is a modification of GETLIT, resulting in Figure 79.

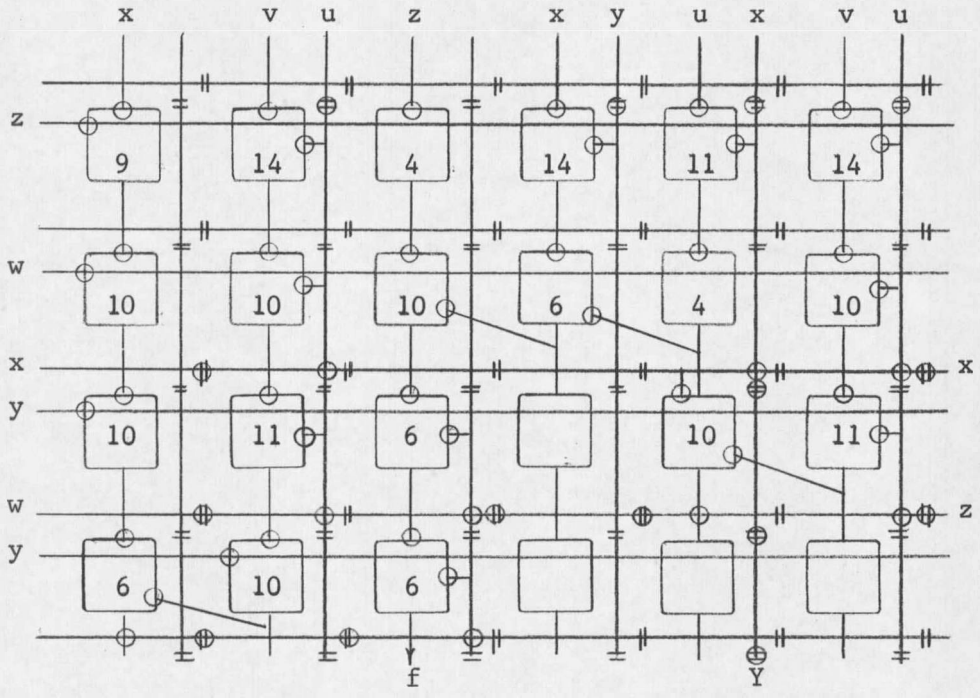


Figure 78. A Fifth Realization of Example 4.

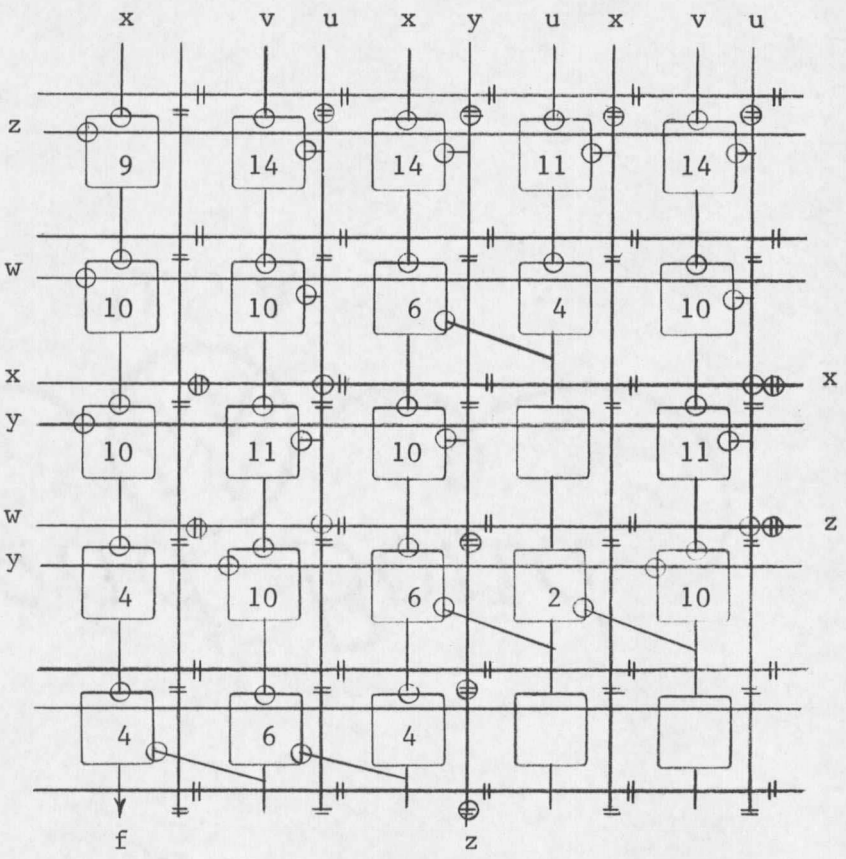


Figure 79. A Sixth Realization of Example 4.

If, in addition to the change in the order of realization of substrings, the method of collection is modified, Figure 80 results. In this case, as in Figure 77, the first two cascades have been united in the most accessible location, cell(4,1), and the output bussed to the right. In Figure 80, 18 cells are used logically while 2 are used to buss information.

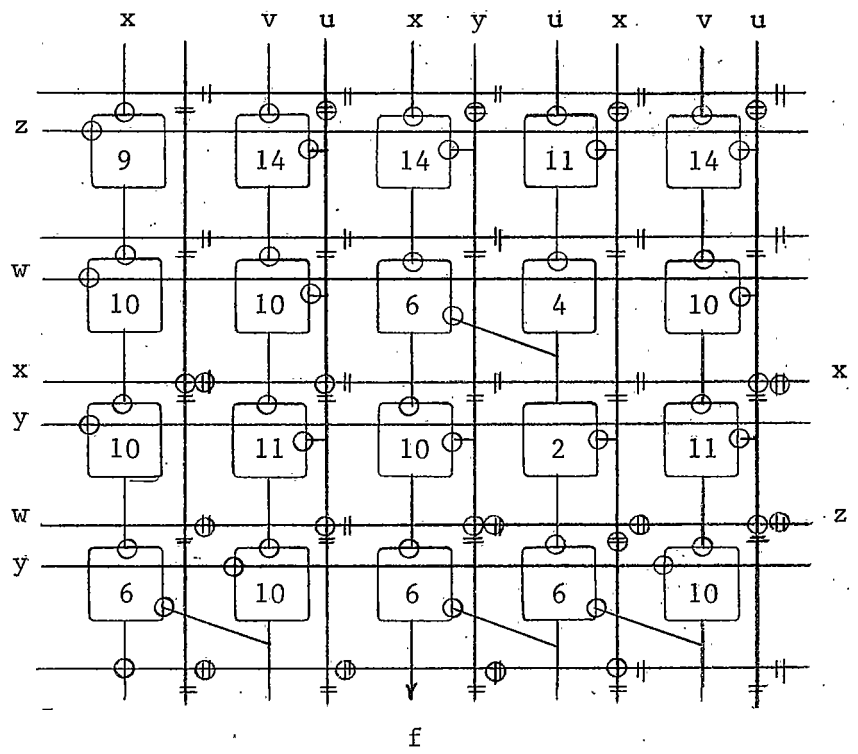


Figure 80. Final Realization of Example 4.

For the given representation of f, Figures 46 and 80 represent just about as good a realization as can be made. There are 18 (dyadic) operators in the function, requiring 18 cells for logical use. The only rectangular possibilities with 18 cells are 6 by 3 and 9 by 2. The author has been unable to produce such an array for this example. One can note that a measure of the efficiency of the realization is the number of logical cells versus the total number of cells. In this case, Figure 80 shows an array which is 90 per cent efficient.

The above illustration of algorithm modification dealt with an unconstrained realization of a function. Similar savings may sometimes be obtained during the synthesis of functions subject to constraints. For example, Figure 81 shows an alternate array for DFF4. (See Figure 72 for the algorithmic version.) This realization uses five cells whereas the old one needs six, a saving of 17 per cent. The saving is made by allowing the OUTPUT procedure to use cell(3,1) to buss the internal function Y onto Hbuss four. This could be extended allowing the use of two or more cells in a column to bring a function to a lower point in the array. In general, this is wasteful of cells, but in this case it saved one.

The modifications which have been discussed are typical of those which could be made in an interactive manner with the algorithm. At

any point in the synthesis procedure the designer may heuristically intervene and modify a step or steps, allowing the algorithmic procedure to be followed afterward. The point of the intervention will normally be either to save cells, or to guide the algorithm to clustering unused cells.

The designer will often be making use of the human ability to form a gestalt when he interacts with the algorithm. Many of the alternate paths he may determine would never be found by the blind algorithm. In other cases such as the one in Figure 81, he will be applying an alternate which often would be wasteful of cells. At some point in the realization, the designer may become aware of the problem causing repeated backups in the algorithm. For example, the trouble may lie in the inability to buss a certain variable into a certain area because all the Vbusses into that area were previously used. He may then alter the realization to free a Vbuss for use. This might be done by making use of some isolated cells to do buss work instead of a neighboring Vbuss.

The possibilities for modification are exceedingly numerous. However, in Table 16, the author has listed the ones which have occurred during synthesis of the functions that he has considered. These alternatives should give the designer some feel for the choices that he may have.

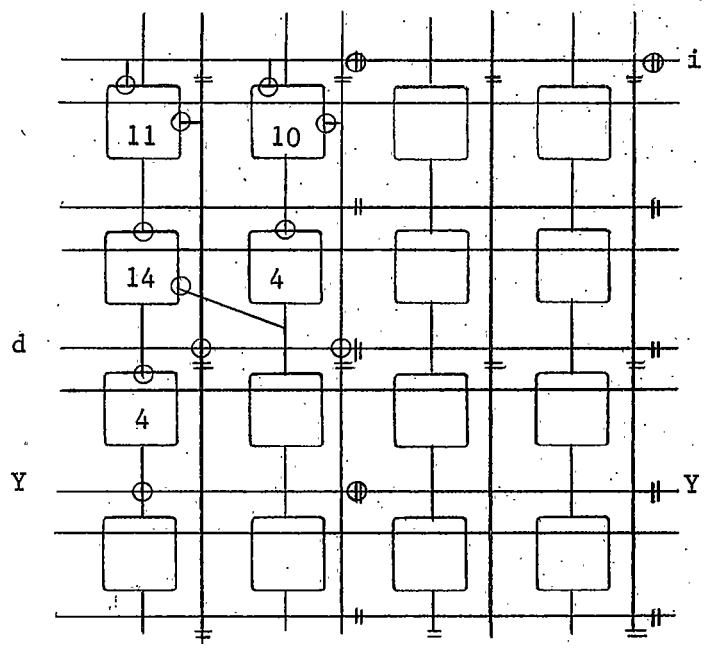


Figure 81. Modified Realization of DFF4

TABLE 16. BUSS SYNTHESIS ALGORITHM MODIFICATIONS.

#	Modification	Comment
1	Fit in vacant cells	The best cell to be next used in creating a function may not be the one specified by LOCCEL. The improvement between Figure 77 and Figure 78 is such an example.
2	Start function in used column	Suppose that only cell(4,3) is used as an indirect input in column three in realizing a cascade in column two. Suppose that cab++ is the next elementary string. Then it can be realized in the top two cells of column three instead of using column four as the algorithm indicates.
3	Route with wobbly paths	The route algorithm does not allow paths which move away from the destination at any point. One can conceive of a buss path that wanders all over the array, though usually this wastes precious bussing power.
4	Try indirect input as a last resort	In some cases, the variable can more profitably be put into the y input of a cell than through another cell indirectly into the x input.
5	Try additional indirect cells	The algorithm allows a search through four cells for an indirect output. Rarely, an additional cell or two may profitably be used for bussing an indirect input.

TABLE 16. Continued.

#	Modification	Comment
6	Use busses first for the set and reset equations	The algorithm assumes that the reset equation will be realized reasonably near (to the right and above) the flip-flop cell. This is not always true. It may save cells to use as a first try Vbusses and Hbusses.
7	Realize simple substrings after the elementary ones associated with them.	As shown in Figure 79, this often will effect a savings of a column.
8	Collect in convenient cells	The example of Figures 76-80 illustrates the saving of this modification.

III.4.9 Summary

In section III.4 buss arrays were introduced, an algorithm based on P-notation developed for synthesizing arbitrary Boolean functions in them, examples were constructed and an interactive, heuristic approach to the synthesis was explained. This method of synthesis is powerful enough to deal with constrained arrays, iterated problems, and register logic. The method comprises a workable engineering approach to cellular synthesis.

CHAPTER IV

SUMMARY, CONCLUSIONS, AND RECOMMENDATION FOR FUTURE RESEARCH

IV.1 Summary

In the first portion of this thesis (sections III.2-III.3) a method of synthesis of arbitrary Boolean functions in a cellular array called style 1 was developed. The style 1 array is similar to Minnick's type A cobweb array except that the cells are allowed to produce any of the 16 functions of two variables. The synthesis method was based upon P-notation, a notational scheme in which the operator is written immediately after its arguments instead of between them. A correspondence was determined between arbitrary Boolean functions written in P-notation and style 1 arrays. Procedures were developed to translate functions into P-notation, and to manipulate the P-notation strings into desirable forms. A form was considered more desirable than another when it produced a realization in a smaller array. Some modifications of the array which can be made by the designer after the function is embedded were discussed. These modifications can, upon occasion, result in a saving in the number of cells used.

The manipulations of P-notation representations of functions were presented in a heuristic, rather than an algorithmic system. The expertise of the designer is called upon to choose his way to the best solution he can find. Some guidelines for these choices were advanced.

The style 1 array was shown to be inadequate for the class of problems labeled iterated problems. The iterated approach is being used when the designer develops an array for a set of functions and then places these arrays side by side making a larger array, with I/O connections between the similar arrays, to synthesize a solution to a larger problem. An example of this method is the synthesis of an n-bit adder. One may synthesize n one-bit full adders and place them side by side in an array, appropriately connecting the carry out and carry in lines.

In the second portion of this thesis (section III.4), the family of buss arrays was introduced and an algorithmic method of synthesizing arbitrary Boolean functions within them was developed. The algorithm is guided by the P-notation form of the function given to it, which should be in the best form possible as determined by the principles of the first portion of this thesis. This family of arrays was shown to have sufficient flexibility for iterated problems.

The algorithm performs a series of search and embedding procedures. It locates cells, determines each succeeding variable to be embedded, finds a way to enter the variable in the cell, specifies the cells used, locates flip-flop elements, and routes outputs to the appropriate locations. Locations of the inputs and outputs may be determined by a set of constraints imposed by the designer. If no constraint is given on a particular input or output, a default case is determined by the algorithm.

A heuristic method of interaction with the algorithm was developed. Examples were given which illustrated how savings in the number of cells required could be effected by modifications to a step of the algorithm determined by the designer. An engineering example, the parallel cellular multiplier, was developed to illustrate the working of the algorithm, and modifications of some of the cells were produced to illustrate the savings of the heuristic, interactive approach.

IV.2 Conclusions

In this thesis was presented a workable engineering approach to the embedding of arbitrary Boolean functions in cellular arrays. The approach was heuristic and interactive in nature, calling for the combination of the straightforward abilities of algorithm and the subtle intuitive knowledge of a human. This should be a valuable design tool in the near future as the technology for fabricating large scale integrated circuits is rapidly approaching a sufficient maturity to enable such circuits to find a prominent place in digital equipment.

The generality of these arrays, leading to excessive amounts of computation in an algorithmic synthesis procedure seem to demand, in the author's opinion, a heuristic and interactive approach such as is developed in this thesis. The inability to guarantee a minimal solution does not appear to be serious. The large number of gates available allows the designer to waste some of them. Additionally, this method easily avoids faulty gates within a wafer, both logically, and with burn-out fuses or some other specifying mechanism, faults in the physical array.

IV.3 Future Research

The method of cellular synthesis developed in this thesis is amenable to hand computation. However, for significant problems the amount of labor involved will become quite large. The existence of numerous paths of synthesis that may be traced by the designer in his search for an efficient realization suggests strongly that the system be implemented upon a computer.

A further argument for computerization is the large amount of data that must be stored and manipulated. It is essential that the designer be able to view the arrays at various stages of synthesis, and the iterative process will neccessitate changing the appearance of P-strings and arrays many times. It will also be necessary to recall past stages of solution in order to make sound judgements about succeeding stages of the solution. A computer is very adept at storing and recalling information of this type.

In particular, it is suggested that the computer system used be a graphics system. It does not appear to be feasable to display arrays on a teletype or line printer in the interactive environment. The visual display of a cathode ray tube is quite capable of this feat. Additionally, a graphics input device such as a light pen or tablet may be used to quickly select procedures, determine the nature

of arrays on the display, and alter the array or P-string displayed. For example, one might desire to determine the buss segments which have been specified to a particular variable. The input device could be used to point to a buss segment used by the variable and the computer instructed to enhance all segments connected directly to it.

The implementation of this system must be flexible. The synthesis method developed in this thesis is not presented as a closed set. Indeed, it is the nature of heuristic approaches to be open-ended. The designer will want to add and delete procedures and modify the algorithms as experience is gained with problems. New classes of problems will most probably arise, and the structure should be sufficiently flexible to deal with them.

Implementation of the synthesis system on a computer should largely consist of defining a high level language. The language would be capable of controlling the data displayed and entered, calling the procedures and algorithms, and allowing the designer to interact with the system by modifying P-strings, arrays, and procedures. The author presently conceives of the language as a resident program performing the executive functions, and a series of subroutines which would perform the manipulations.

The executive program could itself exist in several levels. For example, the top level might consist of several commands to set the mode of the system. Such might be:

BEGIN NEW PROBLEM

DEFINE EQUATIONS

DEFINE CONSTRAINTS

MANIPULATE EQUATIONS

DO BUSS SYNTHESIS

SAVE SOLUTION UNDER name

PLOT SOLUTION name

At the next level could exist the commands which detail what can be done for each of the above listed commands. For example, under DEFINE CONSTRAINTS would exist a series of commands similar to those in Figure 50. These would allow the user to define the constraints under which a buss array would be synthesized. For another example, the commands under MANIPULATE EQUATIONS would be calls to subroutines implementing the manipulative procedures developed in this thesis (Table 10) plus a procedure for modifying, inserting and deleting other procedures.

Each of the commands would carry with it a result to be presented on the graphics output device. For example, inherent with each P-string is a style 1 array. It would be desirable to

view this array, and further to be able to modify it directly, with the P-string being modified accordingly by the computer. Thus the designer could work both with the P-string and its array, modifying either to produce a better realization. During buss synthesis each stage of the synthesis carries with it a partially specified array. If this array were displayed and commands made available, the designer could modify the array or the course of the algorithm according to his experience with the design of arrays.

It is feasable to implement the system outlined above on present day graphics computer systems. With such a tool at the disposal of the designer, efficient design of cellular arrays for arbitrary Boolean functions should be speedily accomplished.

BIBLIOGRAPHY

BIBLIOGRAPHY

1 Bowden, B. V., editor, Faster Than Thought, Sir Isaac Pitman and Sons, Ltd., London, 1953, 416 pp.

2 Brooking, M. E.; King, W. F., III; and Giusti, A., A Survey of In-house Work on Cellular Logic, Unpublished memorandum, Air Foce Cambridge Research Laboratory, 1965.

3 Cannon, Lynn Elliot, A Cellular Computer to Implement the Kalman Filter Algorithm, Unpublished doctor's thesis, Montana State University, Bozeman, Montana 1969, 217 pp.

4 Davis, Martin, Computability and Unsolvability, McGraw-Hill, Inc., 1958, 210 pp.

5 Ernst, George W., and Newell, Allen, GPS: A Case Study in Generality and Problem Solving, Academic Press, Inc., New York, 1969, 297 pp.

6 Hamblin, C. L., "Translation to and From Polish Notation," Computer Journal, 5:210-213, 1962.

7 Kautz, W. H., "Testing for Faults in Combinational Cellular Logic Arrays," 1967 Switching and Automata Theory Symposium.

8 Kautz, W. H., "Diagnosis and Testing of Cellular Arrays," Properties of Cellular Arrays for Logic and Storage, Stanford Research Institute Project 5876, Scientific Report 3, Menlo Park, California, July, 1967.

9 Kautz, W. H., "Fault Testing and Diagnosis in Combinational Digital Circuits," IEEE Transactions on Electronic Computers, C-17:352-366, April, 1968.

10 Khambata, Adi J., Introduction to Large-Scale Integration, Wiley-Interscience, New York, 1969, 190 pp.

11 King, W. F., and Giusti, A., "Can Logic Arrays be Kept Flexible?" Electronic Design, 13:57-61, May 24, 1965.

12 Levy, S.; Winder, R. O.; and Mott, T. H., Jr., "A Note on Tributary Switching Networks," IEEE Transactions on Electronic Computers, EC-13:148-151, April, 1964.

13 Maitra, K. K., "Cascaded Switching Networks of Two-Input Flexible Cells," IRE Transactions on Electronic Computers, EC-11:136-143, April, 1962.

14 McCluskey, E. J., Introduction to the Theory of Switching Circuits, McGraw-Hill Co., New York, 1965, 318 pp.

15 Miller, R. E., Switching Theory: Combinational Circuits, John Wiley and Sons, Inc., New York, 1:351, 1965.

16 Minnick, R. C., "Cutpoint Cellular Logic," IEEE Transactions on Electronic Computers, EC-13:685-698, December, 1964.

17 Minnick, R. C., "Application of Cellular Logic to the Design of Monolithic Digital Systems," Proceedings of the Symposium on Microelectronics and Large Systems, Spartan Books, Washington, 1965.

18 Minnick, R. C., "Cobweb Cellular Arrays," Proceedings of the AFIPS Fall Joint Computer Conference, 1965, 27:327-341, Spartan Books, Washington, 1965.

19 Minnick, R. C., "Survey of Microcellular Research," Journal of the Association for Computing Machinery, 14:203-241, April, 1967.

20 Minnick, R. C., "Cellular Networks," in Network and Switching Theory: A NATO Advanced Study Institute, 497-520 pp., edited by Guiseppe Biorci, Academic Press, Inc., New York, 1968.

21 Minnick, R. C., and Short, R. A., Cellular Linear-Input Logic, Final Report, Stanford Research Institute Project 4122, Contract AF 19(628)-498, Menlo Park, California, (available from DDC, No. AD-433802).

22 Minnick, R. C., and others, Cellular Logic, Final Report, Stanford Research Institute Project 5087, Contract AF 19(628)-4233, Menlo Park, California, (available from DDC, No. AD-643178).

23 Newell, Allen, and Ernst, George W., "The Search for Generality," in Proceedings of IFIP Congress '65, vol. 1, 17-24 pp., edited by Wayne A. Kalenich, Spartan Books, Inc., Washington, 1965.

24 Parkin, Sandra, APL 360 Reference Manual, Science Research Associates, (subsidiary of IBM), 1968.

25 Quine, W. V., "The Problem of Symplifying Truth Functions," American Mathematical Monthly, 59:521-531, October, 1952.

26 Quine, W. V., "A Way to Simplify Truth Functions," American Mathematical Monthly, 62:627-631, November, 1955.

27 Randell, B., and Russel, L. J., Algol 60 Implementation, Academic Press, New York, 1964, 300 pp.

28 Sklansky, J., "General Synthesis of Tributary Switching Networks," IEEE Transactions on Electronic Computers, EC-12:464-469, October, 1963.

29 Spandorfer, L. M., and Murphy, J. V., Synthesis of Logic Functions on an Array of Integrated Circuits, Scientific Report No. 1 for UNIVAC Project 4645, Contract AF 19(628)-2907 Sperry Rand Corporation, UNIVAC Engineering Center, Blueke 11, Pa., October, 1963.

30 Stone, H. S., and Korenjak, A. J., "Canonical Form and Synthesis of Cellular Cascades," IEEE Transactions on Electronic Computers, EC-14:852-862, December, 1965.

31 Thurber, Kenneth James, Fault Location in Cellular Arrays, Unpublished doctor's thesis, Montana State University, Bozeman, Montana, 1969, 118 pp.

32 Thurber, Kenneth James, "Fault Location in Cellular Arrays," in Proceedings of the Fall Joint Computer Conference '69, vol. 35, 81-88 pp., Spartan Books, Inc., Washington, 1969.

33 Weiss, C. Dennis, "The Characterization and Properties of Cascade Realizable Switching Functions," IEEE Transactions on Computers, C-18:624-633, July, 1969.

34 Weiss, C. Dennis, "Optimal Synthesis of Arbitrary Switching Functions with Regular Arrays of 2-Input 1-Output Switching Elements," IEEE Transactions on Computers, C-18:839-856, September, 1969.

MONTANA STATE UNIVERSITY LIBRARIES
arch D378.B453
Heuristic interactive synthesis of cellu
RL
3 1762 00185948 5

[redacted]
D378
B453
cop.2
Berges, G. A.
Heuristic interactive synthesis of cellular arrays, based on Polish notation.
NAME AND ADDRESS
D378
B453
cop. 2
THES[redacted]