SEL-73-030

Self - Testing Residue Trees

by

Stephen G. Kolupaev

August 1973

Technical Report No. 49

This work was supported by the National Science Foundation under Grant GJ-27527.

DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY · STANFORD, CALIFORNIA

SEL 73-030

SELF-TESTING RESIDUE TREES

by

Stephen G. Kolupaev

August 1973

Technical Report no. 49

DIGITAL SYSTEMS LABORATORY

Dept. of Electrical Engineering Dept. of Computer Science

Stanford University

Stanford, California

This work was supported by the National Science Foundation under grant GJ-27527.

ABSTRACT

Error detection and correction in binary adders often require computing the residue modulo A of a binary number. We present here a totally self-checking network which extracts the residue of a binary input number of arbitrary width, with respect to any odd modulus A. This network has the tree structure commonly used for residue extraction: a binary tree of circuit blocks, where each block outputs the residue of its inputs. The network we describe differs from previous designs in that the signals between blocks of the tree are not binary-coded. Instead, the l-out-of-A code is used, where A is the modulus desired. Use of this code permits the network to be free of inverters, giving it an advantage in speed. The network output is also coded l-out-of-A, and with respect to this code, the residue tree is totally self-checking in the sense of Anderson [3].

The residue tree described here requires logic gates with A inputs, when the modulus desired is A. This makes the basic design somewhat impractical for a large modulus, because gates with large fan-in are undesirable. To extend the usefulness of this network, we present a technique which uses several residue trees of this design, each for a different modulus. The outputs of these residue trees are combined by a totally self-checking translator from the code of multiple residues to the l-out-of-A code. Using this multiple residue scheme, the modulus of each residue tree can be made much smaller than the desired modulus A.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1. INTRODUCTION

The residue check is a widely known method of checking binary addition. The addition $n_1+n_2=sum$ is checked by verifying that $R(n_1) * R(n_2) = R(sum)$, where $R(n)$ is the residue modulo A of n, and * is addition modulo A. As with all error detection methods, the effectiveness of this method is compromised by the probability that the checking circuit fails in such a way that a subsequent error in the addition goes undetected. Ideally, checking circuits should be designed so that any failure causes the checker to signal an error, even when the failure is in the checker itself [I], [2].

The residue tree presented here meets this goal of signalling its own errors. The circuit obtains the residue of its input with respect to a modulus A. The circuit output thus has A distinct states, which we have encoded in the l-out-of-A code. With respect to this code, the residue tree meets the requirements of a totally self-checking [3] network. Assuming only permanent, single stuck-type faults may occur in the network, then

1) If the network has suffered no failures, its output is a code word (one line high, A-l lines low), and is correct.

2) If the network does contain a failure of the type assumed above, then the output is either a code word or it is not. If it is a code word, then it is guaranteed to be the right

code word, in spite of whatever single stuck fault is present. This is called the _fault-secure_ property.

3) If the network contains any particular failure of the type assumed above, we guarantee that at least one particular input, in the presence of that particular failure, will cause the output to be non-code (not l-out-of-A). That is, for the entire network, each and every single stuck-fault has a corresponding set of inputs which reveal the presence of that fault by causing a non-code output. This is the _self-testing_ property.

In the above, 1) defines the normal operation of the network. 2) gives us the confidence that if the network output is a code word, it is the correct code word. Finally, 3) asserts that if we present the network with all of its inputs, and if only code words appear at the network outputs, then the network contains no fault of the type assumed above. Thus this residue tree can be used in a residue checker with this special confidence in its output: provided that only permanent, single, stuck-type faults may arise in the residue tree, the residue tree is fault-free _if and only if_ its output is _always_ a code word.

Wadia [4] has presented a modulo 3 residue tree, which is designed with these self-checking features in mind. His network is "testable", which means that for every single stuck-fault in the network, at least one binary input number will cause an incorrect output. That network is not intended to be totally self-checking by itself. The code used to pass residues between blocks and to display the output residue is the binary code on 2 wires. In his design, all 4 output combinations are valid outputs, and totally self-checking operation results only when the network is made part of a larger network.

The residue network we present here, on the other hand, stands alone as a totally self-checking network. This difference arises directly from choosing the highly redundant l-out-of-A code for block outputs, rather than the non-redundant 2 wire binary code. Using the l-out-of-A code to carry signals between blocks has the immediate disadvantage of requiring more output wires. But, something is gained by paying that price. First, as mentioned above, the residue tree itself is totally self-checking. Second, the design of circuit blocks for any modulus is equally as straightforward as for the lowest modulus, 3. Third, blocks in the residue tree need no inverters, so that high speed can be obtained. Level-merging [2,3] can be applied to this residue tree, resulting in one gate delay per block.

2. BASIC DESIGN FOR ARBITRARY MODULUS A

The totally self-checking residue tree has the basic structure shown in Fig. 1. The common technique of breaking the input number n into several bytes $B_0,B_1,B_2,\ldots$etc. is used. For example, let us assume that there are 4 bytes, as in Fig. 1. Further assume that each byte contains 4 bits. Thus the input number n is a 16-bit

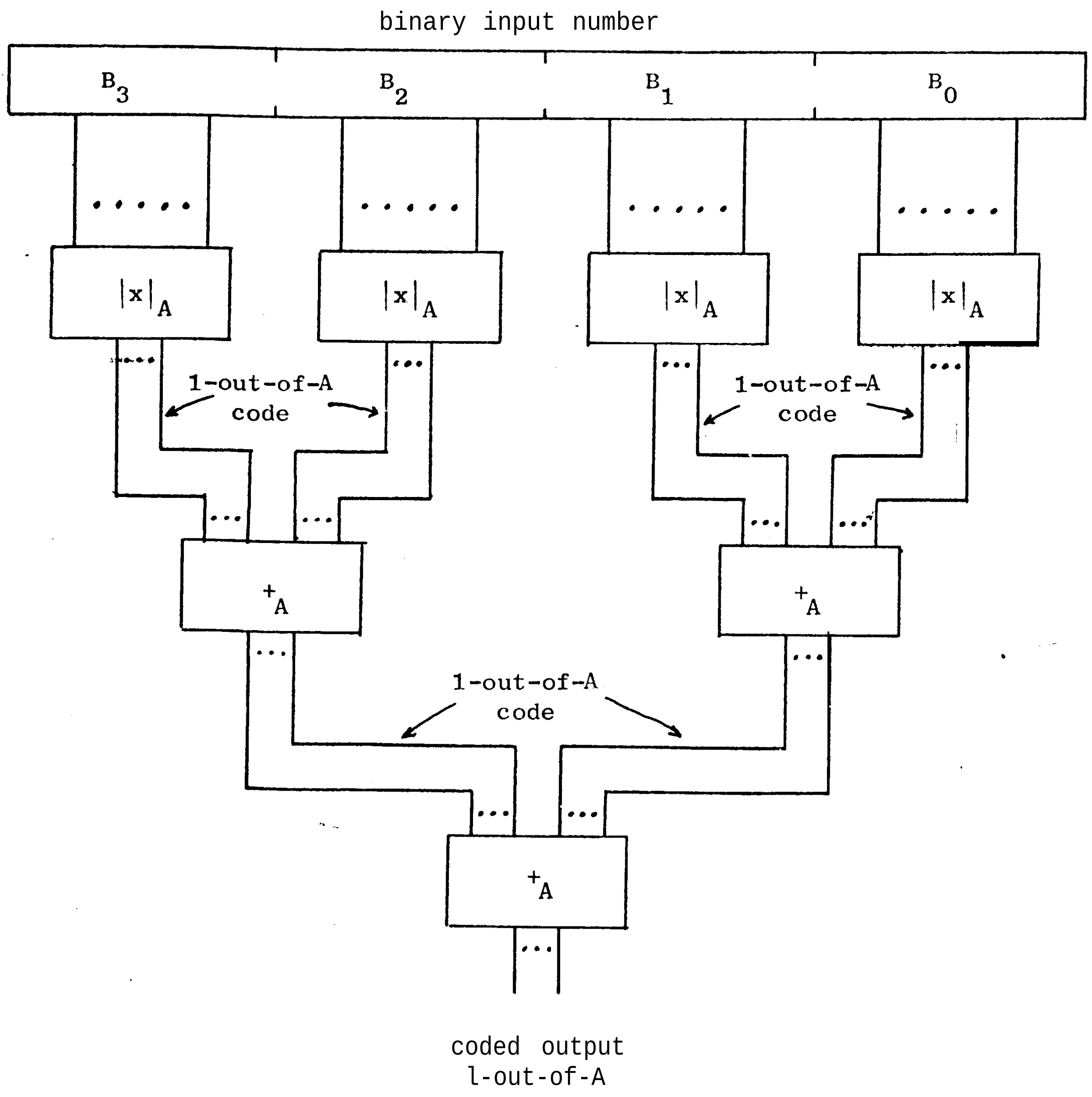


Figure 1. Modulo A residue tree.

binary number. Let each byte have the value seen in the 4 bits which it includes. Thus,

$$0 \le \left(B_0, B_1, B_2, B_3\right)_2 \le 15_{10}$$

The value of the input number is then the weighted sum of the 4 bytes:

$$n = B_3 \cdot 2^{12} + B_2 \cdot 2^8 + B_1 \cdot 2^4 + B_0$$

Because of the homomorphism relating modulo A addition with ordinary addition*, the input number modulo A $|n|_A$ is given by

$$|n|_A = |B_3 \cdot 2^{12}|_A +_A |B_2 \cdot 2^8|_A +_A |B_1 \cdot 2^4|_A +_A |B_0|_A . \qquad (1)$$

Here $+_A$ denotes modulo A addition. Circuit blocks labelled $|x|_A$ in Fig. 1 obtain the residue modulo A of the weighted input bytes. This is the first step in evaluating equation (1) above, and these blocks are the first rank of blocks seen in Fig. 1. Once these residues are obtained, they are added modulo A by circuit blocks labelled $+_A$ in Fig. 1. Because of the associativity of addition modulo A, it is immaterial what order is chosen for the addition. We have chosen the order implicit in Fig. 1, so that all $|x|_A$ output signals pass through the same number of $+_A$ blocks. This makes the number of gate delays from each bit of the input number to the residue tree output equal.

The interblock signals in Fig. 1 and the corresponding partial results of evaluating equation (1) above, each take on the values

0,1,2,...,A-1. In this design, each block has A outputs $<R^0, R^1, R^2, \ldots, R^{A-1}>$. The outputs of all blocks are encoded in the 1-out-of-A code given in the following table:

*A good review of residue arithmetic is given in Chapter 2 of Szabo and Tanaka [5].

Inter block code

residue	R^0	R^1	R^2	R^3	R^{A-2}	R^{A-1}
0	1	0	0	0...	0	0
1	0	1	0	0...	0	0
2	0	0	1	0...	0	0
.	.	.	.	.	.	.
.	.	.	.	.	.	.
A-1	0	0	0	0...	0	1

If the residue tree is free of faults, each block will have exactly one output high. Which output that is determines what residue is being emitted, as

$$R^i = 1 \text{ iff the residue is } i.$$

In order that the residue tree can be totally self-checking, all lines must be tested for stuck-at-0 and stuck-at-1, and hence all lines must at different times carry logic 0 and logic 1. Having chosen the 1-out-of-A code, we are then obliged to have each block output exhibit all A different residues. Thus each byte must contain at least $\lceil \log_2 A \rceil$* bits. Referring to equation (1), a typical $|x|_A$ block performs the following reduction modulo A, where B is the content of the byte, and 2^i is the weight of its least significant bit.

$$|B \cdot 2^i|_A .$$

Because of the homomorphism between modulo A multiplication and ordinary multiplication, we can re-write the above as

$$|B \cdot 2^i|_A = \big|\ |B|_A \cdot |2^i|_A\ \big|_A . \qquad (2)$$

It can be seen in (2) that if a byte B contains fewer than $\lceil \log_2 A \rceil$ bits, the term $|B|_A$ would not take on all A different values.

*$\lceil x \rceil$ is the least integer greater than or equal to x.

Since $|2^i|_A$ is fixed, the product (2) would not, either. Thus the requirement that each byte contain $\lceil \log_2 A \rceil$ bits is certainly a necessary condition for the block output $|x|_A$ to take on all A values of residue. This is not a sufficient condition, as we can show in a simple example.

Suppose we have a byte B containing 3 bits, a modulus $A=6$, and the weight of the byte is 2^3. In the following multiplication table, we show what values the product (2) assumes, when the 3 bits of B take all 8 different combinations.

B	$\|B\|_6$	product $\big\| \|B\|_6 \cdot \|2^3\| \big\|_6$
0	0	0
1	1	2
2	2	4
3	3	0
4	4	2
5	5	4
6	0	0
7	1	2

note: $|2^3|_6=2$

What happened in this example was that the product (2) was always some multiple of the greatest common divisor of A and the byte's weight 2^i, which in this case is GCD(6,8) = 2. In Appendix A, we show that restricting the modulus A to be odd is sufficient for the multiplication . of the integers modulo A (Z_A) by any power of 2 to be an onto map from Z_A to itself. Then for the $|x|_A$

blocks of Fig. 1 to exhibit all A values of residue, we require that each byte contain at least $\lceil \log_2 A \rceil$ bits, and that the modulus A be odd.

Once these restrictions are met, the $|x|_A$ blocks in Fig. 1 will emit all A values of residue. We must also have the assurance that the $+_A$ blocks will do the same. Since the binary input number is assumed to take on all possible values, the bytes of the input number exhibit all possible combinations of residues. Then each $+_A$ block in Fig. 1 receives all A^2 combinations of residues, and its output ranges through all A values of residue, as desired.

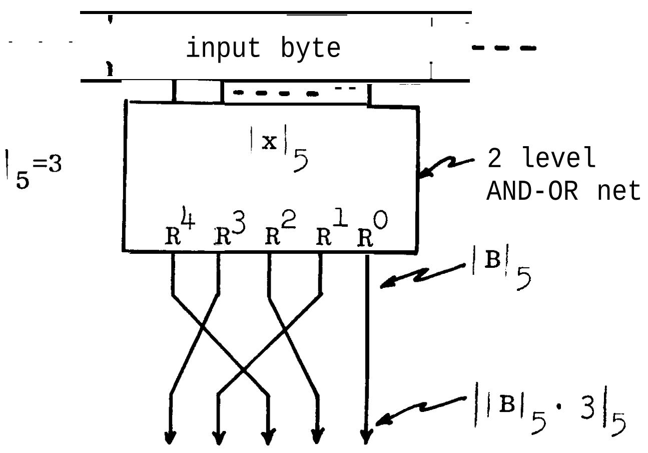
The design of an $|x|_A$ block is quite straightforward. The number seen in its byte is first reduced to its residue (this is the evaluation of $|B|_A$ in equation (2)). That residue is formed in the l-out-of-A code. This reduction is performed by a 2 level AND-OR network. Suppose there are k bits in the byte to be reduced. The network completely decodes k bits to 1 line out of 2^k, -using 2^k k-input AND gates, and then OR's together lines which correspond to the same residue. A simple example of this is shown in Fig. 3 for a 4-bit byte and modulus 3.

Once the reduction $|B|_A$ is done, the multiplication modulo A by the weight of the byte remains. Since the multiplication is a 1-1 onto map from the set of residues $\{0,1,2,\ldots,A-1\}$ to itself, each residue $|B|_A$ is mapped to a different residue by the product $\left| |B|_A \cdot |2^i|_A \right|_A$. No two residues $|B|_A$ map to the same product, because of the odd choice of A.

Then to effect this multiplication, we can simply re-label the A residue lines appropriately, and the multiplication is done without any gates. We shall show this re-labelling by drawing a "network" which is simply a crossover of wires. An example is a byte of 3 bits, with weight 2^3, for modulus 5. This example is more fully developed in Fig. 5.

$\|B\|_5$	$\|B\|_5 \cdot_5 \|8\|_5$
0	0
1	3
2	1
3	4
4	2

note: $|8|_5 = 3$



The design of the $+_A$ block is little more complicated than the $|x|_A$ block. Two l-out-of-A coded residues X and Y are its inputs, $X = \langle X^0, X^1, .., X^{A-1} \rangle$, and $Y = \langle Y^0, Y^1, ..., Y^{A-1} \rangle$. The X lines carry the X residue, and the Y lines carry the Y residue. Line X^i is at logic 1 iff the X residue is i, and similarly for Y^i. The output of the $+_A$ block will be called R, and its lines will be labelled similarly, $R = \langle R^0, R^1, ..., R^{A-1}$. The output R is to be the sum modulo A of the inputs X and Y. The output residue is 0 if $\langle X,Y \rangle = \langle 0,0 \rangle$, or $\langle 1,A-1 \rangle$, or $\langle 2,A-2 \rangle$, etc.

In general, the output residue is i when $\langle X,Y \rangle = \langle 0,i \rangle$, or $\langle 1,i-1 \rangle$, or $\langle 2,i-2 \rangle$, etc. Because of the l-out-of-A encoding of the X and Y inputs, the logical proposition "the X residue is i" is the same as the logical proposition "line X^i is 1." Then the output lines $R^0, R^1, ..., R^{A-1}$ should carry the logic values of the following logical equations

$$R^0 = X^0Y^0 + X^1Y^{A-1} + \ldots + X^{A-1}Y^1$$

$$R^1 = X^0Y^1 + X^1Y^0 + \ldots + X^{A-1}Y^2$$

$$\dot{R}^{A-1} = X^0Y^{A-1} + X^1Y^{A-2} + \ldots + X^{A-1}Y^0$$

The $+_A$ block is designed by simply implementing these equations with logic gates. Each output R^i is realized by an independent 2 level AND-OR network, just as given above. Examples for A=3 and A=5 appear in Figs. 4 and 6.

In the sections which follow, we will show an example of the designs for modulo 3 and for modulo 5 residue trees. Following these examples, we shall prove that this basic design is totally self-checking for single stuck-at faults.

3. SELF-TESTING OPERATION MODULO 3

A modulo 3 residue tree is shown in Fig. 2. The first rank of the tree, i.e., those circuits connected directly to the input number n, is composed of 4 identical circuits called $|x|_3$ blocks.

The first rank circuit reduces a 4 bit weighted byte to its residue modulo 3. Because the byte widths are 4 bits, their weights are 1, 2^4, 2^8, and 2^{12}. But each of these weights has a residue of 1 modulo 3, so the product of byte residue with its weight is the same as the byte residue in all 4 circuits. The weight of each byte can then be ignored. The $|x|_3$ block emits its output residue in the l-out-of-3 code, and its outputs are labelled R^0, R^1, and R^2. R^0 is logic 1 iff the residue of the 4 bit input is 0, that is, the input is the binary encoding of 0, 3, 6, ... 12, or 15. Similarly with outputs R^1 and R^2.

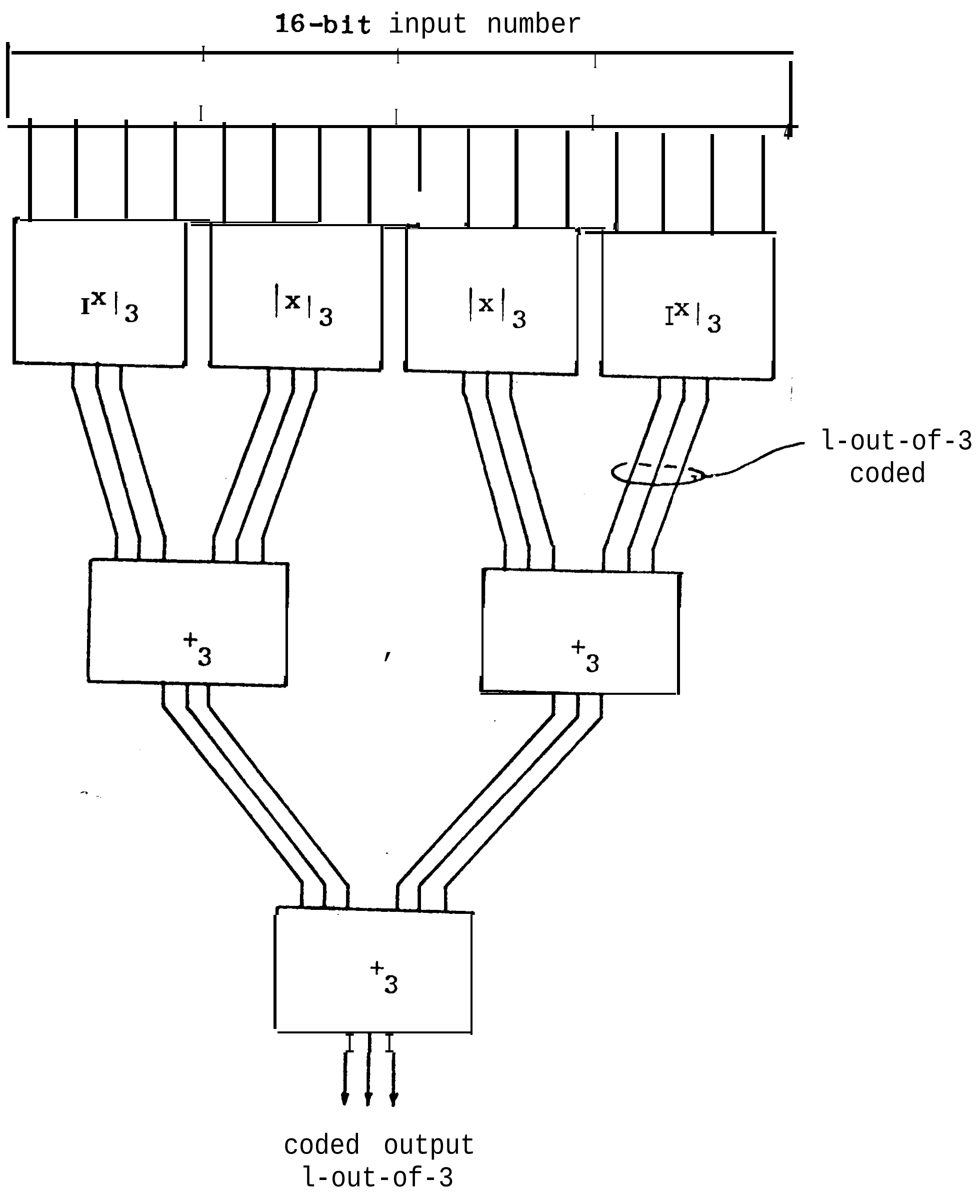
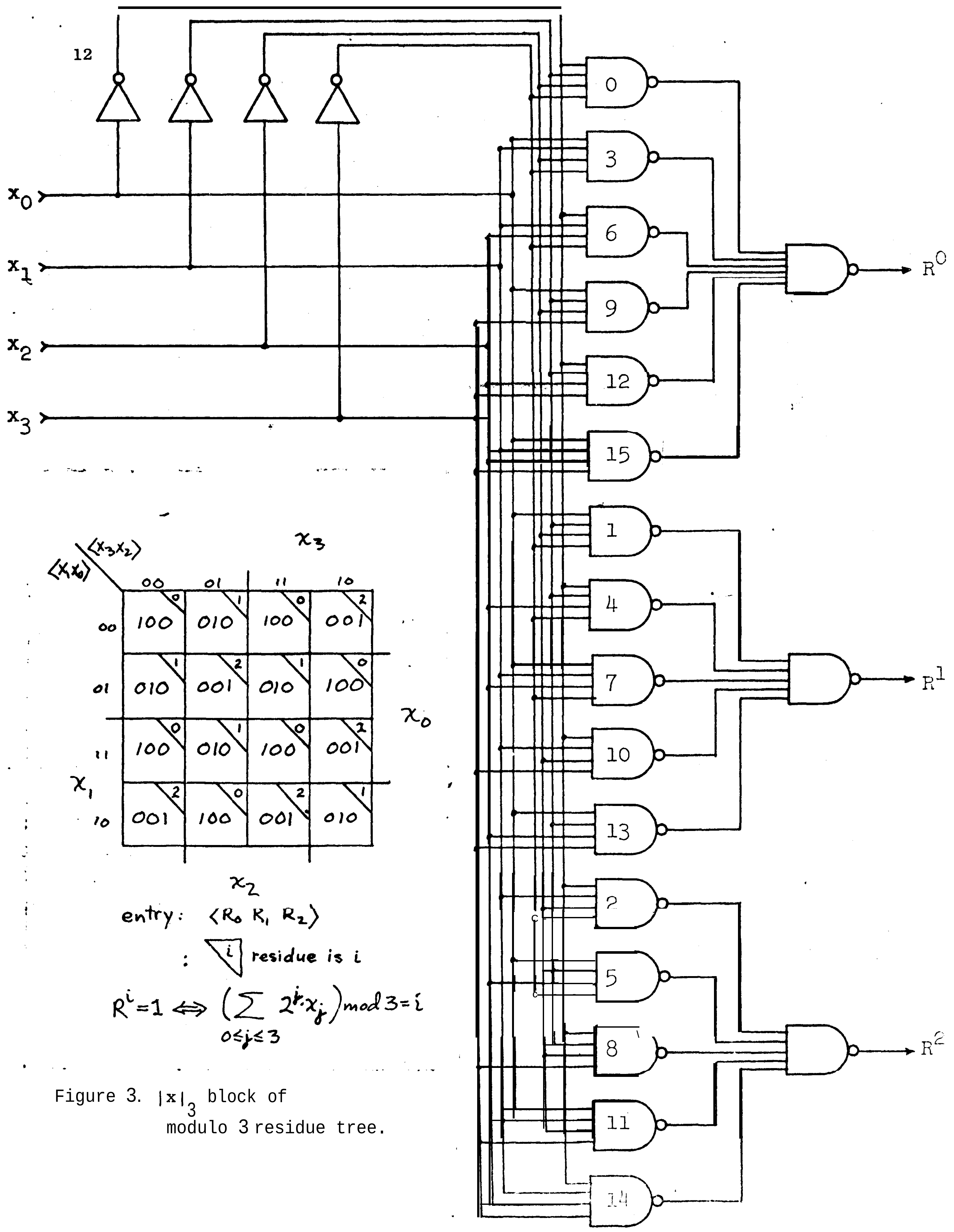


Figure 2. 16-bit modulo 3 residue tree.



$\langle x_1 x_0 \rangle$ \ $\langle x_3 x_2 \rangle$	00	01	11	10
00	100 ^{0}	010 ^{1}	100 ^{0}	001 ^{2}
01	010 ^{1}	001 ^{2}	010 ^{1}	100 ^{0}
11	100 ^{0}	010 ^{1}	100 ^{0}	001 ^{2}
10	001 ^{2}	100 ^{0}	001 ^{2}	010 ^{1}

x_3 x_0 x_1 x_2

entry: $\langle R_0\ R_1\ R_2 \rangle$

: $\boxed{i}$ residue is i

$$R^i = 1 \Leftrightarrow \Big(\sum_{0 \le j \le 3} 2^{j} \cdot x_j\Big) \bmod 3 = i$$

Figure 3. $|x|_3$ block of modulo 3 residue tree.

The $|x|_3$ block shown in Fig. 3. It realizes $R^0 = \Sigma(0,3,6,9,12,15)$, $R^1 = \Sigma(1,4,7,10,13)$, and $R^2 = \Sigma(2,5,8,11,14)$ with 4 inverters followed by 2 levels of NAND logic. The prime implicants of R^0, R^1, and 4^2 are single isolated cells on the Karnaugh map. Thus, this realization is a 4 bit to l-out-of-16 decoder, followed by summing gates to collect the respective implicants of R^0, R^1, and R^2. Except for the inverters, the circuitry for each R^1 is disjoint from the others.

The second rank block $+_3$ is shown in Fig. 4. Its inputs are 2 residues, each carried on 3 lines. We shall name these the X residue $X = \langle X^0, X^1, X^2 \rangle$ and the Y residue $Y = \langle Y^0, Y^1, Y^2 \rangle$. Its output R is carried on 3 lines $\langle R^0, R^1, R^2 \rangle$. The output is the l-out-of-3 encoding of the sum (modulo 3) of the X and Y residues.

Inverters are unnecessary because of the l-out-of-3 encoding of the inputs. The functions realized are

$$R^0 = X^0Y^0 + X^1Y^2 + X^2Y^1$$

$$R^1 = X^0Y^1 + X^1Y^0 + X^2Y^2$$

$$R^2 = X^0Y^2 + X^1Y^1 + X^2Y^0.$$

Each of the 9 NAND gates in Fig. 4 connected directly to the primary inputs corresponds to one of the 9 possible pairs of X and Y residues. Residue pairs with the same sum modulo 3 are collected by a second level NAND gate.

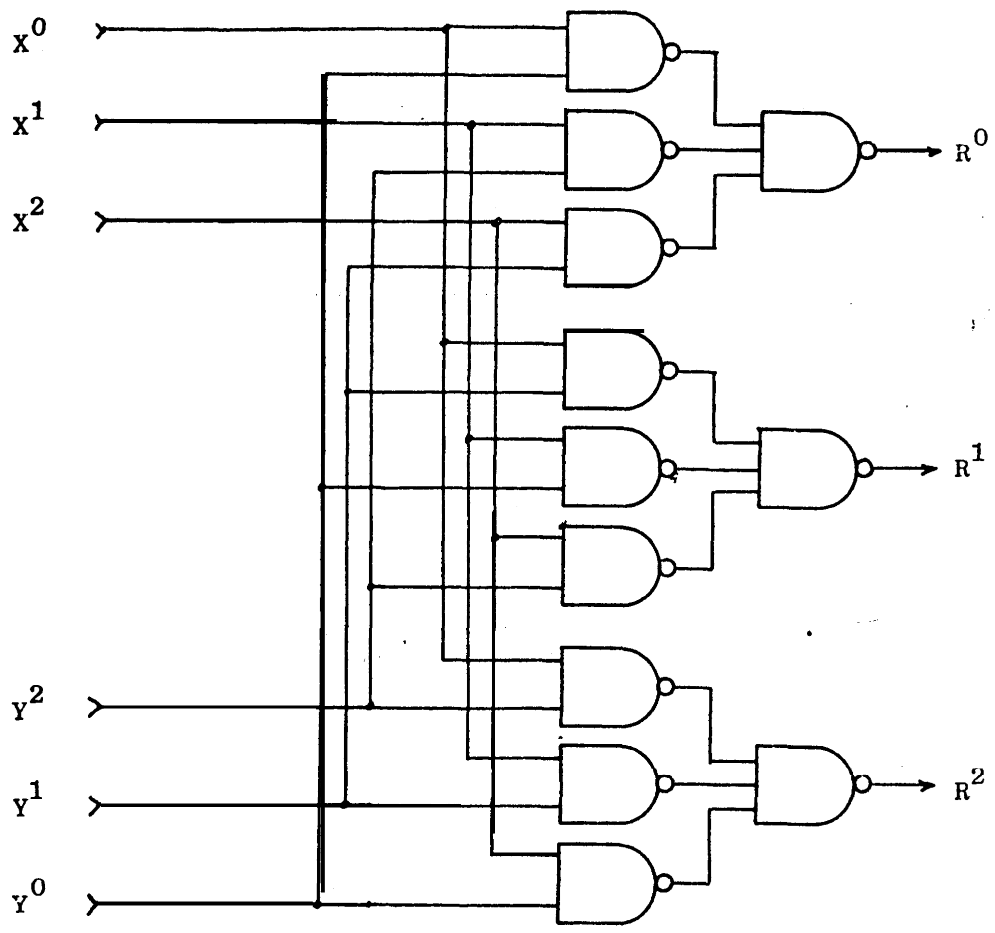


Figure 4. $+_3$ block of modulo 3 residue tree.

4. MODULO 5 RESIDUE TREE

A modulo 5 design is given as an example in Figs. 5, 6, and 7. The input is divided into 3 bit bytes, and thus the byte weights are 1, 2^3, 2^6, 2^9, etc. The residues of these weights are 1, 3, 4, 2, etc., respectively. Consequently, the multiplication of byte residue by byte weight is explicitly present as wire crossovers in Fig. 7. The width of the input number is not stated, so it might be necessary for the leftmost (most significant) $|x|_5$ block to have a byte width different from 3. That byte width is constrained to be at least 3 bits, since $\lceil \log_2 5 \rceil = 3$. In this example we have chosen to make all but one $|x|_5$ cell have the minimum byte width of 3. The last $|x|_5$ cell must then have the appropriate width to finish off the input number which would then be 3, 4, or 5, depending on the width of the input number. For example, a 32 bit input number would require a width of 5 bits for the last $|x_5|$ cell. Whatever the width of that cell turns out to be, it has basically the same design as the 3 bit $|x|_5$ block in Fig. 5. The decoder width, and output gate fan-ins would be the only difference.

The basic 3 bit wide $|x|_5$ block is shown in Fig. 5, together with specifications of the wire crossovers for any byte weight. The same complete decoding is the basis of the block, in this case 3 bits to l-out-of-g. After decoding, appropriate products are summed together by second level NAND gates. Adjacent cells of the Karnaugh map can be seen to have different residues. This is generally true, for adjacent cells correspond to inputs which differ by a power of 2, and they could have the same residue only if the modulus divided that power of 2 difference. We are restricted to an odd modulus, so this can never happen.

The $+_5$ block is shown in Fig. 6. Its structure follows the basic design closely. With two 5 wire inputs $<X^0,X^1,X^2,X^3,X^4>$ and $<Y^0,Y^1,Y^2,Y^3,Y^4>$, its 5 wire output $<R^0,R^1,R^2,R^3,R^4>$ is given by the logic equations given in the basic design section, for A=5:

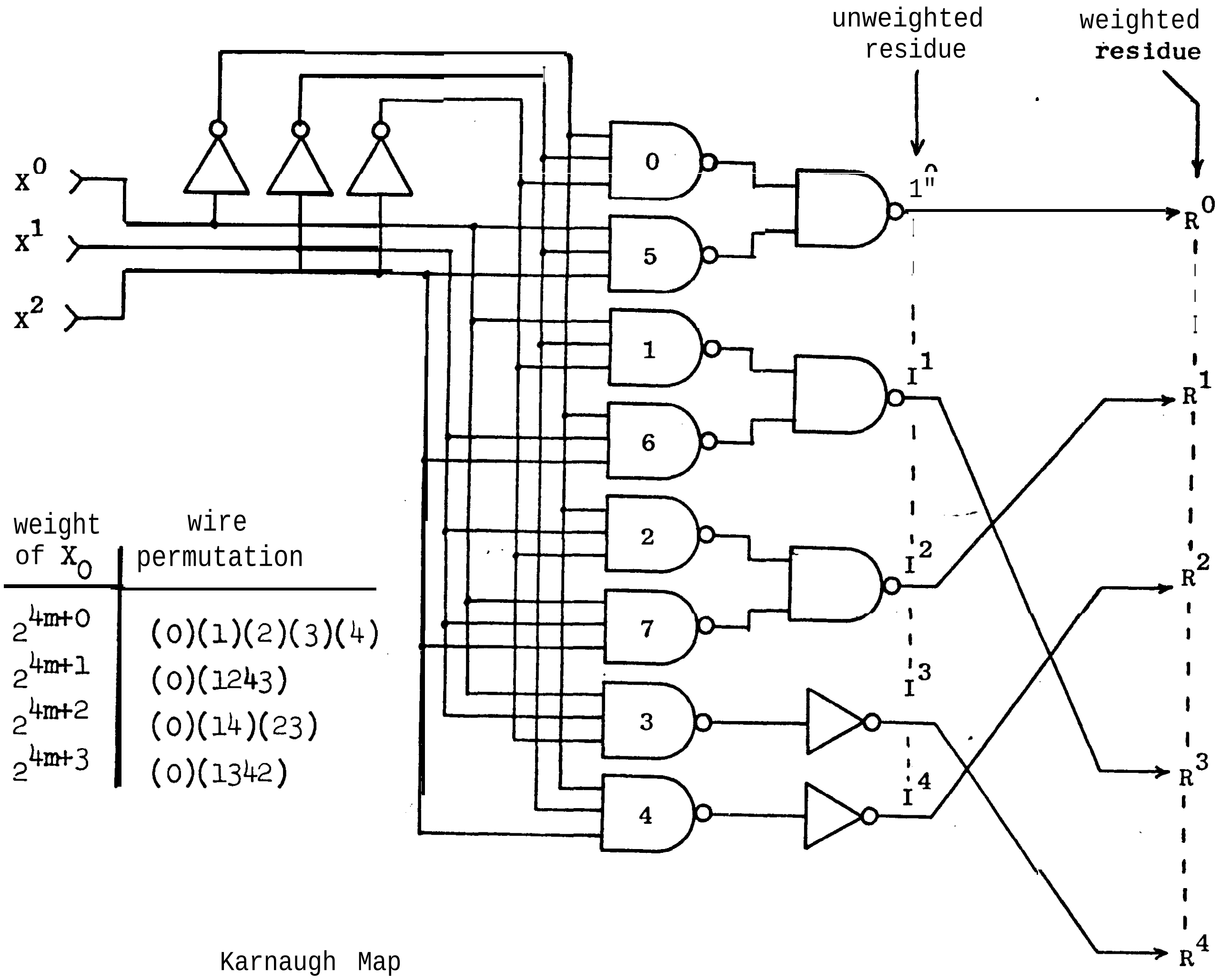
$$R^0 = X^0Y^0 + X^1Y^4 + X^2Y^3 + X^3Y^2 + X^4Y^1$$

$$R^1 = X^0Y^1 + X^1Y^0 + X^2Y^4 + X^3Y^3 + X^4Y^2$$

$$R^2 = X^0Y^2 + - - -$$

$$R^3 = X^0Y^3 + - - -$$

$$R^4 = X^0Y^4 + - - -$$



weight of X_0	wire permutation
2^{4m+0}	(0)(1)(2)(3)(4)
2^{4m+1}	(0)(1243)
2^{4m+2}	(0)(14)(23)
2^{4m+3}	(0)(1342)

Karnaugh Map

$x^0 \diagdown x^2x^1$

			x^2	
	10000 0	00100 2	01000 1	00001 4
x^0	01000 1	00010 3	00100 2	10000 0
		x^1		

entry: $\langle R^0R^1R^2R^3R^4\rangle$

$\boxed{i}$: residue is i

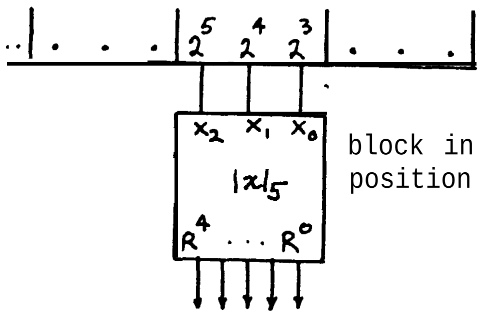


Figure 5. $|x|_5$ block of mod 5 residue tree, showing wire crossovers for group weight 3' (mod 5).

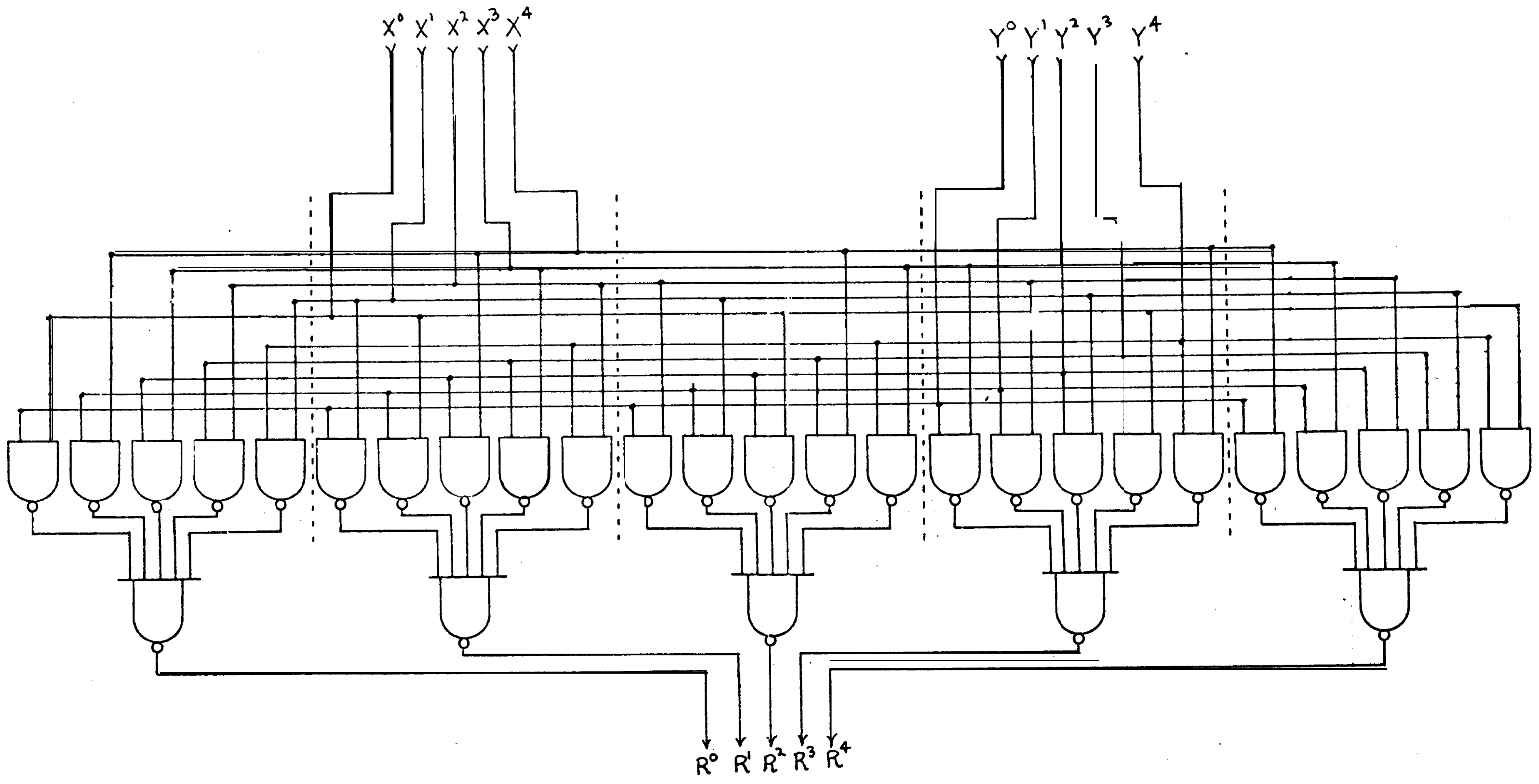

X^0 X^1 X^2 X^3 X^4
Y^0 Y^1 Y^2 Y^3 Y^4
r
R^0 R^1 R^2 R^3 R^4

Figure 6. $+_5$ block of modulo 5 residue tree

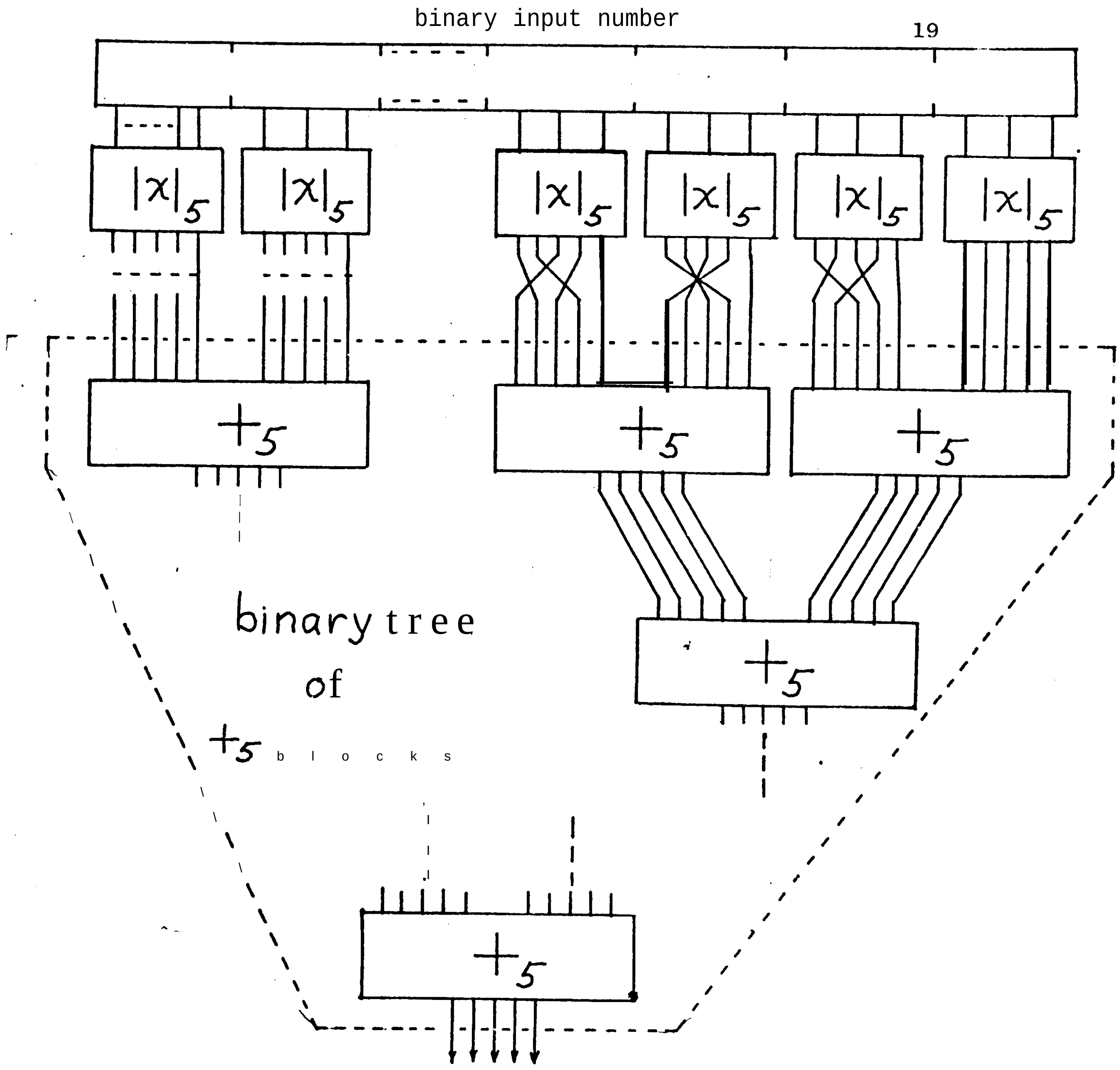
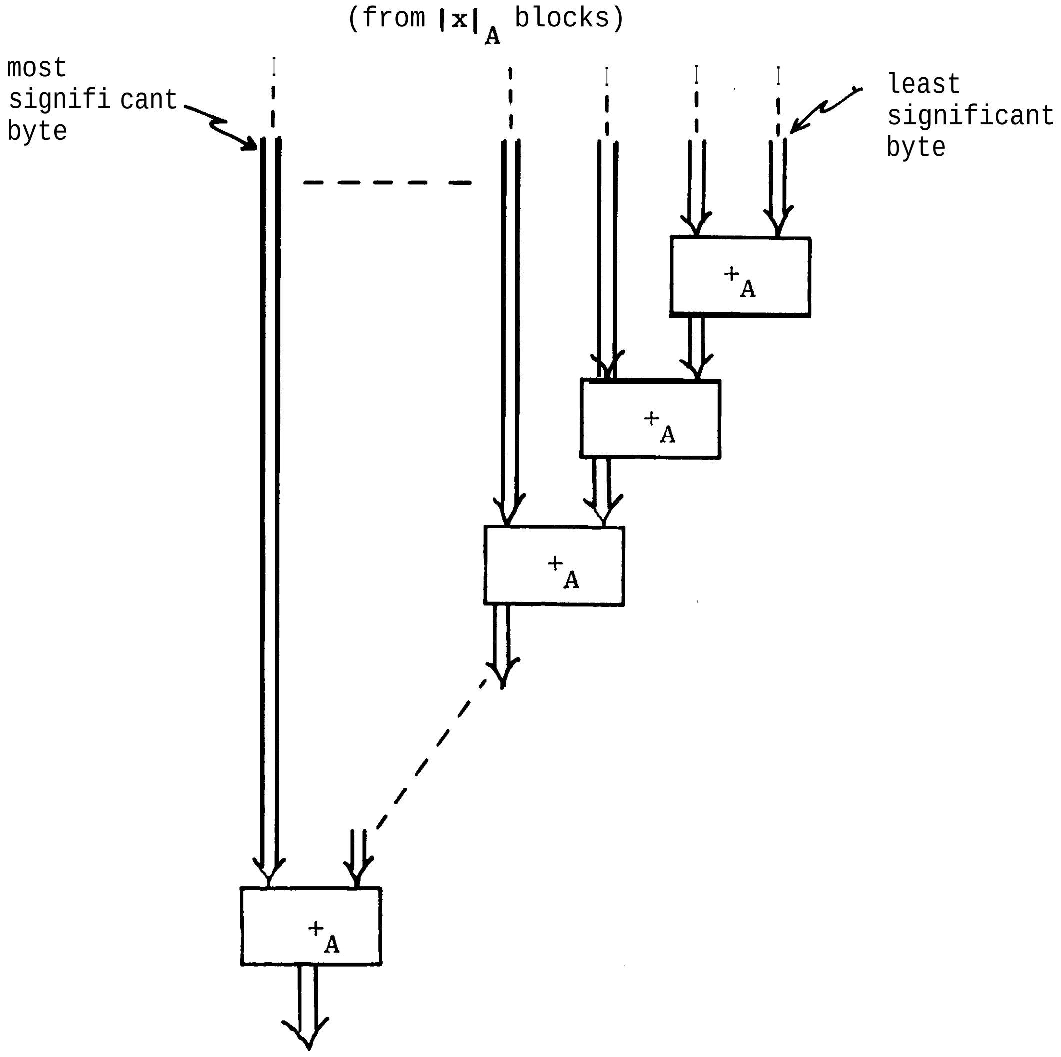


Figure 7. Modulo 5 residue tree.

The example for an input width of 32 bits and a modulus of 5 is not the only design possible. The $|x|_5$ blocks need not have the same width. Their widths are each at the designer's option, subject to the minimum width constraint of 3. Choosing the minimum width $|x|_5$ cell gives the smallest gate count over the $|x|_5$ cells, except for end effects: an $|x|_5$ cell of width w has w inverters, and 2^w+5 decoding and summing gates. A good example of the end effect is that for modulus 5 and an input number of width 32 bits, the smallest gate count over the $|x|_5$ blocks is realized when 8 blocks have the minimum width of 3 bits, and 2 blocks have a width of 4 bits. Several specific designs for a 32-bit wide, modulo 5 residue tree are given in Appendix B.

Minimizing the gate count over the $|x|_A$ cells in a general design for modulus A, however, increases the number of $+_A$ cells needed to sum their outputs. This could possibly require an extra level in the $+_A$ tree, over the number of levels required for a design with wider $|x|_A$ cells.

The $+_A$ trees seen in Figs. 1, 2, and 7 have all appeared balanced. The basic design does not, however, require such a balance, Indeed, the example of a 32-bit wide modulo 5 residue tree would have an unbalanced arrangement of $+_A$ blocks, since no balanced binary tree has 10 terminal nodes. A designer might wish to further imbalance the $+_A$ tree, in order to minimize the propagation delay through the network for certain bits. For example, in a simple ripple-carry adder without end-around carry, the least significant output bits would be ready sooner than the most significant ones. If it were desired to have the residue of this output with as small a delay as possible, a design with short propagation delay for the most significant bits, and a long propagation delay for the least significant bits would be faster. A $+_A$ tree which has this characteristic has a great deal of imbalance:



Minimization of gate count or propagation time of a residue tree is an involved matter, which we will not present in this paper. This minimization is essentially a solution of system of integer equations, for which we have used simple exhaustion of cases. With the application of MSI, a minimum gate count design might not have minimum package count, and obtaining minimum delay or package count depends heavily on what is available in MSI. We give a sample design using part MSI and part small-scale integration in Appendix B, for a 32 bit wide modulo 7 residue tree.

5. PROOF OF TOTALLY SELF-CHECKING OPERATION

In order that the residue tree be totally self-checking, we must prove that a single fault can never cause the output to be a code word other than the correct one (fault-secure), and that at least one input number exists for each single stuck-fault, such that the output with that input and that fault is a non-code output (self-testing).

We proceed by first showing that any non-code (not I-out-of-A) output, from any block of the network, propagates to the output, giving a non-code output. Such propagation occurs only through $+_A$ blocks, as seen in Fig. 1. We reproduce here the logical equations for the $+_A$ block, given in the basic design section.

$$R^0 = X^0Y^0 + X^1Y^{A-1} + - - - - \quad + X^{A-1}Y^1$$

$$R^1 = X^0Y^1 + X^1Y^0 + - - - - \quad + X^{A-1}Y^2$$

$$R^{A-1} = X^0Y^{A-1} + X^1Y^{A-2} + - - \quad + X^{A-1}Y^0$$

These can be condensed into a single general form, the sum of products

$$R^i = \sum_{\substack{0 \le j \le A-1 \\ k=(i-j) \bmod A}} X^j Y^k \qquad (3)$$

Suppose we have a non-code input with k l's ($k \ge 2$) on the X residue, and some correctly coded Y residue. Examining (3), we have A^2 2 input AND gates, each set to output a 1 for the occurrence of a unique X,Y residue pair. With this non-code input, k different X,Y residue pairs seen present. Each pair has a different sum modulo A, so the output has as many l's as the X inputs. Since the X and Y inputs may be exchanged without changing the output, the $+_A$ network propagates a multiple l's input as a multiple l's output, provided only that its other input is a l-out-of-A code word. This proviso is satisfied by the single-fault assumption, which we have already made.

Now consider the case of either the X or Y input being all zero. Referring to (3), obviously no R^1 can be 1, since all products are zero. Thus an all-zero non-code input to a $+_A$ block propagates to its output. The all-zero and multiple l's cases exhaust all the non-code words on A lines. Then if a single fault occurs in some block, and some input number causes that block output to be non-code, all blocks which are successors of the faulty one will have a non-code output. Thus the residue tree output will be non-code.

We must now show that as the input number takes on all values, each block receives inputs which are sufficient to cause at least one non-code block output,for any particular single stuck fault in that block. Further, we must show that under any particular stuck fault within a block, no correctly coded input can cause the block output to be a different code word from the correct one. There are no stuck-faults between blocks -- they are the same as a block output line being stuck.

The second point (fault-secureness) is easiest to deal with. For the $+_A$ blocks, (see Figs. 4 and 6) each output line is realized by a different subnetwork. Then a single fault can cause an error of maximum Hamming distance 1, while the code is distance 2. Then it is impossible for a fault to change the output of a $+_A$ block from one code word to another.

For the $|x|_A$ block, the output subnetworks are disjoint, except for the inverters (see Figs. 3 and 5). An inverter output stuck-at-l causes 2 adjacent inputs to be sensed by the decoder, when the corresponding input line is 1. Adjacent inputs have different residues (since $A \neq 2^i$), so 2 output lines go up, giving a non-code output. An inverter output stuck-at-0 gives an all-zero non-code output when the corresponding input line is 0. The other faults in the $|x|_A$ block can affect only one output line, and the code distance assures that none of these faults can change one code word to another.

The first point (self-test property) remains to be dealt with -- each block must receive sufficient inputs, under normal operation, such that at least one non-code block output occurs for each fault in that block. For the $|x|_A$ block, we have already verified this for inverter faults. The remaining faults can change only one output lead, so if an input combination "detects" one of these fauits, the output for that input is distance 1 away from the correct output, and hence is a non-code output, whenever that fault is present.

It is easy to verify that the normal inputs detect every one of the remaining faults. Since the normal inputs to a $|x|_A$ block are all possible combinations, we need only verify that the $|x|_A$ block is irredundant. For if so, there are no "undetectable" faults, and the set of normal inputs will detect all faults. The terms which the output gates sum are all fundamental products, or single cells on the Karnaugh map. Since all adjacent cells have different residues, the prime implicants of each output line are single cells on the Karnaugh map. Since each output line is realized by summing the appropriate single cells of the Karnaugh map, the realization is a sum of prime implicants, and hence is irredundant.

The $+_A$ block must now be examined to see if the A^2 combinations of input residues, which it receives under normal operation, are sufficient to cause at least one non-code output for each single fault. The networks which realize each of the A output lines are disjoint, so when a pair of input residues detects a fault, only one output line will be wrong. Because the l-out-of-A code has distance 2, the output then is non-code. So we will show that each fault of each output subnetwork is detected by one of the A^2 residue pairs presented to the block.

If the network is 2 levels of NAND logic, its behavior under no faults and under single faults is the same as the equivalent 2 level AND-OR realization. Because the output subnetworks are disjoint, and each is realized in 2 levels of logic, we need only test the AND gate inputs for stuck-at-l and stuck-at-O. [7] These tests cover all the OR gate faults and the AND gate stuck-output faults as well.

Consider the subnetwork which realizes R^1, and specifically consider the gate

$$X^a Y^{(i-a) \bmod A} \quad \text{where} \quad a,i \in \{0,1,2,\ldots, A-1\}.$$

To test the X input for stuck-at-1, we need a 0 on X^a, a 1 on $Y^{(i-a) \bmod A}$, and at least one 0 on all the other AND gates of this subnetwork. Any residue pair with Y residue (i-a) mod A will put the 1 on $Y^{(i-a) \bmod A}$, and referring to (3), it will put a 0 on the Y inputs of all the other AND gates in the subnetwork. Any X residue other than a puts the required 0 on lead X^a, so we have A-1 valid residue pairs which test the X^a lead for stuck-at-1. Similarly, an X residue of a and any Y residue other than (i-a) mod A tests the Y input of this gate for stuck-at-1.

To test this same gate for inputs stuck-at-0, we need a 1 on the X^a lead and a 1 on the $Y^{(i-a) \bmod A}$ lead, and at least one 0 on each of the other AND gates in the subnetwork. The residue pair with X residue a and Y residue (i-a) mod A furnishes the 1's, and puts two 0's on every other AND gate in this subnetwork.

So we have sufficient input residue pairs to detect all faults at the gate $X^a Y^{(i-a) \bmod A}$ in the R^1 output subnetwork. Since the argument above is valid for any a and any i in the range $\{0,1,\ldots,A-1\}$, we have sufficient residue pairs to detect all faults of all AND gates in all output subnetworks, and hence, all faults in all gates in the $+_A$ block.

With this result, we can now say that every block in the residue tree will exhibit a non-code output for each fault within. We have already shown that no fault in any block can cause its output to change from one code word to another, and we have shown that any non-code input to a $+_A$ block causes a non-code block output. We conclude that the residue tree is totally self-checking when all input numbers are given as inputs.

6. BIRESIDUES FOR LARGE MODULI

For large moduli, the fan-in required at the output gates of the $+_A$ block becomes excessively large. For modulus 3 (Fig. 3), 3 inputs were needed. For modulus 5 (Fig. 5), 5 inputs were needed, and in general, modulus A requires A inputs. There is -a practical limit on how many lines can be OR'ed together, even with the wired-OR technique, or with small-scale expandable gates. This difficulty can be circumvented by using two or more distinct totally self-checking residue trees, with moduli P, Q, R, etc., chosen so that their least common multiple equals or exceeds A. Their outputs are then combined by a translator, which translates from the code of multiple residues to l-out-of-A. The translator consists of A disjoint 2-level AND-OR networks. If the moduli P, Q, R, etc., are relatively prime and their product equals A, the translator reduces A single AND gates, one for each output residue.

An example is given here using two residue trees, one modulo 3, and one modulo 5, to obtain, the modulo 15 residue of an input number. The circuit is given in Fig. 8. The translator is very similar to the $+_A$ block, of the basic design. By similar reasoning, it propagates non-code inputs (i.e., inputs which have all zero or multiple-l's from one of the residue trees) as non-code outputs, and it is tested by code inputs.

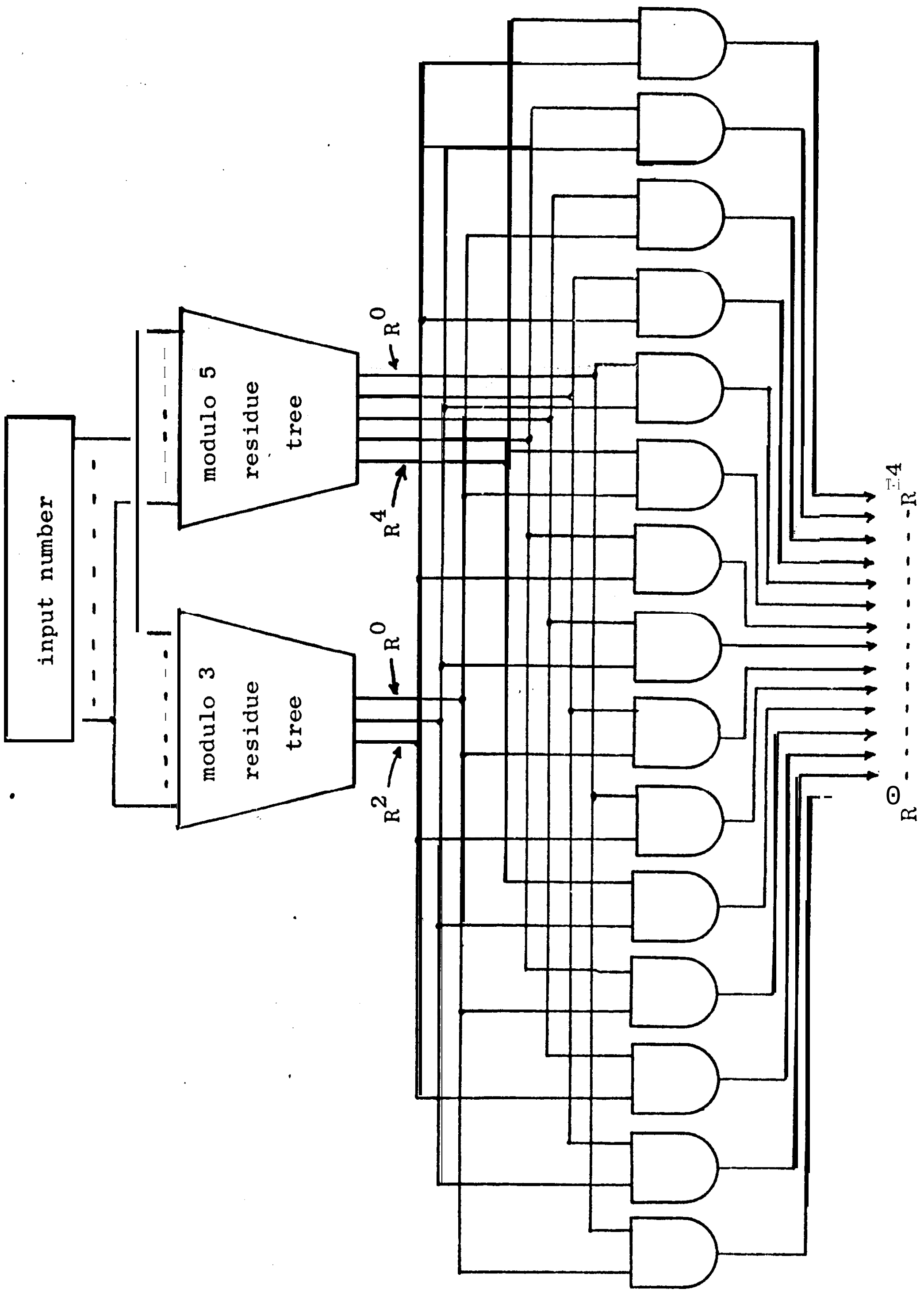


Figure 8. Modulo 15 residue network.

7. USE IN A TOTALLY SELF'-CHECKING RESIDUE CHECKER

An immediate application of the totally self-checking residue tree is checking addition. The checking technique is the long-recognized residue check. Suppose an adder network obtains the sum of two inputs, n_1 and n_2. Let the adder output be named "sum." Thus the correct addition is given alegbraically by

$$n_1 + n_2 = \text{sum},$$

where + is addition modulo M. If the adder is one's complement, M is one less than some power of two. If the adder is two's complement, M is equal to some power of two. In either case, the homormorphy of modulo addition holds.when A divides M, so that

$$|n_1|_A +_A |n_2|_A = |\text{sum}|_A. \tag{4}$$

If, because of some failure in the adder, its output differs from the correct output by a number which is not a multiple of A, the actual (incorrect) output and the inputs, n_1 and n_2, will not satisfy (4). Thus the residue check consists of obtaining $|n_1|_A$, $|n_2|_A$, and $|\text{sum}|_A$, and then checking that they satisfy (4). A network which performs this check is shown in Fig. 9.

If the totally self-checking residue tree is used to extract the three residues, the $+_A$ block is used to add the input residues, and a totally self-checking equality checker EQ can be designed for l-out-of-A coded inputs, the checker shown in Fig. 9 will be totally self-checking. Other totally self-checking addition checkers have already been found in [4]. We show Fig. 9 as an exercise in the use of the residue tree presented here.

Referring to Fig. 9, if a single fault occurs in one of the two residue generators $|n_1|_A$ or $|n_2|_A$ or in the modulo A adder $+_A$, the output of $+_A$ will be non 1-out-of-A for some set of checker inputs n_1, n_2, sum. For this set of inputs, the equality checker will signal failure, just as if the sum itself were in error. If, however, a fault occurs in the binary adder which is detectable by the residue code, the equality checker will see two properly encoded (1-out-of-A) but different signals X,Y, and it will signal a failure. If, instead, a fault occurs in the residue generator $|sum|_A$, that generator output will be non 1-out-of-A for some set of inputs, and EQ will sense the resultant disparity of X and Y. Lastly, if a fault occurs in the equality checker EQ, we aren't sure what may happen. If EQ is totally self-checking with respect to the inputs it receives when the residue generators, the binary adder, and $+_A$ are all working correctly, then the whole residue checker is totally self-checking. Otherwise, all we can say is that _most of it_ is. The question of whether EQ can be realized in a totally self-checking fashion will be dealt with in another paper. All we shall say here is that for A=3, EQ cannot be made totally self-checking, while for A=15, it can.

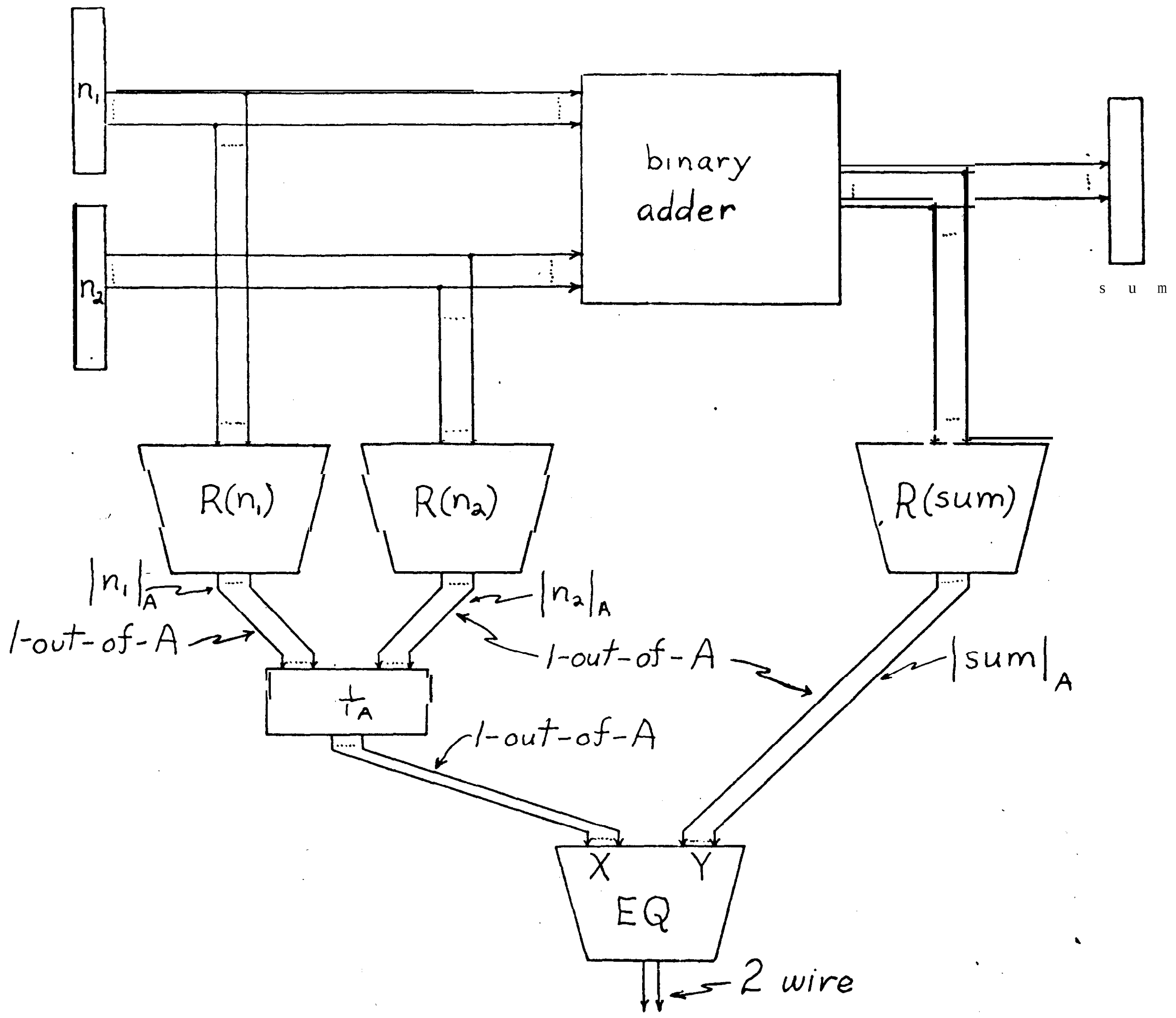


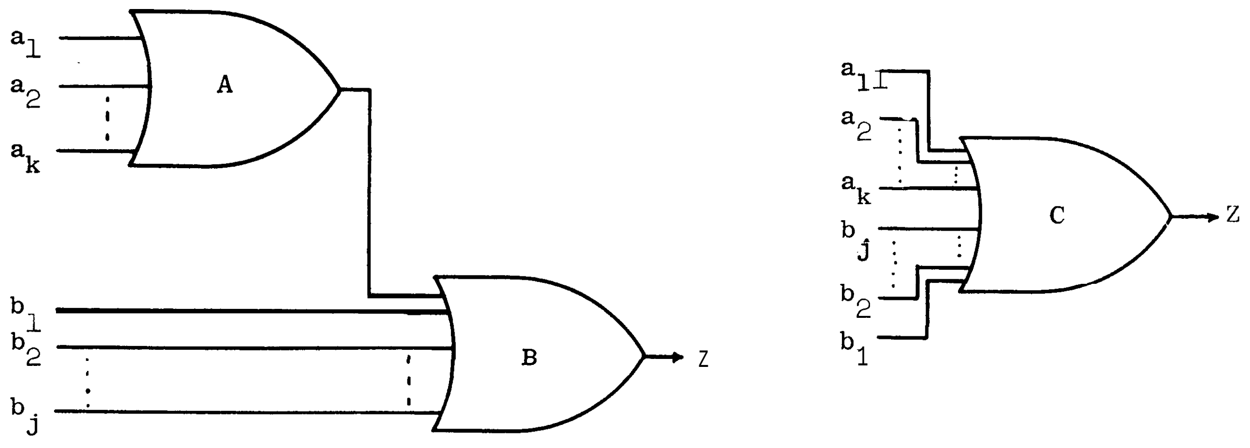
Figure 9. Residue checker for addition.

8. LEVEL MERGING OF SECOND RANK CELLS

The residue tree for a general modulus A (seen in Fig. 1) requires 2 levels of logic for each second rank cell unless wired-OR is practical. For very wide input numbers, the total delay thus incurred may be prohibitively large. The writer is indebted to one of the anonymous reviewers of this paper, for suggesting the level-merging technique as a means of reducing this delay. This technique was used in references [2] and [3].

The second rank cell ("$+_A$" in Fig. 1) can be realized by a 2 level OR-AND network. Then by using alternating layers of AND-OR and OR-AND $+_A$ blocks, adjacent layers can be "merged" together, leaving only one gate delay per layer. The merge operation simply replaces any cascade of OR gates with a single, equivalent OR gate, and does likewise with cascaded AND gates.

If the network with alternating AND-OR and OR-AND $+_A$ blocks is totally self-checking, the network will be totally self-checking after the merge operation, as follows. Suppose a network contains two cascaded OR gates A and B,



with lines labelled as shown. Gate C has the same truth table as the A,B cascade, when both are fault-free. Further, for each single stuck-fault of gate C, there is one single stuck-fault of the cascade A,B which has the same truth table. Stuck-at-0 on a_i, b_i, and Z of gate C produce the same truth table as stuck-at-0 on a_i, b_i, and Z, respectively, of cascade A ,B. Similarly for stuck-at-l on these lines. Thus each fault of gate C mimics a fault in the original cascade, so a parent network which is totally self-checking and contains cascade A,B, would be totally self-checking if the cascade were merged into the single gate C. The same argument extends to cascaded AND's.

The AND-OR second rank cell which we used earlier is equivalent to the sum-of-products expression

$$R^k = \sum_j (X^j Y^{k-j})$$

e.g. $$R^2 = X^0Y^2 + X^1Y^1 + X^2Y^0 + X^3Y^4 + X^4Y^3$$

for modulus 5 (A=5)

Here, justaposition is the AND operation, $\sum$ is the OR operation, and the arithmetic in the superscript position is done modulo A.

The thoughtful reviewer proposed an OR-AND cell which has nearly the same truth table. To describe it conveniently we shall introduce a little notation. Let X^a represent the a^th lead of the X group, just as before. But let

$$Y^a = Y^0+Y^1+\ldots+Y^{a-1} \qquad +Y^{a+1}+Y^{a+2}+\ldots+Y^{A-1}$$

which is the OR of all the Y leads, except for the a^th one. With this notation the proferred OR-AND cell is equivalent to the product-of-sums expression

$$R^k = \prod_j (X^j + Y^{\overline{k-j}})$$

$$R^2 = (X^0+Y^2)(X^1+Y^1)\ldots(X^4+Y^3)$$

$$= (X^0+Y^0+Y^1+Y^3+Y^4)(X^1+Y^0+Y^2+Y^3+Y^4)\ldots$$

$$\ldots(X^4+Y^0+Y^1+Y^2+Y^4)$$

for modulus 5 (A=5).

Here, + is the OR operation, $\prod$ is the AND operation, and the superscript arithmetic is once again modulo A. The corresponding network is seen in Fig. 10. We use "output subnetwork" to denote all the predecessor gates of a particular output line. A complete OR-AND all for A=3 is given in Fig. 11.

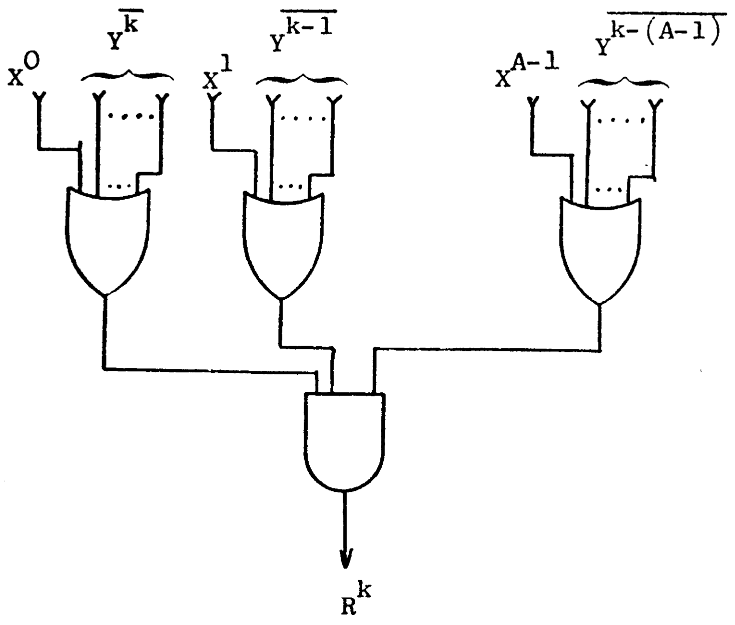


Figure 10. k$\underline{\text{th}}$ subnetwork of OR-AND $+_A$ block.

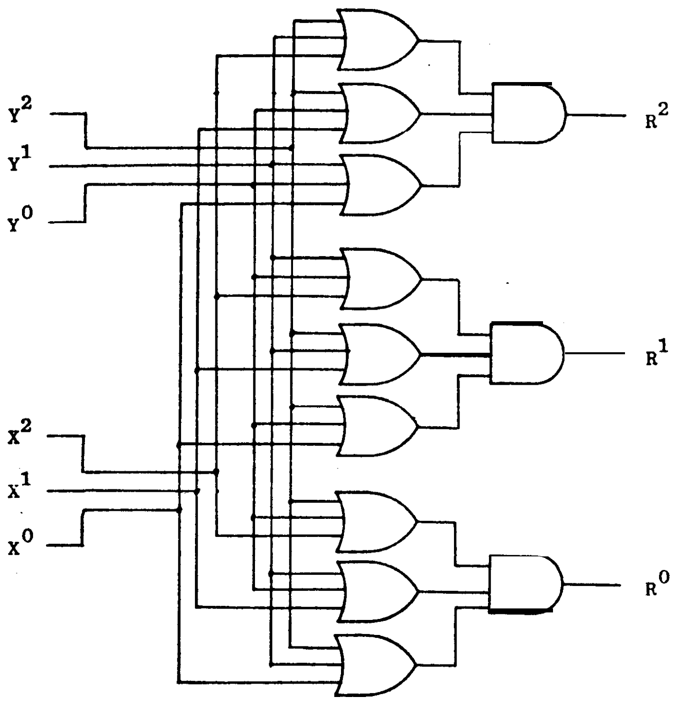


Figure 11. Complete OR-AND $+_3$ block.

We shall first verify that this cell produces the correctly encoded modulo A sum of its inputs. Next, we shall verify that, given an all-zero or multiple-l's input, the cell output will be all-zero or multiple-l's respectively. Finally, we shall verify that the A^2 properly encoded combinations of X and Y residues completely test this cell for single stuck-at faults. Once these claims are established, it follows that this cell may be used interchangeably with the $+_A$ cell in the residue tree of Fig. 1. Then AND-OR and OR-AND $+_A$ cells could be used in alternate layers, and the tree would still be totally self-checking. Then cascaded AND gates could be collapsed into single gates with no change in the residue tree's normal, fault-free behavior, or in its behavior under a single stuck-fault. Similarly for OR gates. The final result of the level-merging operation is an m-layer tree of second rank $+_A$ cells with only m+1 gate delays. Of course, this increase in speed isn't free. For the A=5 modulus, we need 25 (A^2) 5-input (A-input) gates per second-rank cell, compared to the 5 (A) 5-input (A-input) gates needed when level merging isn't used.

Suppose the OR-AND cell is presented with correctly coded (l-out-of-A) residues u and k-u, on the X and Y inputs, respectively. We shall use modulo A arithmetic in the following. R^k is true as follows. $\forall j$, either $j=u$, and X^j is true, or else $\neq u$, and Y^{k-u} is in Y^{k-j} so Y^{k-j} is true. Then all the OR gates of the kth output subnetwork have true outputs, so R^K is true. R^m, $m\neq k$, is false, as follows. In this subnetwork, the OR gate with Y inputs $Y^{\overline{k-u}}$ has no Y inputs true, and its X input, $X^{m-(k-u)}$ is false, because only X^u is true, while m-(k-u)#u. Then the AND gate of this subnetwork has at least one false input and indeed, it has exactly one false input, for any other OR gate has Y^{k-u} among its inputs and hence has a true output. Then the R^m subnetwork has a false output.

Suppose the OR-AND cell is presented with an all-zero Xinput, while the Y input residue is u, and is properly encoded. Then R^k is false as follows, $\forall k$. R^k can be true only if all its OR gates have true outputs, which is the case only if every OR gate has a Y input true. But the $j\underline{^{th}}$ OR gate, where k-j=u, has no Y inputs true,

Suppose the OR-AND cell is presented with a properly encoded Y residue, but 2 X lines are up (true). Let X^u, and X^{u+e} be true, where $e\neq 0$, and let Y^{k-u} be true. Then R^k and R^{k+e} are both true, by the same argument which was used regarding normal operation.

Suppose the OR-AND cell is presented with an all-0 Y input, while the X input is properly encoded. Then R^k can be true iff X^j is true, $\forall j$. But this is patently false, so R^k is false, $\forall k$.

Suppose the OR-AND cell is presented with a properly encoded X residue, but 2 Y lines are up. By the same argument used for normal operation, two R lines are up.

This establishes the operation of the OR-AND cell for properly encoded inputs, and for inputs having one bit incorrect. Next, we shall verify that the set of A^2 properly encoded input pairs X,Y is a complete fault detection test set for single faults. We shall use a deductive argument, as before. The network in Fig. 10 would be tested for all single stuck faults by a test set which

1)(bubble-1 test) places a single "1" input on each of the OR gates. Over the entire test set, for any particular OR gate, each input of that gate must be visited by the "1", and

2)(all-zero test) for each OR gate, places all 0's on its inputs, while placing at least one "1" on every other OR gate.

Happily, every properly encoded input sets up a bit pattern as described in either 1) or 2), on every one of the output subnetworks. No proper input creates a bit pattern on the OR gates of any output subnetwork, which is not one of the single-fault test patterns described in 1) and 2). And moreover, all the test patterns required by 1) and 2) exist in the set of A^2 properly encoded inputs. We shall verify this, in the following, and conclude that the OR-AND cell in Fig. 9 is tested for single faults by the normal inputs (X and Y residues both 1-out-of-A).

Suppose we apply all A input codewords, which should result in an output residue of k, for some $k \in Z_A$. These are all the possibilities of $X^u=1$, $Y^{1-u}=1$, $X^J=0\ \forall j \neq k$, $Yj=0\ \forall j \neq k-u$. Consider the $j^{\underline{th}}$ OR gate of the $k^{\underline{th}}$ output subnetwork, for some $j \in Z_A$. Now if $j=u$, that gate has its X lead at 1, and all of its Y leads at 0, since $Y^{k-j} = Y^{k-u}$ is the Y lead which is absent. If $j \neq u$, that gate has $X^J=0$, while Y^{k-u} is among its inputs, and $Y^{k-u}=1$. Thus every gate has one input 1, and all others 0. If we fix j, and examine the $j^{\underline{th}}$ OR gate while u ranges through all of its values, the single 1 is seen to visit all A inputs of the gate. So for any gate j, and for any output subnetwork k. Thus the set of properly encoded inputs contains all the tests in 1).

Now suppose we apply the same set of input codewords, but now consider the $m^{\underline{th}}$ output subnetwork, where $m \neq k$. The OR gate with inputs Y^{k-u} has all 0 inputs, as we have already discussed in the section on normal operation of the OR-AND cell. All the other OR gates of this group have "1" outputs, since Y^{k-u} is included in their inputs. As u ranges over all its values, all the OR gates of this subnetwork get the all-zero test required in 2) above, one at a time. Thus, when the input results in an output residue of k, and all A such inputs are applied, the kth output subnetwork gets all the tests of 1), while the mth output subnetwork gets all the tests of 2), $\forall m \neq k$. Then the set of properly encoded inputs (X residue is l-out-of-A and Y residue is l-out-of-A) contains all the tests mentioned in 1) and 2) above, and the level merging scheme gives a totally self-checking residue tree.

REFERENCES

[1] W. C. Carter and P. R. Schneider, "Design of dynamically checked computers," IFIPS 1968, pp. 878 ff.

[2] W. C. Carter, D. C. Jessup, and A. B. Wadia, "Implementation of checkable acyclic automata," IBM Report No. RC-3324, April 1971.

[3] D. A. Anderson, "Design of self-checking digital networks using coding techniques," University of Illinois Coordinated Science Laboratory Report No. R-527, September 1971. Also published as Ph. D. thesis.

[4] A. B. Wadia, "Investigation into the design of dynamically checked arithmetic units," IBM Report No. RC-2787, February 1970.

[5] N. S. Szabo and R. I. Tanaka, Residue Arithmetic and its Applications to Computer Technology, McGraw-Hill, 1967.

[6] M. R. Paige, "Generation of diagnostic tests using prime implicants," University of Illinois Coordinated Science Laboratory Report No. R-414, May 1969.

[7] Z. Kohavi and D. Spires, "Designing sets of fault-detection tests for combinational logic circuits,' IEEE Trans. on Computers, Vol. C-20, December 1971.

APPENDIX A

Proof that multiplication of the integers Z_A (Z_A is the set of integers modulo A) by the group weight $s = (2^w) \bmod A$ is a l-l onto map from Z_A to itself, so long as A is odd. This is proven by proving the two propositions below. The hypothesis above follows directly from these two propositions.

Proposition A: That multiplication by s (modulo A) of the integers Z_A is a l-l onto mapping from Z_A to itself iff GCD(s,A) = 1.

Proposition B: T h a t A) =GCD(iff $|^n|_A$ GCD(r,A) = 1. Here $|n|_A$ means the residue (n) modulo A.

Proof of A

Suppose GCD(s,A) $\neq$ 1. Then $\exists\ a_1, a_2 \in Z_A$, where al $\neq a_2$, such that: $s \cdot (a_1 - a_2) = A$. Then $s \cdot a_1 \equiv s \cdot a_2$ (modulo A) and the multiplication is not a l-l map. Proves the only if part of the proposition by having proven the contrapositive here.

Now to prove the if part of proposition A, suppose the multiplication is not l-l. Then $\exists\ a_1, a_2 \in Z_A$, where al $\neq a_2$, $a_1 > a_2$ such that

$$s \cdot a_1 \equiv s \cdot a_2 \quad \text{(modulo A).}$$

Then $s \cdot (a_1 - a_2) = mA$, where m = integer. Since $0 < (a_1 - a_2) < A$, $(a_1 - a_2)$ does not contain all the factors of A, and therefore s must contain the balance of them, and GCD(s,A) $\neq$ 1. Then the contrapositive holds: $GCD(s,A) = 1 \Longrightarrow$ (multiplication is l-l). Since any l-l map from any set Z to itself is necessarily onto, so is this multiplication. Proves the <u>if</u> part.

∎

Proof of B

$GCD(r,A) = 1 \Longleftrightarrow GCD(r^n,A) = 1$, so we shall prove $GCD(|r^n|_A, A) = 1 \Longleftrightarrow GCD(r^n,A) = 1$. Indeed, let $r^n = t$, and we shall prove $GCD(|t|_A,A) = 1 \Longleftrightarrow GCD(t,A) = 1$. This is all that is needed to prove the proposition.

Now $|t|_A + mA = t$ (m integer)

Suppose $GCD(t,A) \neq 1$. Then $GCD(t,A) = s$, some $s > 1$. Then $A = f_1 \cdot s, t = f_2 \cdot s$.

$$|t|_A = t-mA$$

$$|t|_A = f_2 \cdot s - m \cdot f_1 \cdot s$$

$$|t|_A = s \cdot (f_2 - m \cdot f_1)$$

Then the contrapositive holds: $[GCD(|t|_A,A) = 1] \Longrightarrow [GCD(t,A) = 1]$.

Suppose now $GCD(|t|_A,A) \neq 1$. Then $GCD(|t|_A,A) = s$, some $s > 1$. Then $A = f_1 \cdot s$, $|t|_A = f_2 \cdot s$.

$$t = |t|_A + mA$$

$$t = f_2 \cdot s + m \cdot f_1 \cdot s$$

$$t = s \cdot (f_2 + m \cdot f_1)$$

$$GCD(t,A) \geq s,\ s \neq 1.$$

Then the contrapositive holds: $[GCD(t,A) = 1] \Longrightarrow [GCD(|t|_A,A) = 1]$.

∎

APPENDIX B

Design Examples

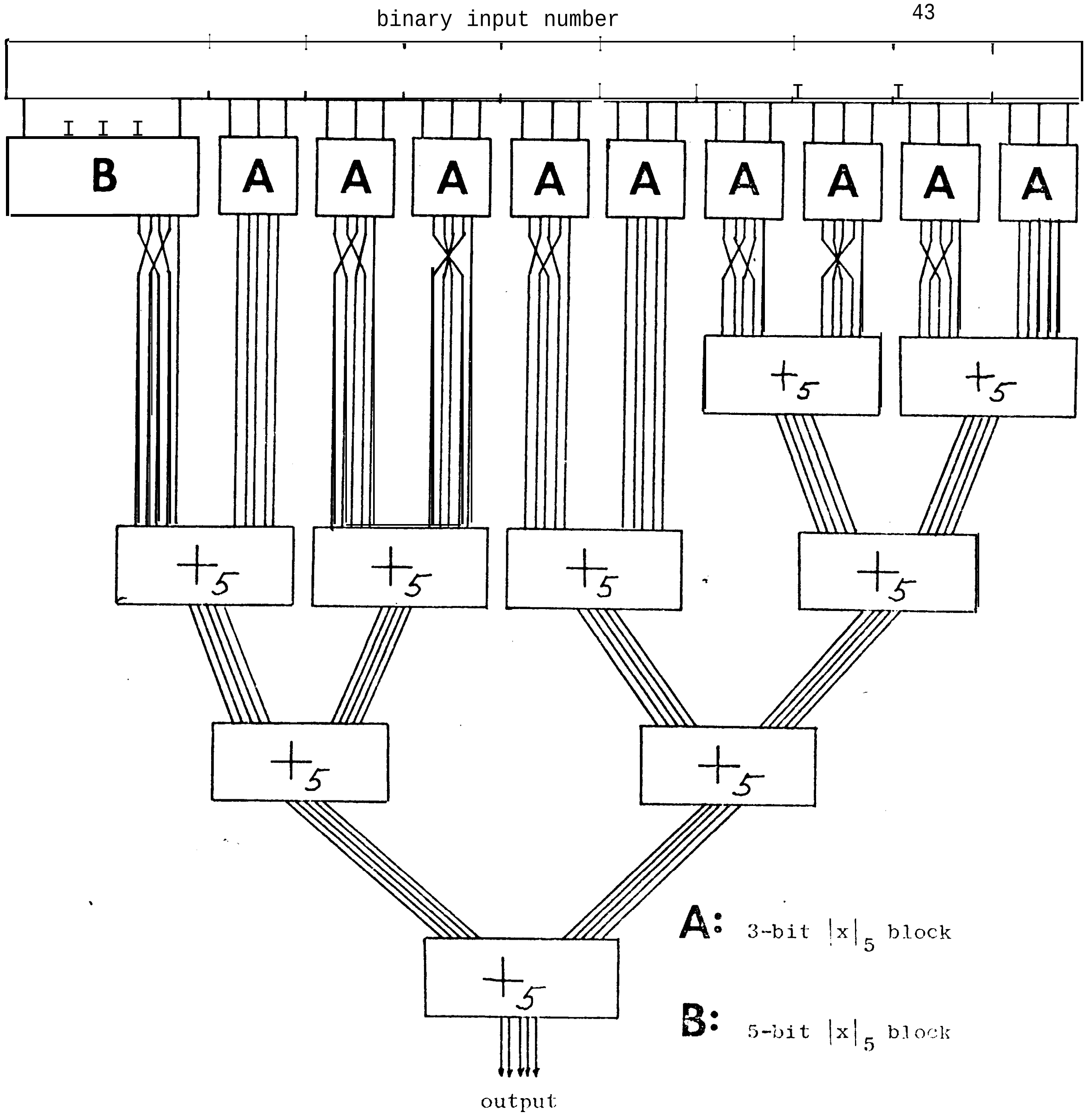


Figure B1. 32 bit wide modulo 5 residue tree.

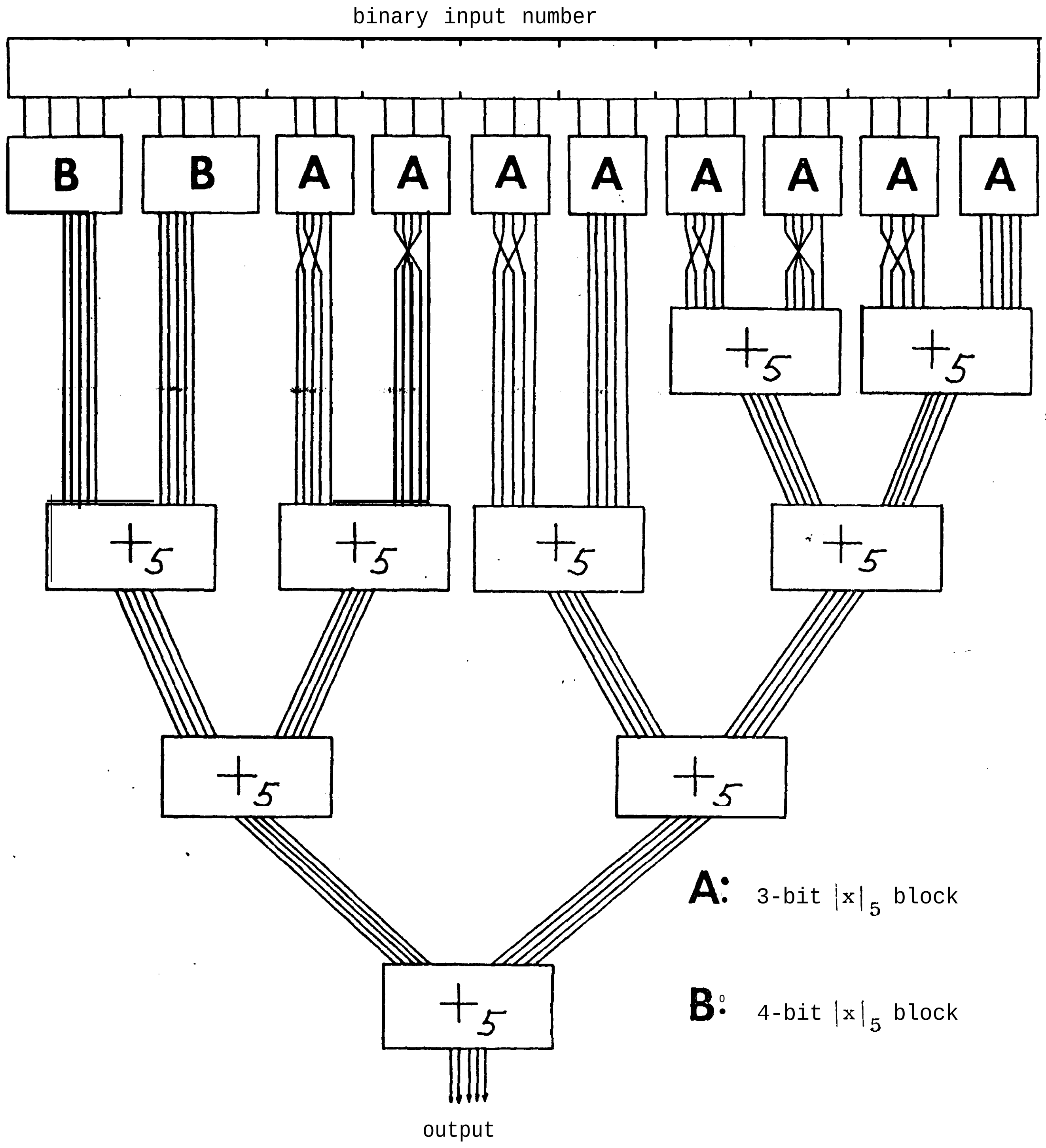
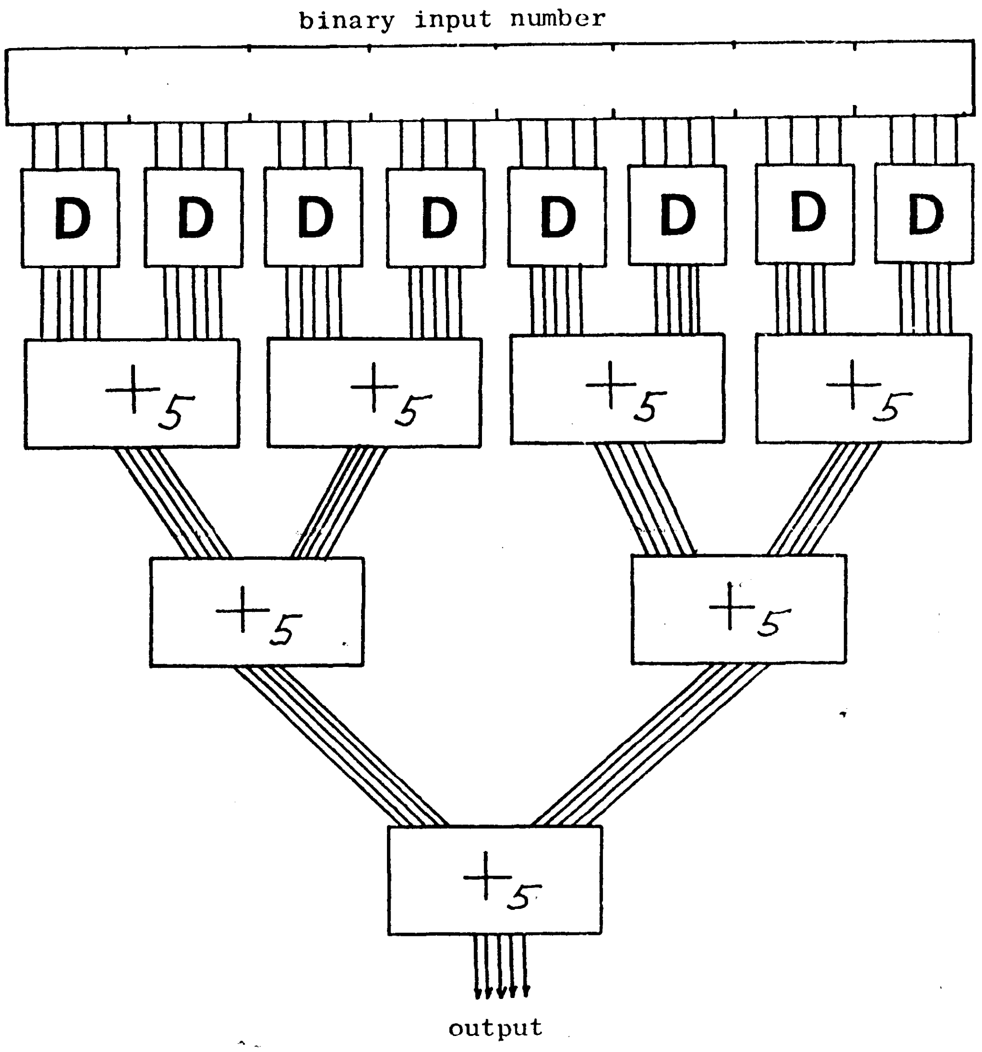


Figure B2. 32 bit wide modulo 5 residue tree.



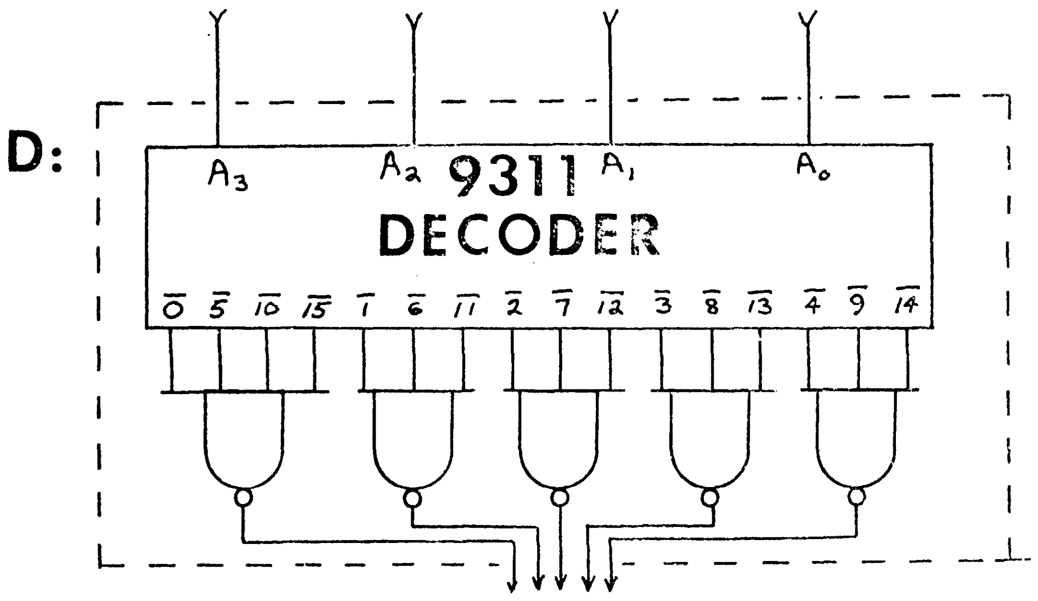
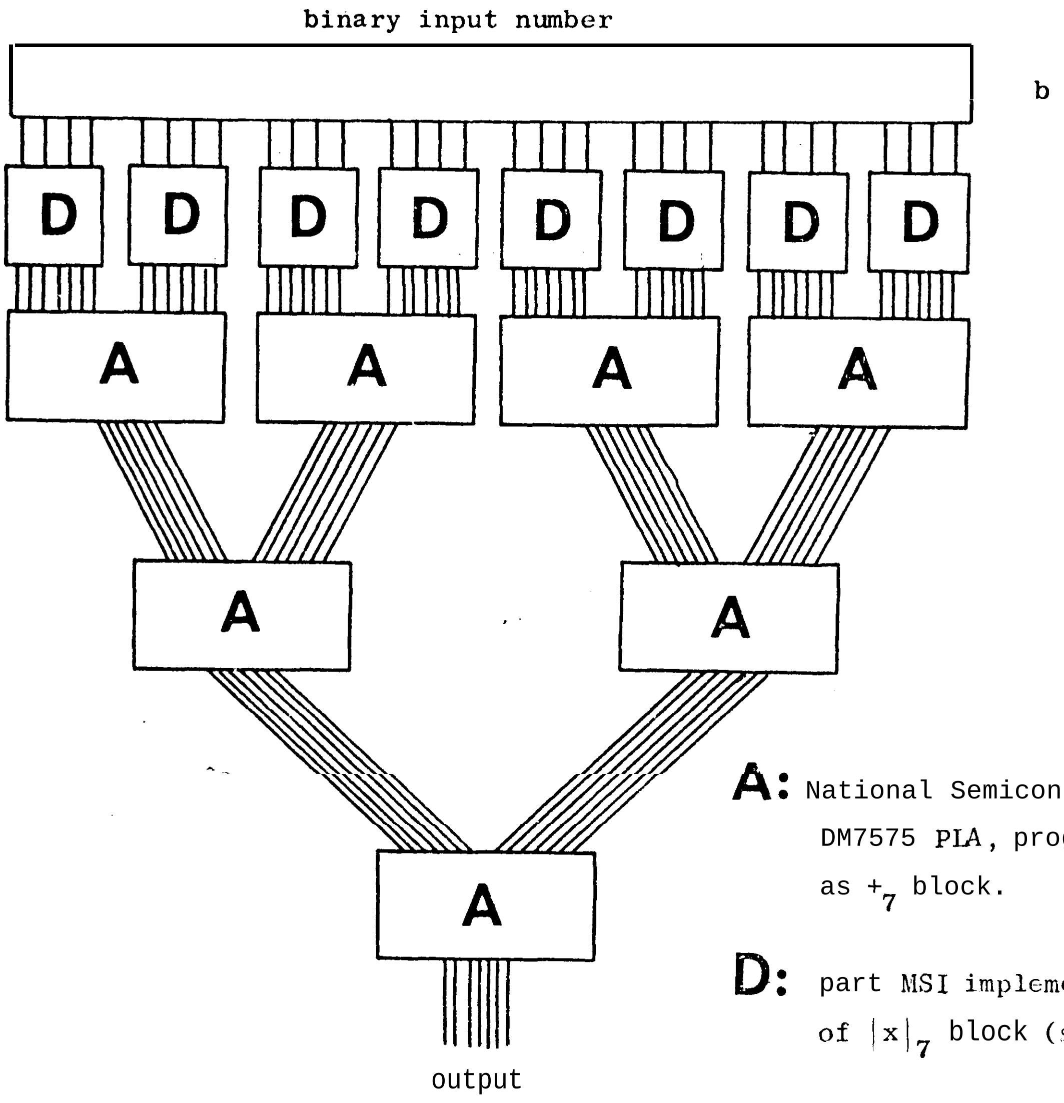


Figure B3. 32 bit wide modulo 5 residue tree using MSI decoders.



A: National Semiconductor-DM7575 PLA, programmed as $+_7$ block.

D: part MSI implementation of $|x|_7$ block (see Fig. B5).

Figure B4. 32 bit wide modulo 7 residue tree using MSI decoders and programmed logic array.

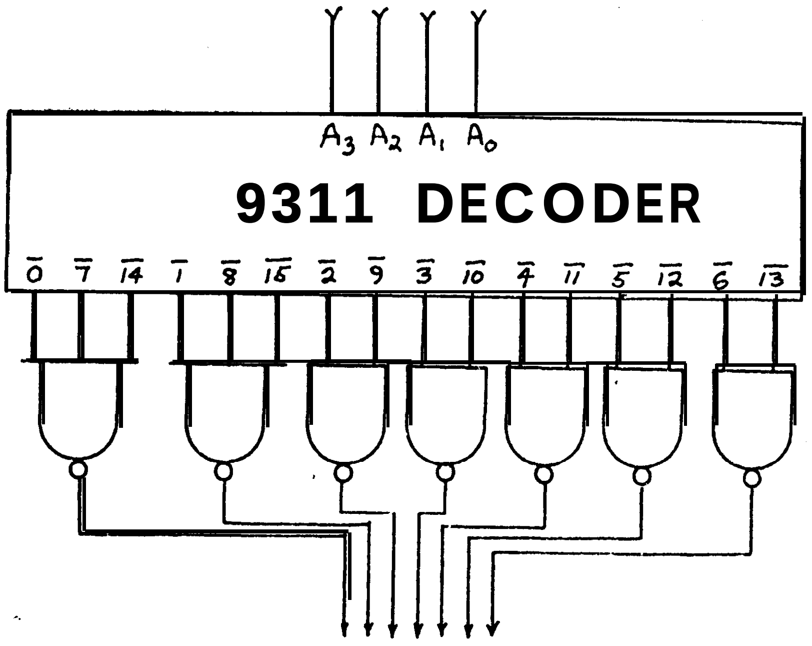

A_3 A_2 A_1 A_0
9311 DECODER
$\bar{0}$ $\bar{7}$ $\overline{14}$ $\bar{1}$ $\bar{8}$ $\overline{15}$ $\bar{2}$ $\bar{9}$ $\bar{3}$ $\overline{10}$ $\bar{4}$ $\overline{11}$ $\bar{5}$ $\overline{12}$ $\bar{6}$ $\overline{13}$

Figure B5. $|x|_7$ block of MSI modulo 7 residue tree.