Rectangular Davio Lattice Structures

Scott Guthridge and Marek Perkowski
Department of Electrical and Computer Engineering
Portland State University
Portland, OR 97207, USA
mperkows@ece.pdx.edu

Abstract

Integrated circuits with a highly regular, repeated structure outperform IC's with complex logic interconnects, in transistor density, gate delay time, and testability. These advantages drive the search for highly regular structures. In [4] a triangular Davio lattice array has been introduced and its testability. A new structure, rectangular Davio lattice array is introduced here that has even more advantages for testability and nano-technology design.

1. Rectangular Lattice Architecture

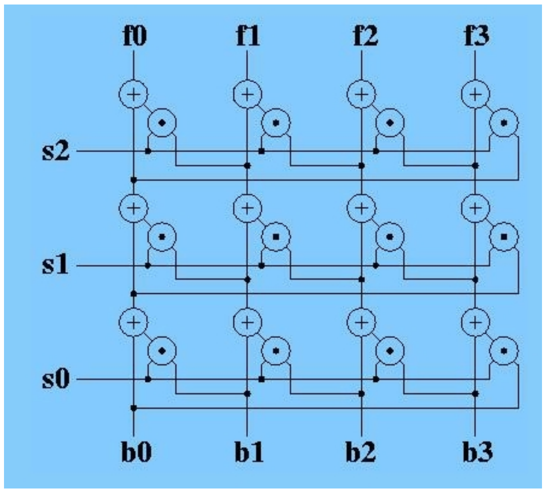


Figure 1. Rectangular easily testable Lattice Architecture using Positive Davio Gates

The rectangular lattice from Figure 1 is an **m by n** array of Davio gates connected much like the triangular structure from [4]. The upper part of the triangle has been filled out, creating a separate output for each column. One further difference is that signal lines from column zero wrap around to the right-most column, giving the structure an overall left-right rotational symmetry around rotations of the the **b** vector. As we'll soon see, an **N** by **N+1** rectangular structure, like the triangular structure, is capable of generating any symmetric function. But the additional outputs the rectangular structure produces significantly enhance both synthesis and testability.

2. Transformation by a Lattice Row

When an **s** input is zero, control inputs are passed upward through the row without modification. When an **s** input is one, the control signals are XOR'ed with a left-rotated copy of the signals. Mathematically:

$\mathbf{b_j' = b_j \oplus b}_{\mathbf{((j+1)\ mod\ cols)}}$

Applying the rule recursively, we can see how the control signals are transformed as they pass upward through multiple rows that have s equal to one:

$\mathbf{b_j' = b_j \oplus b}_{\mathbf{((j+1)\ mod\ cols)}}$
$\mathbf{b_j'' = b_j \oplus [b}_{\mathbf{((j+1)\ mod\ cols)}}\mathbf{]\hat{}2 \oplus b}_{\mathbf{((j+2)\ mod\ cols)}}$
$\mathbf{b_j''' = b_j \oplus [b}_{\mathbf{((j+1)\ mod\ cols)}}\mathbf{]\hat{}3 \oplus [b}_{\mathbf{((j+2)\ mod\ cols)}}\mathbf{]\hat{}3 \oplus b}_{\mathbf{((j+3)\ mod\ cols)}}$

$\mathbf{b_j'''' = b_j \oplus [b_{((j+1)\ mod\ cols)}]\text{^}4 \oplus [b_{((j+2)\ mod\ cols)}]\text{^}6 \oplus [\ b_{((j+3)\ mod\ cols)}]\text{^}4 \oplus b_{((j+4)\ mod\ cols)}}$

...

Symbol **a^b** means **b** times XORing of **a**. These equations are very much like the equations for the triangular structure [4] except that when the number of rows with **s = 1** becomes equal to the number of columns in the lattice, the subscripts wrap around and new terms on the right begin to interfere with existing terms. Consider, for example, a 4x4 lattice. The rightmost term in the **b''''** equation above, $\mathbf{b}_{\mathbf{((j+4)\ mod\ 4)}}$ is the same as the first, $\mathbf{b}_{\mathbf{j}}$. Combining the first and last terms gives us:

$\mathbf{b_j'''' = [\ b_j\ ]\text{^}2 \oplus [\ b_{((j+1)\ mod\ cols)}]\text{^}4 \oplus [\ b_{((j+2)\ mod\ cols)}]\text{^}6 \oplus [\ b_{((j+3)\ mod\ cols)}]\text{^}4}$

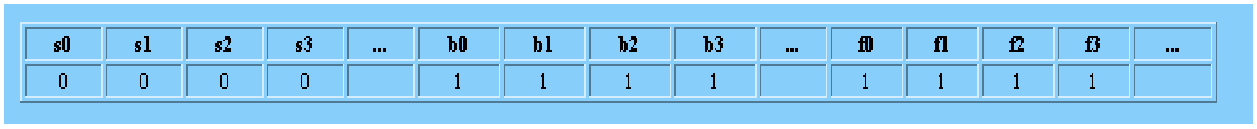
The powers on all terms are even, thus $\mathbf{b_j}''''$ = 0, regardless of the initial **b** vector! Note that this 4x4 lattice is a special case -- not all lattice sizes will go to zero like this. A three-column wide lattice, for example, goes into a three-line repeating pattern.

3. Logic Synthesis

Logic synthesis for the rectangular structure is similar to the triangular structure. We simply equate the $\mathbf{b_j}$ values with the appropriate S(**i**) values and then solve the resulting system of linear equations:

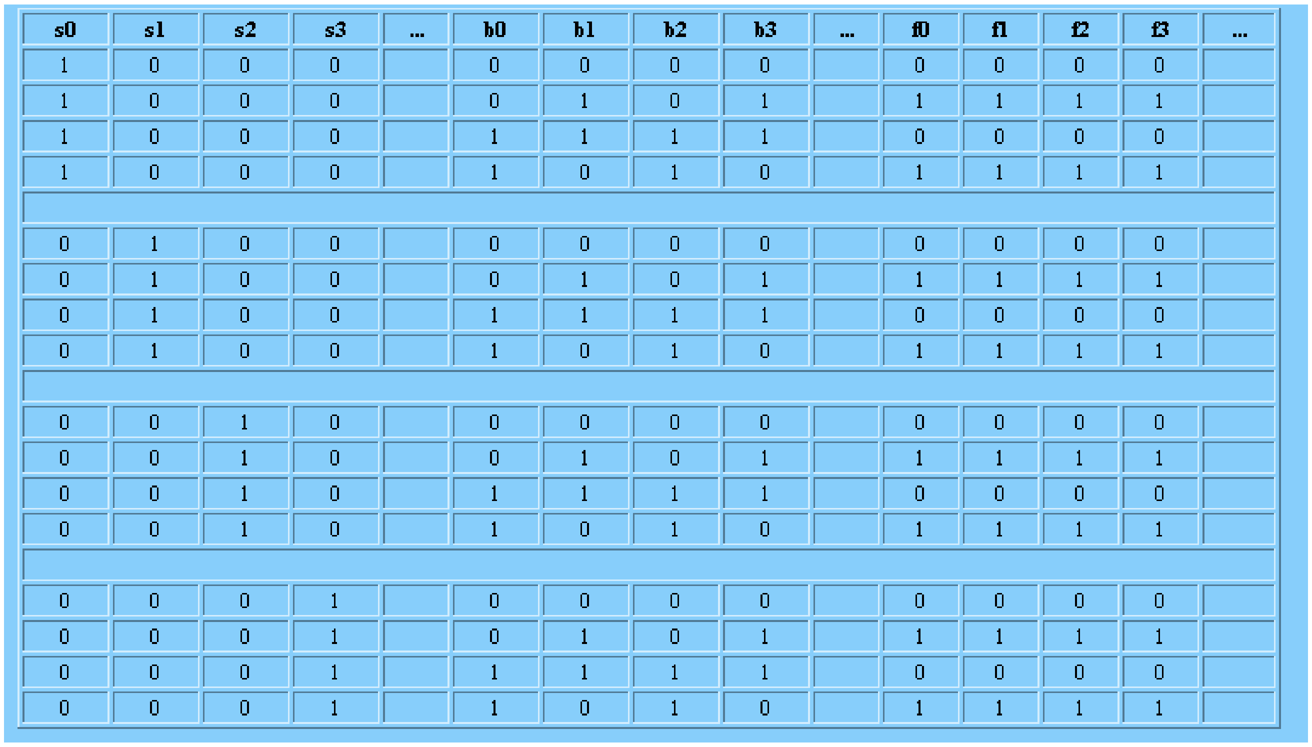
S(0) = bj
S(1) = bj'
S(2) = bj''
S(3) = bj'''
S(4) = bj''''
...

Table 1a Test to find all single stuck-at faults in a rectangular array



s0	s1	s2	s3	...	b0	b1	b2	b3	...	f0	f1	f2	f3	...
0	0	0	0		1	1	1	1		1	1	1	1	

Table 1b Test for the second state of testing, ,used to walk each Davio gate through its universal tests



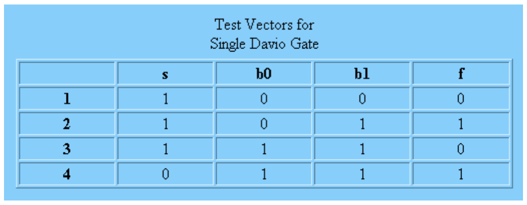
s0	s1	s2	s3	...	b0	b1	b2	b3	...	f0	f1	f2	f3	...
1	0	0	0		0	0	0	0		0	0	0	0	
1	0	0	0		0	1	0	1		1	1	1	1	
1	0	0	0		1	1	1	1		0	0	0	0	
1	0	0	0		1	0	1	0		1	1	1	1	
0	1	0	0		0	0	0	0		0	0	0	0	
0	1	0	0		0	1	0	1		1	1	1	1	
0	1	0	0		1	1	1	1		0	0	0	0	
0	1	0	0		1	0	1	0		1	1	1	1	
0	0	1	0		0	0	0	0		0	0	0	0	
0	0	1	0		0	1	0	1		1	1	1	1	
0	0	1	0		1	1	1	1		0	0	0	0	
0	0	1	0		1	0	1	0		1	1	1	1	
0	0	0	1		0	0	0	0		0	0	0	0	
0	0	0	1		0	1	0	1		1	1	1	1	
0	0	0	1		1	1	1	1		0	0	0	0	
0	0	0	1		1	0	1	0		1	1	1	1	

The synthesis can be done for any output, $\mathbf{f}_j$. The other outputs, however, are dependent on the same **b** values, simply rotated, and so can not be independently synthesized. We'll see later on, though that although these other outputs may not be independent, they can still be useful.

Something that should be noted at this point is that if the number of rows is greater than the number of columns in the lattice, the coefficient matrix in the resulting system of equations may not be of full rank, and thus the desired solution may not exist. For example, in the case of a 4x4 lattice, whether desired or not, S(4) is always zero. The converse statement, however, holds. As long as the number of columns is greater than the number of rows, it's not possible for a given **j** to wrap all the way around, thus it's possible to synthesize any symmetric function over the rows of the lattice. Note that sometimes it's desirable to disable the wrap-around connections from the last column to the first column. When this is done, the modulo operations disappear from the equations above, and the synthesis is like that for the triangular lattice with the one exception that all columns produce an output.

4. Universal Test

It's here that we first see just how valuable these extra output lines are. Instead of a single output for the entire lattice structure, we now have as many outputs as there are columns. These

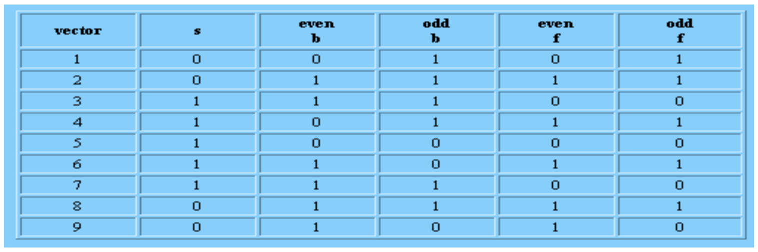


Test Vectors for
Single Davio Gate

	s	b0	b1	f
1	1	0	0	0
2	1	0	1	1
3	1	1	1	0
4	0	1	1	1

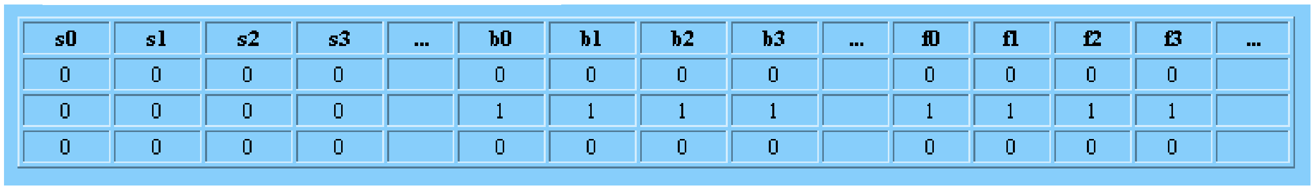
Table 2

Table 3. Example of covering both even and odd columns



vector	s	even b	odd b	even f	odd f
1	0	0	1	0	1
2	0	1	1	1	1
3	1	1	1	0	0
4	1	0	1	1	1
5	1	0	0	0	0
6	1	1	0	1	1
7	1	1	1	0	0
8	0	1	1	1	1
9	0	1	0	1	0

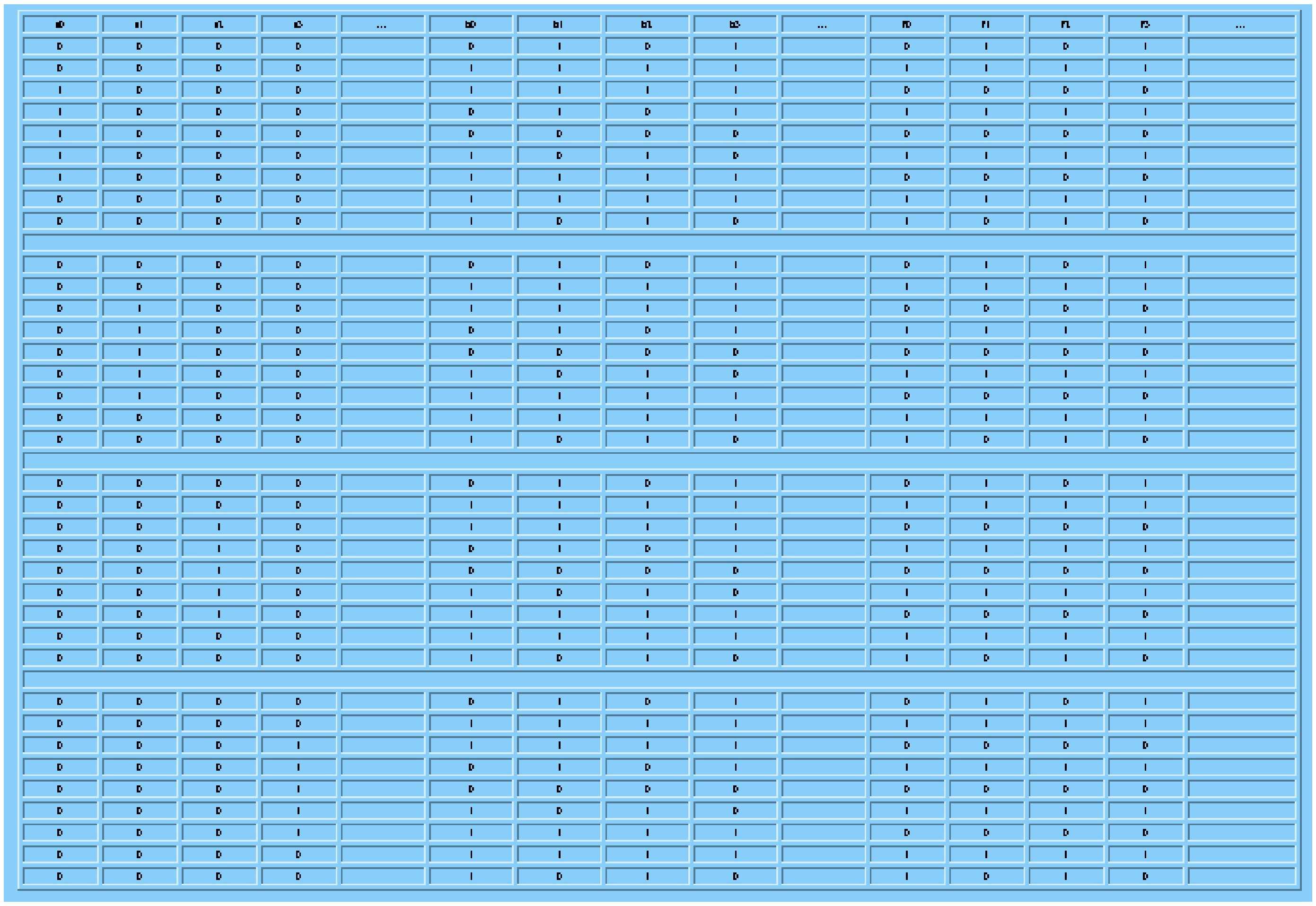
Table 4a, First stage of improved test



s0	s1	s2	s3	...	b0	b1	b2	b3	...	f0	f1	f2	f3	...
0	0	0	0		0	0	0	0		0	0	0	0	
0	0	0	0		1	1	1	1		1	1	1	1	
0	0	0	0		0	0	0	0		0	0	0	0	

outputs greatly increase observability of the structure during testing. In the rectangular structure, we can test entire rows at a time (a strategy that was only partially successful in the triangular lattice case). To find all single stuck-at faults, first apply this test, shown in Table 1(a). The first test vector tests for stuck-at-0 faults in the lattice columns and stuck-at-1 faults on the AND gate inputs. Next, walk ones on the s inputs, cycling through the four b test vectors on each row as shown in Table 1(b) These test vectors walk each Davio gate through its universal test shown in Table 2. ***Caution****:* when the number of columns in the lattice is odd, we can run into problems on the last column. It will still have a value of 1 on the initial test vector, but for all the other test vectors, the last column will always be zero.

Table 4b. Second stage of improved test



s0	s1	s2	s3	...	b0	b1	b2	b3	...	r0	r1	r2	r3	...
0	0	0	0		0	1	0	1		0	1	0	1	
0	0	0	0		1	1	1	1		1	1	1	1	
1	0	0	0		1	1	1	1		0	0	0	0	
1	0	0	0		0	1	0	1		1	1	1	1	
1	0	0	0		0	0	0	0		0	0	0	0	
1	0	0	0		1	0	1	0		1	1	1	1	
1	0	0	0		1	1	1	1		0	0	0	0	
0	0	0	0		1	1	1	1		1	1	1	1	
0	0	0	0		1	0	1	0		1	0	1	0	
0	0	0	0		0	1	0	1		0	1	0	1	
0	0	0	0		1	1	1	1		1	1	1	1	
0	1	0	0		1	1	1	1		0	0	0	0	
0	1	0	0		0	1	0	1		1	1	1	1	
0	1	0	0		0	0	0	0		0	0	0	0	
0	1	0	0		1	0	1	0		1	1	1	1	
0	1	0	0		1	1	1	1		0	0	0	0	
0	0	0	0		1	1	1	1		1	1	1	1	
0	0	0	0		1	0	1	0		1	0	1	0	
0	0	0	0		0	1	0	1		0	1	0	1	
0	0	0	0		1	1	1	1		1	1	1	1	
0	0	1	0		1	1	1	1		0	0	0	0	
0	0	1	0		0	1	0	1		1	1	1	1	
0	0	1	0		0	0	0	0		0	0	0	0	
0	0	1	0		1	0	1	0		1	1	1	1	
0	0	1	0		1	1	1	1		0	0	0	0	
0	0	0	0		1	1	1	1		1	1	1	1	
0	0	0	0		1	0	1	0		1	0	1	0	
0	0	0	0		0	1	0	1		0	1	0	1	
0	0	0	0		1	1	1	1		1	1	1	1	
0	0	0	1		1	1	1	1		0	0	0	0	
0	0	0	1		0	1	0	1		1	1	1	1	
0	0	0	1		0	0	0	0		0	0	0	0	
0	0	0	1		1	0	1	0		1	1	1	1	
0	0	0	1		1	1	1	1		0	0	0	0	
0	0	0	0		1	1	1	1		1	1	1	1	
0	0	0	0		1	0	1	0		1	0	1	0	

5. Improved Test

The high observability of the rectangular lattice structure makes it pretty easy to extend our test above to cover stuck-open faults in the Davio gates and bridging faults between adjacent columns. Instead of using only 4 test vectors per row, we use 9. These vectors test every transistor in a typical CMOS implementation of the Davio gate for stuck-at and stuck-open faults, and cover both the even and odd columns in the lattice from Table 3. The initial test above isn't actually needed in this approach, but it's easy to do and gives a quick indication of whether there are major problems in the lattice or not. Let's keep it, but force the **b** values up then back down to detect any stuck-open faults in the columns, shown in Table 4(a). We still walk ones on the **s** inputs, but notice that all s inputs must be zero for the first two and last two rows of the test vector blocks, Table 4(b).

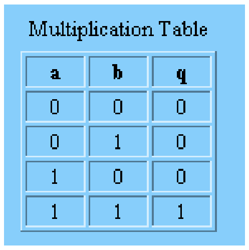
Caution: As with the previous testing strategy, if the lattice has an odd number of columns, the right-most output behaves differently from the others. Here, the last output will have the expected value from the table when all **s** inputs are zero and will always be zero when an **s** input is on.

Summarizing: Due to the greater observability provided by the per-column outputs of the rectangular structure, the rectangular structure is much easier to test. The testing method just given detects all single stuck-at faults, all stuck-open faults and all bridging faults between adjacent columns. Further, the number of test vectors required grows linearly with the number of table rows. So, where a 4x4 lattice requires 36 vectors, the 40x40 only requires 360. Compare this to the quadratic growth algorithm developed for the triangular structure in [4]. For a 4x5 triangular lattice, that test requires 50 vectors. For a 40x41 lattice, however, it requires 4100 vectors! The rectangular structure's additional outputs are also quite useful for synthesis.

6. Linear Transformations in Galois Field(2)

This section covers matrix methods for analysis and synthesis of the Davio gate lattice structures. All matrix operations here are performed in the Galois Field(2) system, in which multiplication is like AND (Table 5) and addition is like XOR (Table 6):

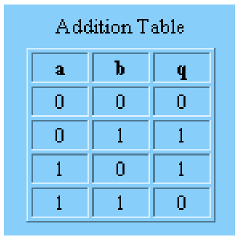
Table 5



Multiplication Table

a	b	q
0	0	0
0	1	0
1	0	0
1	1	1

Table 6



Addition Table

a	b	q
0	0	0
0	1	1
1	0	1
1	1	0

6.1 Row Transformation Matrix, R

To model the transformation of the control signals by a row with **s** = 1, we define the row transformation matrix, **R**. Using the notation from [4] and previous sections, each right-side multiplication of **b** by **R** adds a prime to **b**:

$\mathbf{b' = b * R \; b'' = b' * R}$

...

So, if the lattice has **i** **s** inputs set to one, or in other words, if **s** is in S(**i**), the output of the lattice is related to the control inputs, b by the relation:

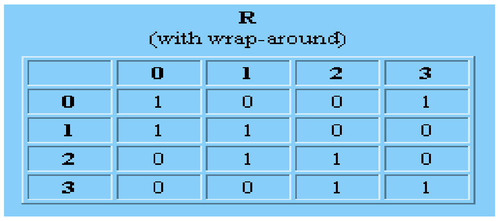
$\mathbf{f = b * R^{i}}$

The **R** matrix has two forms depending on whether or not the last column signals wrap-around to the first column. Examples are shown in Tables 7(a) and 7(b).

The **R** matrix is always square with dimensions based on the number of columns in the lattice. The major diagonal always contains ones, representing signals passing upward through the XOR part of the Davio gates. Off-diagonal terms represent the signals from the neighboring column to the right feeding into the AND portion of the Davio gates. The only difference between the 7(a) and 7(b) forms is the value of the upper-right corner square. When this value is one, it represents a value wrapping-around from the first column to the last. When zero, the AND portion of last column Davio gates always receive zero, thus the last column Davio gates become degenerate - they simply pass the $\mathbf{b}_{\mathbf{(cols-1)}}$ control input through without modification.

Another difference between the two forms is that the 7(a) form is never invertible while the 7(b) form always is. This follows from the fact that in the 7(a) form, any row of the matrix can be calculated by adding all the other rows together, thus each row is a linear combination of the other rows. In the 7(b) form, we can always diagonalize the matrix by starting at the top and adding each row to the row immediately below. This difference, though interesting, probably isn't extremely significant. It only seems to matter in the case that we have an **f** vector and the number of **s** inputs set, and want to calculate the **b** vector.

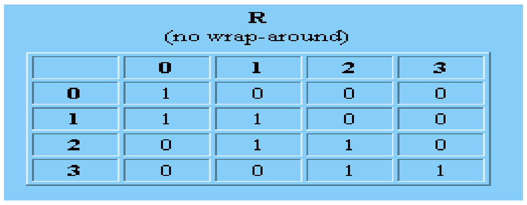
Table 7(a)



R
(with wrap-around)

	0	1	2	3
0	1	0	0	1
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1

Table 7(b)



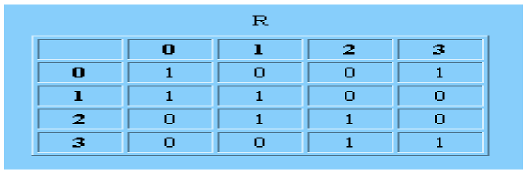
R
(no wrap-around)

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1

6.2 Powers of the R Matrix

In [4], it was mentioned that it's possible to implement asymmetric functions using the lattice structure if an external input is repeated on the **s** inputs of two or more rows. A transformation by **i** repeated rows can be described by $\mathbf{R}^{\mathbf{i}}$. This section explores powers of the **R** matrix and develops a method to simplify repeated inputs. Let's look at a few powers of the Table 7(a) **R** matrix, Table 8(a), Table 8(b), Table 8(c), Table 8(d).

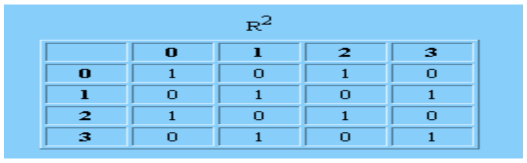
Table 8(a) Matrix R



R

	0	1	2	3
0	1	0	0	1
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1

Table 8(b) Matrix R^2

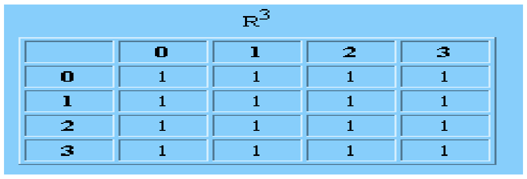


R^2

	0	1	2	3
0	1	0	1	0
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1

Notice that in the $\mathbf{R}^2$ case, the matrix still has the major diagonal of ones representing the control signals passing upward through the lattice, but instead of the off-diagonal terms representing a rotation from the neighboring column, it reflects a rotation from the column one beyond the neighbor. Another way of looking at it is that the column immediately to the right is XOR'ed into the current column twice, canceling itself. Meanwhile, the column two to the right is XOR'ed in once. Look now at the $\mathbf{R}^4$ case. Here, the column two to the right is XOR'ed into the current column twice while the column four to the right is XOR'ed in once. In this case, though, the column four to the right is the current column itself and everything cancels and the row always produces zeros.

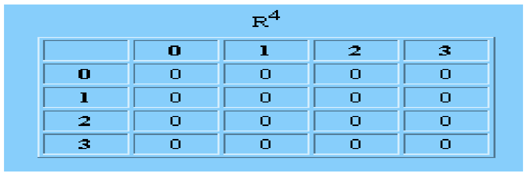
Table 8(c) Matrix R^3



R^3

	0	1	2	3
0	1	1	1	1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1

Table 8(d) Matrix R^4



R^4

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

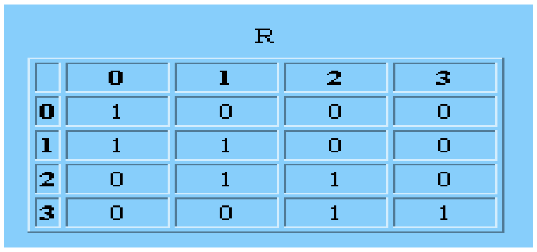
It should be start to become clear that for any power of **R** in the form

$$R^{2^k}$$

, **k** integer ≥ 0, the repeated line transformation is equivalent to rewiring the inter-column connections to jump every $\mathbf{2}^{\mathbf{k}}$ rows.

Taking **R** to the third power, however, is not equivalent to rewiring the columns to jump every 3 rows. Here, we don't have the luxury of the multiple transformations canceling out the intermediate terms. $\mathbf{R}^3$ could be implemented, however, by connecting the external input both to a $\mathbf{R}^2$ row and a regular row.

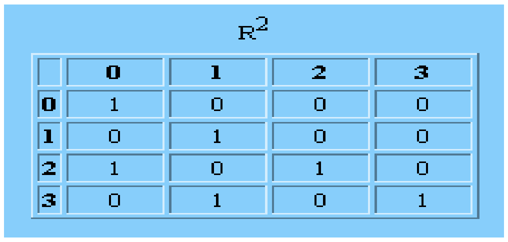
Table 9(a). Matrix R



R

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1

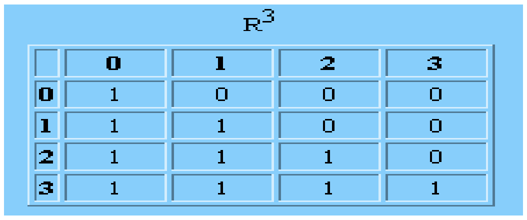
Table 9(b) Matrix R^2



R^2

	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	1	0	1	0
3	0	1	0	1

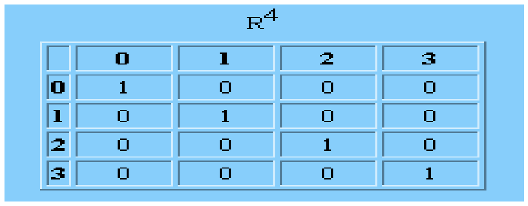
Table 9(c) Matrix R^3



R^3

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	1	1	0
3	1	1	1	1

Table 9(d) Matrix R^4



R^4

	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

This property suggests an enhancement to the rectangular lattice structure. The lattice should contain several regular rows, then a few $\mathbf{R}^2$ rows, a few $\mathbf{R}^4$ rows, a few $\mathbf{R}^8$, etc. Then, for example, if a synthesis problem required a multiple of say, 9 rows, we could implement it by wiring a regular row with an R8 row, using the size and gate delays of only two rows!

What about powers of the table 7(b) form? They are shown in Tables 9(a), 9(b), 9(c) and 9(d).

As we can see, the same canceling of intermediate terms happens here as well, so the optimization applies equally to this case.

6.3 (b -> S) Transformation Matrix, T

The **R** matrix gives us a nice tool to determine the output function **f** given **b** and **s**, but it doesn't really give us a way to compute the **b** vector needed to realize a given desired symmetric function. This section develops a transformation matrix that converts a given **b** vector to its corresponding symmetric function, **S**, and the inverse transformation than can be used to find the **b** necessary to implement a specific symmetric function, **S**.

The **T** transformation and, if it exists, the $\mathbf{T}^{-1}$ transformation, are used as follows:

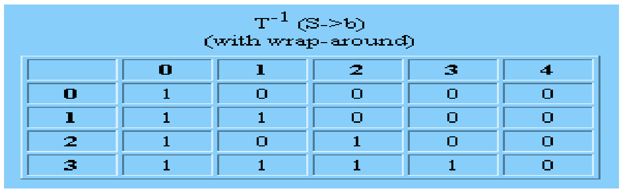
$$\mathbf{S} = \mathbf{T} * \mathbf{b}^T$$
$$\mathbf{b} = [\mathbf{T}^{-1} * \mathbf{S}]^T$$

...

where the superscripted, **T**, indicates the transpose operation and the **S** vector describes the symmetric function appearing on output $\mathbf{f_0}$. Note that **b** is a row vector while **S** is a column vector.

Like **R**, the **T** matrix has a regular pattern and is easily synthesized with a computer. For the wrap-around and no-wrap-around lattice architectures [Figure 7(a) and 7(b)], respectively, the **T** matrix has the forms from Tables 10(a) and 10(b).

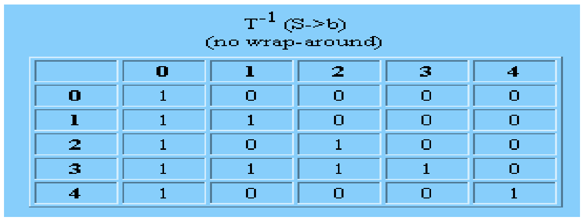
Table 10(a)



T^{-1} (S->b)
(with wrap-around)

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	0	1	0	0
3	1	1	1	1	0

Table 10(b)



T^{-1} (S->b)
(no wrap-around)

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	0	1	0	0
3	1	1	1	1	0
4	1	0	0	0	1

Following directly from the math given in [4] and section 3, the **T** matrix can be constructed as follows:

Construction of **T** (with wrap-around)

$\mathbf{T}_{0,0} = \mathbf{1}$

$\mathbf{T}_{0,j} = \mathbf{0}, \quad \mathbf{1 <= j < cols}$

$\mathbf{T}_{i,0} = \mathbf{T}_{(i-1),(cols-1)} \oplus \mathbf{T}_{(i-1),0}, \quad \mathbf{1 <= i <= rows, 1 <= j < cols}$

$\mathbf{T}_{i,j} = \mathbf{T}_{(i-1),(j-1)} \oplus \mathbf{T}_{(i-1),j}, \mathbf{1 <= i <= rows, 1 <= j < cols}$

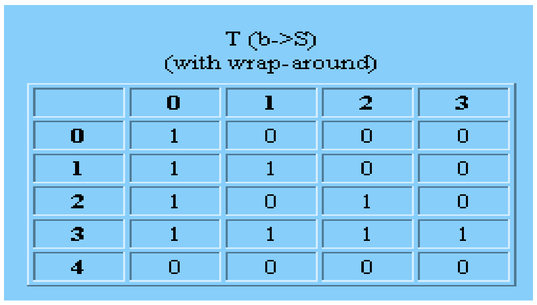
Construction of **T** (no wrap-around)

$\mathbf{T_{0,0} = 1}$
$\mathbf{T_{0,j} = 0, \quad 1 <= j < cols}$
$\mathbf{T_{i,0} = T_{(i-1),0}, \; 1 <= i <= rows}$
$\mathbf{T_{i,j} = T_{(i-1),(j-1)} \oplus T_{(i-1),j}, \; 1 <= i <= rows, \; 1 <= j < cols}$

6.4 Synthesis Using T^{-1}

The **T** matrix has dimension (**rows**+1) x **cols**. Because **T** is not necessarily square, the inverse matrix, $\mathbf{T}^{-1}$ must be computed using the pseudo-inverse method. Even then, if **cols <= rows**, the pseudo- inverse may not exist. For example, in the 4x4 matrix, $\mathbf{T}^{-1}$ exists for the wrap-around architecture, Table 11(a), but does not exist for the no-wrap architecture. If we increase the number of columns by one, making a 4x5 matrix, then the inverse exists for the no-wrap case, Table 11(b).

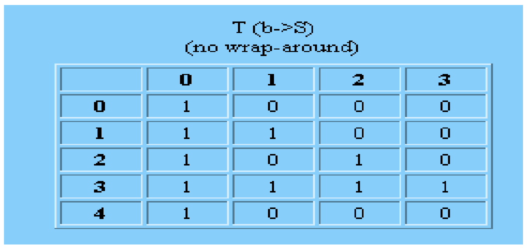
Table 11(a)



T (b->S)
(with wrap-around)

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	0	1	0
3	1	1	1	1
4	0	0	0	0

Table 11(b)



T (b->S)
(no wrap-around)

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	0	1	0
3	1	1	1	1
4	1	0	0	0

Because we are using the pseudo-inverse, even when $\mathbf{T}^{-1}$ exists, there's still no guarantee that the **b** vector it produces generates the desired symmetric function. Therefore, it is important to multiply the generated **b** back by **T** as a check. For example, suppose we want to implement the AND function using the 4x4 wrap-around lattice. The desired **S** vector on $\mathbf{f_0}$ is:

	S
S(0):	**[0]**
S(1):	**[0]**
S(2):	**[0]**
S(3):	**[0]**
S(4):	**[1]**

Going back to Table 10(a), we can already see that we're in trouble, because independent of b, an input of S(4) always produces zero in the 4x4 wrap-around structure! Let's continue with the synthesis, anyway to see what happens.

Multiplying **S** by $\mathbf{T}^{-1}$ [Table 11(a)] yields the a **b** vector of **[0 0 0 0].** Multiplying back by **T** as a check shows us that we've in fact implemented the "zero" function:

	S
S(0):	**[0]**
S(1):	**[0]**
S(2):	**[0]**
S(3):	**[0]**
S(4):	**[0]**

The problem is that the 4x4 wrap-around lattice is not capable of implementing 4-input AND. If we had chosen a value of **cols** greater than **rows**, 4x5, for example, or if we had tried to implement a function where S(4) = 0, it would have worked. For example, in the case of the 4x5 lattice, we would have calculated $\mathbf{b} = [\ 0\ 0\ 0\ 0\ 1\ ]$, which does produce the AND function on $\mathbf{f_0}$. Or, sticking with the 4x4 lattice, we could have implemented a function where S(4) = 0, say, 4-input XOR:

	S
S(0):	**[0]**
S(1):	**[1]**
S(2):	**[0]**
S(3):	**[1]**
S(4):	**[0]**

Here, multiplication by $\mathbf{T}^{-1}$ gives us $\mathbf{b} = [\ 0\ 1\ 0\ 0\ 0\ ]$ and the resulting function at $\mathbf{f_0}$ is indeed XOR.

For the case of lattice structures in which the number of columns is greater than the number of rows by one, we can prove that $\mathbf{T}^{-1}$ exists, and that the $\mathbf{T}^{-1}$ synthesis method works for any arbitrary symmetric function. Looking at the way the **T** matrix is constructed, we can see that with or without wrap-around, we get a square, lower triangular matrix with a diagonal of all ones. This matrix is of full rank and therefore has an inverse, [3]. Therefore, there is a 1-1 mapping between **b** and **S**. For any **S**, there is a corresponding **b** such that $\mathbf{S} = \mathbf{T} * \mathbf{b}^{\mathbf{T}}$.

6.5 Synthesis of Multiple Symmetric Functions

Section 6.4 presented a method for synthesizing arbitrary symmetric functions at lattice output $\mathbf{f_0}$. From section 6.1, we know that the other lattice outputs are based on the same control vector, **b** and side inputs $\mathbf{s_i}$**.** What do we get on these other outputs? How many symmetric functions can we create using all outputs?

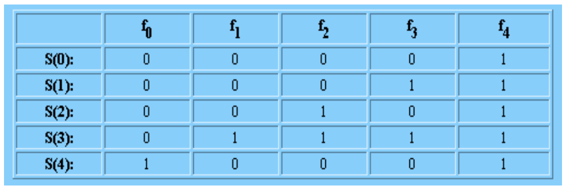
To answer the first question, we can calculate $\mathbf{f_j}$ using either the **T** or **R** matrices. The wrap-around lattice structure has rotational symmetry about the columns, so we can calculate the **S** vector at any output, $\mathbf{f_j}$**,** by rotating the elements of **b** by **j** elements to the left and multiplying by **T**.

A more general technique, however, is to use the **R** matrix to build a matrix of the **S** vectors for all columns, the **f** matrix. This approach is more general because it works on both the wrap-around and no-wrap variants, and it calculates all outputs at the same time.

We construct the **f** matrix as follows. The zero-th row of the **f** matrix is the **b** vector calculated for output $\mathbf{f_0}$. Subsequent rows, **i**, in the **f** matrix are calculated by repeated multiplication by **R**. Stated mathematically, row **i** is $\mathbf{b} * \mathbf{R}^{\mathbf{i}}$. Thus the rows of **f** represent the number of **s** inputs asserted, the **i** in S(**i**), while the columns of **f** represent lattice column outputs, $\mathbf{f_j}$.

Using the 4x5 lattice AND example above, **b** was calculated as **[0 0 0 0 1].** The corresponding **f matrix** is shown in Table 12.

Table 12. The f matrix for the example from section 6.5



	f_0	f_1	f_2	f_3	f_4
S(0):	0	0	0	0	1
S(1):	0	0	0	1	1
S(2):	0	0	1	0	1
S(3):	0	1	1	1	1
S(4):	1	0	0	0	1

The function at the first column is the one we chose. The 4x5 lattice gives us four other functions that we don't get to choose, for free:

$\mathbf{f_0}$ is the 4-input AND function we synthesized.
$\mathbf{f_1}$ is symmetric function S(3).

$\mathbf{f_2}$ is symmetric function S(2)+S(3).
$\mathbf{f_3}$ is 4-input XOR.
$\mathbf{f_4}$ is the function, TRUE.

Of these, at least AND and XOR are probably useful directly. But much more significantly, we can see from the lower-triangular structure of this matrix that it has full rank, meaning that it forms a basis for ALL (32) symmetric functions in 4 variables! To realize any of these functions, all we need to do is selectively choose a subset of the column outputs and combine them in an XOR collecting plane.

Many, but not all **b** vector choices result in an f matrix that forms a basis for all symmetric functions. The control vector, **b = [0 0 0 ... 0 1],** however, used in conjunction with an **n x (n+1)** lattice structure always works. [Proof is left to the reader.]

The next section is a web-based program that generates **R, T, f** matrices, and computes **b** vectors to implement arbitrary symmetric functions.

7. Davio Lattice Analysis/Synthesis Program

A program to aid in the analysis and synthesis of symmetric logic functions implemented by a lattice of Davio gates has been developed [5]. The program is fully interactive and can be used through Internet.

8. Example

Figure 2 illustrates a 4-bit full adder implemented using the rectangular lattice structure. Several optimizations have been done to arrive at this solution. Let's start at the beginning and work our way through all steps to illustrate the entire process. First, both the carry and sum outputs of a one-bit full adder are symmetric functions of the three inputs. When we try to extend the adder to two or more bits, we no longer have a purely symmetric function of all inputs. We can fix this by connecting a single external input to two or more lattice rows. For the adder, the process is really straightforward: connect the **x0, y0,** and cin inputs directly to lattice rows. Connect **x1** and **y1** each to two rows, **x2** and **y2** each to four rows, and finally **x3** and **y3** each to eight rows. At this point, we have a fully symmetric function in 31 inputs representing our adder.

The desired functions are the columns of this matrix, the rows of which are simply the numbers from 0 to 31 in binary, as shown in Table 13. Using the program from [5], we can calculate the **b** vectors necessary to implement any of these functions. To be confident that $\mathbf{T}^{-1}$ exists, we have to set the number of columns to one more than the number of rows. We have 31 rows, so we'll need 32 columns. Using the wrap-around structure, the results are:

c_3: b = [<16 zeros>1<15 zeros>]
s_3: b = [<8 zeros>1<23 zeros>]
s_2: b = [<4 zeros>1<27 zeros>]
s_1: b = [<2 zeros>1<29 zeros>]
s_0: b = [<1 zero>1<30 zeros>]

These results couldn't be much more convenient -- all desired outputs are simply rotations of the same **b** vector. Because of the left-right symmetry of the wrap-around lattice, any **b** vector containing a single 1 in an arbitrary position generates all desired outputs! If we line up the 1's vertically in the five computed b vectors above, and left-justify the result, i.e., we use the **b** for output $\mathbf{c_3}$, then the lattice outputs have the following assignments:

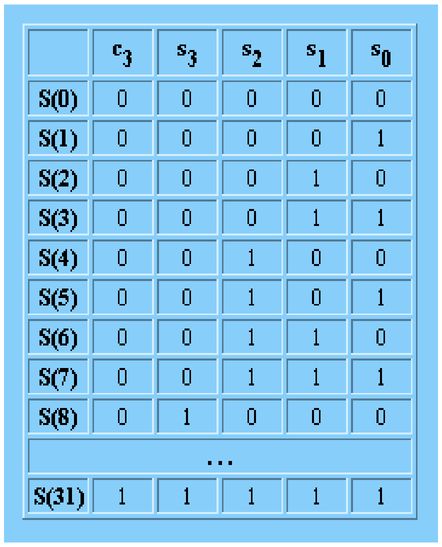
$\mathbf{c_3 = f_0}$
$\mathbf{s_3 = f_8}$
$\mathbf{s_2 = f_{12}}$
$\mathbf{s_1 = f_{14}}$

$\mathbf{s_0 = f_{15}}$

Looking at the whole lattice, we can see that in order for column 17 to rotate anything other than zeros into column 16, it would have to rotate the one at $\mathbf{b_{16}}$ 16 columns left, wrap it from the left side of the lattice to the right and then shift another 16 places. This would require an input of S(32), but we only have 31 **s** inputs. Therefore, it's not possible for column 17 to rotate anything other than a column of zeros into column 16. In fact, the only purpose columns 17-32 are serving is to get rid of any data that's wrapping around from column 0. It should be clear now that if we disable the left-right wrap-around in the lattice, we can eliminate columns 17-32! At this point, we have a 31 x 17 lattice without wrap-around, and a control vector, b, of [<16 zeros> 1]. In [4] we developed the theory behind an optimization that involves collapsing powers of two rows that share a repeated s input into just one row. If we modify our lattice such that it has three ordinary rows, two $\mathbf{R}^2$ rows, two $\mathbf{R}^4$ rows and two $\mathbf{R}^8$ rows, then we can make it so that each external input connects to just one lattice row, and the total number of rows required is equal to the number of inputs, 9.

Notice that the $\mathbf{R}^8$ rows are beginning to be a bit crowded with interconnect wires, which works somewhat against our argument for using the lattice structure in the first place. But up to about $\mathbf{R}^8$, the interconnect wires are still relatively short and the structure is still very regular, so this seems like a good optimization up to at least this power of **R**. The final 9 x 17 lattice design is shown in Figure 2.

Table 13. To adder example: the rows are simply the numbers from 0 to 31 in binary.



	c_3	s_3	s_2	s_1	s_0
S(0)	0	0	0	0	0
S(1)	0	0	0	0	1
S(2)	0	0	0	1	0
S(3)	0	0	0	1	1
S(4)	0	0	1	0	0
S(5)	0	0	1	0	1
S(6)	0	0	1	1	0
S(7)	0	0	1	1	1
S(8)	0	1	0	0	0
...					
S(31)	1	1	1	1	1

9. Conclusions

We presented advantages of rectangular Davio lattices as compared to triangular Davio lattices from [4]. Look to [4] for a discussion of background, applications and complete bibliography, and to [5] for all programs used in reported research. These programs find also applications also to other regular structures with XOR gates, especially in reversible logic. Our future research is on extensions of the presented methods to reversible logic and Single Transistor and Quantum Dot circuits.

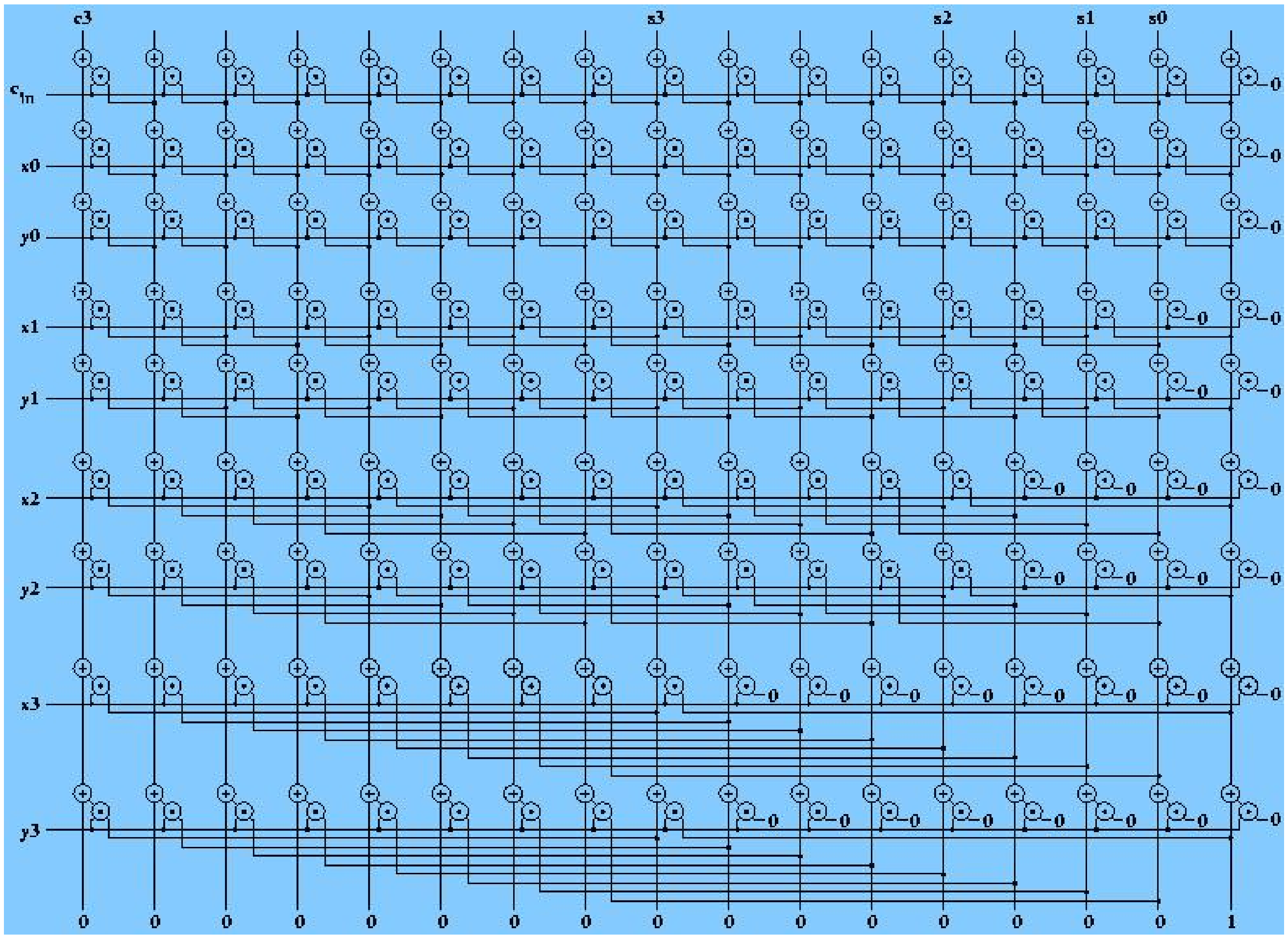


Figure 2. Four Bit Adder with Carry

10. References

[1] S.-M. Kang and Y. Leblebici, CMOS Digital Integrated Circuits, *2nd ed: McGraw Hill*, 1999.
[2] P. Lindgren, R. Drechsler, B. Becker, Synthesis of Pseudo Kronnecker Lattice Diagrams.
[3] G. Strang, Linear Algebra and its Applications, *3rd ed: Harcourt*, 1988, pp. 1-116.
[4] S. Guthridge and M. Perkowski, Test Generation for Triangular Davio Lattice Structures, *Proceedings of this symposium*.
[5] S. Guthridge. WWW Page
http://063-170214108.dslnorthwest.net/scooter/ece510td/project/index.html