Interconnect Test Pattern Generation Algorithm For Meeting Device and Global SSO Limits With Safe Initial Vectors

Kendrick Baker
Raytheon Company
Plano, TX 75023
k-baker@raytheon.com

Mehrdad Nourani
Center for Integrated Circuits & Systems
The University of Texas at Dallas
Richardson, TX 75083
nourani@utdallas.edu

ABSTRACT

This paper presents a method of creating a true/complement pattern set of optimal size that satisfies simultaneous switching limit constraints. This method is an improvement over previous methods in that it has better runtime characteristics as well as the ability to handle additional scenarios. It can produce safe initial vectors, produce vectors that consider switching limits by device rather than globally, and reduce (or eliminate) the number of morph vectors required.

1. INTRODUCTION

Device connections are commonly tested using an architecture known as boundary scan [1]. Boundary scan interconnect test vectors tend to change all outputs at once. In large boards or systems where there are many such outputs changing at the same time, there is a potential problem with ground bounce [2,6]. Ground bounce is a name for the phenomenon of the ground plane briefly having a voltage other than true ground. This may occur to the power plane also, and so it is sometimes referred to as *simultaneous switching noise* (SSN).

If this switching noise is sufficiently large, the chip's logic, input buffers, and drivers may not behave as expected [2,6]. Even though devices are designed to handle simultaneous switching, they handle it based on functional switching. For instance, in a functional mode the device may only drive one of its two buses, and the device's power design will allow all of those bus signals to change at once. However, when boundary scan operates both of those buses simultaneously, the power structure of the device may be insufficient to account for the switching noise that occurs.

Such interference can cause faulty operation of boundary scan vectors. The vectors therefore may occasionally need to limit the number of outputs that are allowed to switch at a time. This is known as the *simultaneous switching output limit*, or SSOL. The most common solution is to try to use the same patterns that would be used if there were no SSO limits, but also to use intermediate vectors, called *morph vectors*, that prevent too many outputs from switching at a time.

Work by Hollmann et al. has produced small test pattern sizes, but not strictly true/complement patterns (due to morphing) and not always optimal when taken as a whole true/complement pattern set [4]. The Hollmann paper claimed optimality, but the optimality only applied to half of the codeword. Once complemented and made into a true/complement pattern set it became suboptimal in certain conditions due to excessive morphing between the two codeword halves. One suggestion was put forth that their solution was indeed optimal when considered as a counting pattern, rather than a true/complement pattern, because that codeword half would be all that is required to uniquely identify each net (though there could also be aliasing issues). However, without complementing it to create the true/complement patterns, all zero and all one codewords would be created. This is unacceptable in boundary scan interconnect test vectors, since certain stuck-at faults would then be undetectable on those nets.

In addition, their paper provides a high-level algorithm for calculating the characteristics of the optimal test pattern set, but does not provide an algorithm to generate the patterns once those characteristics are known (as discussed in section 9). While this is conceptually simple, an algorithm still needs to be provided since different approaches would have different efficiencies.

In spite of these drawbacks, the approach used by Hollmann et al. still has many merits. The algorithm presented in our paper is in some ways a variation of and improvement to the method described in the Hollmann paper, in that it uses the same idea of having Pascal's triangle (an implementation of the binomial coefficients used in the Hollmann paper) represent the number of codewords having a certain number of transitions. However, this paper also introduces a method of using that information to quickly generate the codewords, as well as demonstrating how the new method can be applied to create

0-7803-8580-2/04 $20.00 Copyright 2004 IEEE

safe initial vectors, reduce morphing, and allow for multiple device switching limits.

2. TRUE/COMPLEMENT PATTERNS

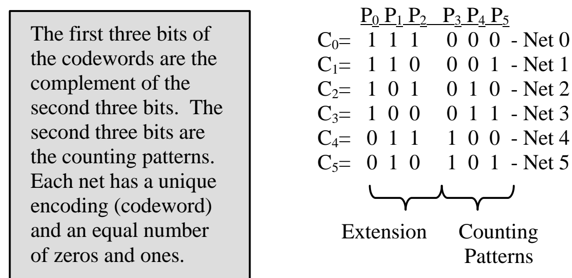
In generating test patterns for boundary scan interconnect tests, each net needs to have a unique encoding (also known as a *codeword*), and each encoding needs to have both ones and zeros in it. Meeting these requirements will allow any shorts, opens, or stuck-at faults to be detected. This assumes all nets have boundary scan control and observability.

Wagner presented an algorithm now commonly used to generate test patterns [5]. These patterns are known as *true/complement patterns*. They not only meet the requirements for detecting shorts, opens, and stuck-at faults, but also provide an improved ability to isolate the faults.

True/complement patterns are generated by using a counting pattern as the basis of the codeword for each net. Then these codewords are extended by prepending the counting pattern with its complement. This gives us two halves to the codewords, a true half and a complement half. The counting patterns provide unique encodings. The complement ensures that all codewords now have the same number of zeros and ones (with at least one of each).

Codewords translate into test vectors by taking the same bit out of each codeword and applying each of those bits in one test vector. Thus, the test vectors (P_x) are the columns in Figure 1, and each row is the corresponding net's codeword (C_y). For the example in Figure 1 there would be six test patterns required to apply these codewords, one for each bit in the codeword.

Figure 1 – True/Complement Patterns



The first three bits of the codewords are the complement of the second three bits. The second three bits are the counting patterns. Each net has a unique encoding (codeword) and an equal number of zeros and ones.

	P_0	P_1	P_2	P_3	P_4	P_5	
C_0=	1	1	1	0	0	0	- Net 0
C_1=	1	1	0	0	0	1	- Net 1
C_2=	1	0	1	0	1	0	- Net 2
C_3=	1	0	0	0	1	1	- Net 3
C_4=	0	1	1	1	0	0	- Net 4
C_5=	0	1	0	1	0	1	- Net 5

Extension (P0–P2) | Counting Patterns (P3–P5)

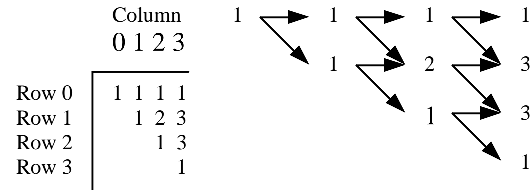
3. PASCAL'S TRIANGLE

Generating the test patterns can also be done by using Pascal's triangle. Before looking at how, a quick review of the characteristics of Pascal's triangle is in order. The triangle can be built based on the equation:

$$P(r,c)=\begin{cases} P(r-1,c-1)+P(r,c-1), & r>0, c>0, r\le c \\ 1, & r=0, r\le c \\ 0, & \text{otherwise} \end{cases}$$

This gives us the triangle in Figure 2 (with zeros not shown), where P(0,0) is the top left corner, r is the row (starting at row 0), and c is the column (starting at column 0).

Figure 2 – Pascal's Triangle and Tree Graph



There are many interesting qualities about this structure. For our purposes, we would like to interpret this triangle as a graph where each number is a node and each node has two outgoing (as well as two incoming) edges. The node's position will be referred to as (r,c), and its value will be P(r,c). One edge goes straight to the right from (r,c) to (r, c+1), while the other goes right and down to (r+1, c+1). Note that this corresponds to how the P(r,c) values are iteratively created.

Using this graph, P(r,c) can be interpreted as the number of unique paths that exist from (0,0) to (r,c). For instance, P(2,3)=3 indicates that there are 3 unique paths from (0,0) to (2,3).

This has a powerful application, in that we can use this graph to represent our encodings. In generating the test patterns, we want to uniquely identify some number of nets. We can take advantage of the Pascal graph's ability to count unique paths by having (0,0) represent a shared starting point for the nets. All of the nets whose codewords start with the same value will be represented at (0,0), so we will be determining two sets of unique paths (one for codewords that start with '0' and one for codewords that start with '1'). Each unique path will represent a different codeword.

Column c represents the number of edges traversed to get from (0,0) to (r,c). Therefore each column symbolizes a test pattern (with c+1 total test patterns) in the codeword half. Each row represents the number of transitions in the codeword. In this way, P(r,c) represents the number of unique paths, which in turn represents the number of unique codewords, that start with the same value and can have r transitions in c+1 patterns. This means that there are $2*\sum_r P(r,c) = 2*2^c = 2^{c+1}$ unique (c+1)-bit code-

words. Each down edge in the unique path indicates a transition (0 to 1 or 1 to 0) in the codeword from one bit to the next, while each straight edge does not.

4. NUMBER OF TEST PATTERNS

The next step of the algorithm is similar to the method presented in the Hollmann paper. The goal of this step is to determine the fewest number of test patterns (smallest c value) that 1) have enough unique encodings available for all the nets, and 2) use no more transitions between any two codewords than allowed by the switching limit, s. The pseudocode for this step is provided in Figure 4.

The first step is to make sure that we have enough encodings for the number of nets we have. This is similar to the traditional test generation method for k nets, where the smallest number of test patterns (for one half of the codeword) is $c+1 = \lceil \log_2(k) \rceil$. This corresponds to the smallest c that satisfies $2*\sum_r P(r,c) \geq k$, which is essentially what the Hollmann method solves for at this point.

In our algorithm, essentially the same thing is done, except that it divides the problem into two parts: one for all codewords beginning with '0' and one for those beginning with '1'. This allows the possibility of having a starting vector that has an unequal number of zeros (k_0) and ones (k_1), which we will take advantage of later. Previous algorithms have assumed that those two parts had the same number of nets ($k_0 = k_1 = k/2$). Our algorithm does not have that limitation, though better results can be obtained when k_0 and k_1 are the same.

To determine the number of columns c, we determine the number of columns that the larger set of nets would require in order to provide each net a unique encoding. This gives us $c = \lceil \log_2(\max\{k_0, k_1\}) \rceil$. Notice that if $k_0 \neq k_1$, then c may be different than if it was calculated based only on k.

For c columns, there are 2^c possible unique encodings ($2^c = \sum_r P(r,c)$). When $k_0<2^c$ (or $k_1<2^c$), we must choose which of the 2^c encodings to use. In this case, we will not need to use all of the rows of the Pascal's triangle. Earlier it was mentioned that each row number in Pascal's triangle represents the number of transitions in the codeword. For instance, nets in row 0 have no transitions. The Hollmann method capitalizes on this by using the smallest rows first when not all of the possible encodings are required.

Our algorithm does the same in that the rows are used starting from r=0 until a row r_0' is reached that satisfies $\sum_{r=0}^{r_0'} P(r,c) \geq k_0$ (and likewise for r_1' and k_1).

Thus, r_0' is the last row we will need to use. This gives us two Pascal maps, M_0 and M_1, which denote how many nets will be used from each row (how many nets will have a given number of transitions). $M_x(r)$ will denote the value of a given row in map x. By construction, $M_x(r) \leq P(r,c)$. Figure 3 shows an example of what the map would be for all nets whose codeword halves start with '0' when c=4. The values in M_0 are simply the Pascal's triangle values from column 4, up to when its sum matches the number of nets in the map.

Figure 3 – Sample map M_0

k=20, k_0=k_1=10, c=4 (5 test patterns), s=8

r	P(r,4)	M_0(r)	T_0(r)	T_0
Row 0	1	1	0 *1 = 0	0
Row 1	4	4	1*4 = 4	+ 4
Row 2	6	5	2*5 = 10	+ 10
Row 3	4	0		14
Row 4	1	0		

Once we have determined r_x' for each map, we then need to determine if too many transitions are required to provide the unique encodings in c+1 test patterns. The number of transitions required for a given row is the number of transitions per net (the row number) times the number of nets, or $T_x(r) = r*M_x(r)$. Therefore, the number of transitions required for a given map is $T_x = \sum_r T_x(r) = \sum_r r*M_x(r)$. These values are then summed over each map (M_0 and M_1, in this case). This provides us with the total number of transitions required, $T = \sum_x T_x$. Figure 3 shows an example of how T_0 is calculated for the given map.

If the total number of transitions required is more than the total number of transitions available ($T > c*s$), then it is not possible for the test to use c+1 or fewer test patterns. If more test patterns are required, c is incremented and the process of generating Pascal maps is repeated until the number of transitions required is acceptable ($T \leq c*s$).

Figure 4 – Create Maps Pseudocode

```
c =max( log2(k0), log2(k1))-1; //-1 for do loop
create_maps(){
  do {
    Increment c;
    Generate M0, M1 for c columns;
    Determine T = T0 + T1;
  } while (T > s*c) //too many transitions
}
```

4.1 Safe Initial Vectors

As mentioned, this algorithm splits the k nets into two groups, one of size k_0 and one of size k_1, where $k_0+k_1=k$. This allows us to easily use any number of zeros and ones in the first vector. Since the first vector can now match the existing state of the board, the test patterns can be generated in such a way that the first vector does not violate the switching limit. This assumes that the existing state of the board can be known (from knowing its reset state or capturing and shifting out its current state before generating the vectors, for instance), so the design of the board (and test plan) needs to accommodate that. It should be noted that when the two maps are the same size they will yield smaller overall test pattern sizes since they are better balanced.

5. MORPH REDUCTION

When Pascal's triangle is used to generate the codeword half in the way described above, the codeword half itself is guaranteed to be as small as it can be while still uniquely identifying each net and not violating the switching limit [4]. However, it then gets complemented and put together with that complement to create the entire codeword. In that case, it is entirely possible for there to be a switching limit violation between the true and complement halves of the codeword.

The solution to this in the Hollmann method was to simply insert morph vectors. However, this may not be necessary. A transition will occur between the two halves when the first and last bits of the codeword half are the same value [6]. This will only occur when an even number of transitions exist in the codeword. In other words, this will only happen to nets in even rows of the Pascal map.

Therefore, by adding the number of nets in each even row of every Pascal map, $t_H = \sum_x \sum_{r,even} M_x(r)$, we can determine how many transitions, t_H, would occur between codeword halves. In addition, we may be able to move some of these nets into odd rows to reduce the morphing that is otherwise required.

The total number of transitions allowed over the c+1 test patterns, t_A, is c*s, where s is the switching limit. However, only $T = \sum_x T_x$ are actually used. If there are fewer transitions used than available (T < c*s), we have room to add transitions to move some nets into (higher) odd rows.

The goal is to move as many nets as possible from even rows to odd rows without exceeding the number of available transitions. This will reduce morphing by as much as possible without requiring additional test vectors. Moving a net from an even row to the following odd row will require one additional transition, since that net will have r+1 transitions instead of r now.

The algorithm will attempt to move as many nets from even rows as possible. It begins at the largest even row. From there, it moves nets to the next odd row that has room for more nets (for the next odd row r for which $P(r,c) > M_x(r)$). Once that even row is emptied, the next largest even row will have its nets moved to the next available odd row. This is repeated until all available c*s transitions are used up (or there are no more than s nets in even rows).

Figure 5 – Sample Morph Reduction

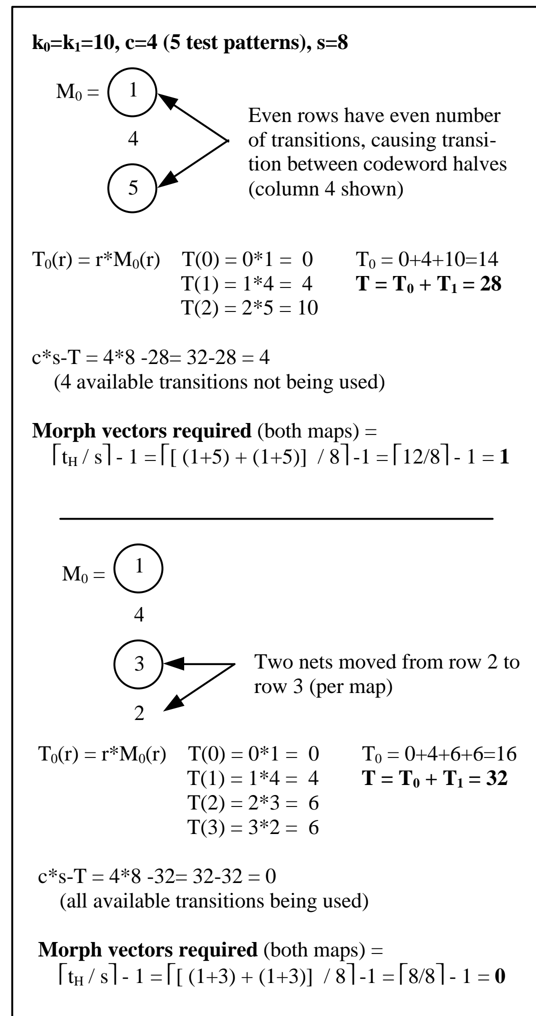


Figure 5 demonstrates how we can take advantage of the available transitions to reduce morphing. This example is for k=20 and s=8. It is being solved in c+1=4+1=5 test patterns, so there are c*s=4*8=32 available transitions, but only 28 are used. However, there are more than s=8 transitions between code-

word halves. Yet by moving some nets from row 2 (two transitions in the codeword) to row 3, the morphing is reduced. This demonstrates that fewer overall transitions within the codeword half may not result in fewer true/complement test patterns.

To get strictly true/complement test patterns with no morphing (*morph elimination*), this same process is used with one difference: if the total available transitions are exceeded before the even rows have s or fewer nets, two new test patterns will be required. This is done by incrementing the number of columns in the codeword half (from c to c+1) and repeating the process from where the Pascal map was generated in section 4. The pseudocode in Figure 6 shows how both morph reduction and morph elimination would be handled.

Figure 6 – Morph Reduction Pseudocode

```
c =max( log2(k0), log2(k1));
do {
      create_maps ();
 } while ( morph_analysis())
                        ---------------------
morph_analysis(){
   while (T<c*s)
      Determine r_odd, r_even for each map;
      for each map
         cost = M_x(r_odd) – M_x(r_even);
         avail_in_odd = P(r_odd,c)-M_x(r_odd);
         Move min(éM_x(r_even)/costù, c*s-T, avail_in_odd)
                from M_x(r_even) to M_x(r_odd) ;
         Set new r_odd or r_even if needed;
         Recalculate T;
   if (t_H>s) AND (eliminate_all_morphs=true)
      return true;   //expand and reanalyze
   else
      return false;  //don't expand and reanalyze
}
```

6. DEVICE SWITCHING LIMITS

Our algorithm, as presented thus far, assumes that the switching limit is a global constraint that applies to the entire board. In practice, however, it may be desirable for the switching limit to apply to one device (or have multiple constraints for multiple devices). This localized switching limit could also be applied to any group of nets that required its own switching limit, such as a cluster of devices or a section of a device. Figure 7 demonstrates how this algorithm would be applied in a sample problem. Figure 8 presents the pseudocode used by this algorithm to accommodate the localized switching limit.

Each device or other group of nets that has its own switching limit, s_y, will have two maps, M_{y0} and M_{y1}, to represent it (one for all codewords starting with '0' and one for all starting with '1'). In addition, there will be two "master maps", M_{M0} and M_{M1}, where $k_0 = \sum_y k_{y0}$ and $k_1 = \sum_y k_{y1}$.

The two master maps will be solved in the same manner as when we considered a global switching limit. Hence M_{M0} and M_{M1} will be made as before, as well as T_{Mx} for each of the master maps. The number of transitions available, $V=(c^* \sum_y s_y) - \sum_x T_{Mx}$, is calculated based on these master maps using the sum of all of the switching limits.

Next, each of the map pairs, $M_{x,y}$, is created. This is done by setting $M_{x,y}(r) = M_{Mx}(r)$ for each row, starting at row 0, until there is no k_{yx} left. For the last row, r', of $M_{x,y}$, $M_{x,y}(r') \leq M_{Mx}(r')$, since there may be insufficient $k_{x,y}$ to use all of the nets available in that row of the master map. This is comparable to building $M_x(r)$ from P(r,c) earlier. Now $M_{x,y}(r)$ is built from $M_{Mx}(r)$.

Once created, each of the map pairs is then analyzed separately to determine how many transitions, T^y, are required for that particular map pair and how many transitions are available to that map, $V_y = c^*s_y - T^y$. Each pair of maps is then sorted (as a pair) based on how many available transitions the pair has.

The pair with the fewest available transitions is used first. Its maps move as many nets up to higher rows as it can. This is similar to the morph reduction, except that we are no longer concerned with odd and even rows. At this point we simply want to move up as many as we can. Starting at the highest row, as many nets are moved to the next row as possible. Then the next highest row repeats the process and so on until all available transitions are used up. If the first map of the pair has moved up all the nets it can but still has transitions available, the next map in the pair repeats the process.

The master maps are then reduced by whatever was used by the map pair, $M_{Mx}(r) = M_{Mx}(r)-M_{x,y}(r)$. Once both master maps are updated, the remaining map pairs M_{yx} are recreated based on the new master maps. The number of transitions available is recalculated, the map pairs are resorted, and the process repeats. If at any point there are no available transitions and yet there are map pairs that cannot be created based on the master map ($T_{yx} > c^*s_y$), the number of columns is expanded by one and the entire procedure starts over (with the recreation of the master maps).

This process can be executed before the morph analysis to allow the morph analysis to be performed also (on each map pair). Morph reduction will be limited, however, since many of the available transitions may have already been used up.

Figure 7 – Sample Device Limit Problem

Net Group (y)	k_{y0}	k_{y1}	s_y
y=1	5	6	6
y=2	8	3	6

Calculate k_0, k_1, c: $k_0 = 5+8 = 13$ $k_1 = 6+3 = 9$ $c=\lceil \log_2(\max\{13,9\}) \rceil=4$

Create M_{M0}, M_{M1}:

r	P(r,4)	$M_{M0}(r)$	$T_0(r)$	$M_{M1}(r)$	$T_1(r)$
0	1	1	0 *1 = 0	1	0 *1 = 0
1	4	4	1 *4 = 4	4	1 *4 = 4
2	6	6	2 *6 = 12	4	2 *4 = 8
3	4	2	3 *2 = 6		
4	1	0			

Calculate T_0, T_1: $T_0 = 0+4+12+6 = 22$ $T_1 = 0+4+8 = 12$

Calculate V: $V = (4*(6+6)) - (22+12) = 48-34 = 14$

Create $M_{x,y}$:

r	$M_{M0}(r)$	$M_{0,1}(r)$	$T_{0,1}(r)$	$M_{0,2}(r)$	$T_{0,2}(r)$
0	1	1	0 *1 = 0	1	0 *1 = 0
1	4	4	1 *4 = 4	4	1 *4 = 4
2	6	0		3	2 *3 = 6
3	2	0			

r	$M_{M1}(r)$	$M_{1,1}(r)$	$T_{1,1}(r)$	$M_{1,2}(r)$	$T_{1,2}(r)$
0	1	1	0 *1 = 0	1	0 *1 = 0
1	4	4	1 *4 = 4	2	1 *2 = 2
2	4	1	2 *1 = 2		

Calculate T^1, T^2: $T^1 = \sum_x \sum_r Tx,0(r) = (0+4) + (0+4+2) = 10$

$T^2 = \sum_x \sum_r Tx,1(r) = (0+4+6) + (0+2) = 12$

Calculate V_0, V_1: $V_1 = 4*6 - 10 = 14$

$V_2 = 4*6 - 12 = 12$ -- fewest available transitions

Adjust map y=2:

r	$M_{M0}(r)$	$M_{0,2}(r)$ →	$M_{0,2}(r)$ →	$M_{0,2}(r)$	$T_{0,2}(r)$
0	1	1	0	0	0 *0 = 0
1	4	4	1	0	1 *0 = 0
2	6	3	5	6	2 *6 = 12
3	2	0	2	2	3 *2 = 6

r	$M_{M1}(r)$	$M_{1,2}(r)$ →	$M_{1,2}(r)$ →	$M_{1,2}(r)$	$T_{1,2}(r)$
0	1	1	0	0	0 *0 = 0
1	4	2	1	0	1 *0 = 0
2	4	0	2	3	2 *3 = 6

$T^2 = (0+0+12+6) + (0+0+6) = 24 \leq 4*6=24$

Adjust M_{M0}, M_{M1}:

r :	$M_{M0}(r)$ -	$M_{0,2}(r)$ =	$M_{M0}(r)$	$T_{0,2}(r)$
0	1	0	1	0 *1 = 0
1	4	0	4	1 *4 = 4
2	6	6	0	
3	2	2	0	

r :	$M_{M1}(r)$ -	$M_{1,2}(r)$ =	$M_{M1}(r)$	$T_{0,2}(r)$
0	1	0	1	0 *1 = 0
1	4	0	4	1 *4 = 4
2	4	3	1	2 *1 = 2

Remaining Map Pairs

Net Group (y)	k_{y0}	k_{y1}	s_y
y=1	5	6	8

Calculate T_0, T_1: $T_0 = 0+4= 4$ $T_1 = 0+4+2 = 6$

Calculate V: $V = (4*(6)) - (4+6) = 24-10 = 14$

Create $M_{x,y}$: $M_{x,1}(r) = M_x(r)$

Calculate T^1: $T^1 = (0+4) + (0+4+2) = 10$

Calculate V_0, V_1: $V_1 = 4*6 - 10 = 14$

Adjust map y=1: Since $M_{x,1}$ is last map pair, $M_{x,1} = M_{Mx}$

Figure 8 – Device Switching Limit Pseudocode

input k_{y0}, k_{y1}, s_y, for each map pair;
calculate master k_0, k_1, s;
c =max($\log_2(k_0)$, $\log_2(k_1)$);
do {
  create_maps(); // make M_{M0} and M_{M1}
} while (multi_maps() OR morph_analysis())
----
multi_maps(){
  While size(M_{M0})>0 or size(M_{M1})>0
    Calculate T_{Mx} for each master map
    Calculate V
    if (V < 0)
      Increment c;
      return true; //insufficient V
    While there are remaining map pairs
      Calculate M_{yx}, T_y for each remaining map pair
      Calculate V_y for each remaining map pair
      Select map pairM_{y0}, M_{y1} with lowest V_y
      if (V_y < 0)
        Increment c;
        return true; //insufficient V_y
      for each master map (0 or 1)
        move as many nets in M_{yx} up to larger rows as possible without exceeding V_y
        for each row in M_{zx}
          $M_{Mx}(r)$ -= $M_{zx}(r)$;
  return false;
}

7. TRANSITION MAPPING

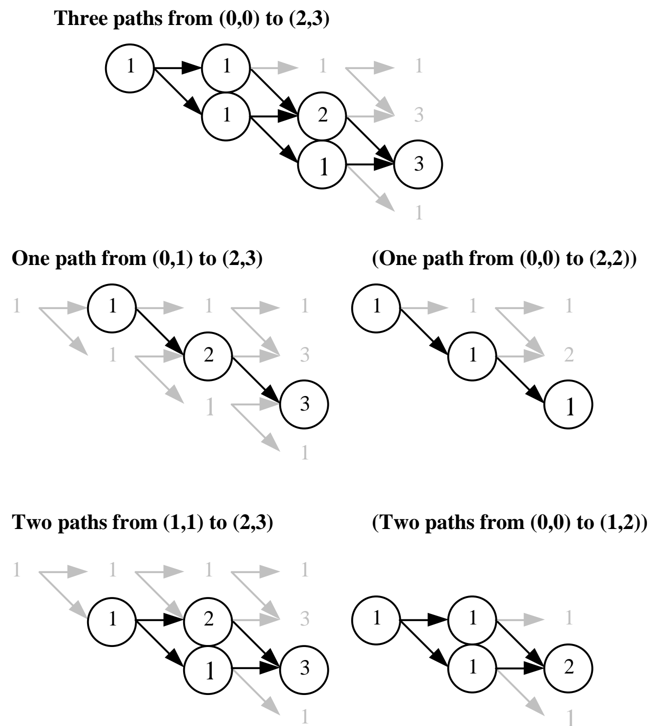
Pascal's triangle has other interesting features, too. In addition to finding how many paths there are from (0,0) to (r,c), it is also possible to find those exact paths. To this point, we have only been considering the P(r,c) nets at (0,0) that end up at (r,c).

When traversing the map, the number of possible paths amounts to counting how many edge traversal paths can get us from (0,0) to (r,c). This same idea can be used when we are at point (i,j) and wanting to get to (r,c). We simply need to know how many moves right and how many moves down along the edges are required to get us from our current location to our destination. Thus, assuming that $0 \leq i \leq r$, $0 \leq j \leq c$, $r \leq c$, and $i \leq j$, to get from (i,j) to (r,c) there are P(r-i, c-j) paths.

There are P(r,c) nets at (0,0) that end up at (r,c). The number of nets at (1,1) that end up at (r,c), assuming $r \geq 1$, $c \geq 1$, is P(r-1,c-1). Therefore, there are P(r-1,c-1) nets at (0,0) that follow the transition edge to (1,1). The number of nets at (0,1) that end up at (r,c) is P(r,c-1), so there are P(r,c-1) nets that follow the non-transition edge to (0,1).

Figure 9 demonstrates this concept. In this example, we are trying to get from (0,0) to (2,3). Our current decision is how many of our P(2,3)=3 nets that start at (0,0) and end up at (2,3) will pass through (0,1) and how many will pass through (1,1). To determine this, we look at how many possible paths there are from (0,1) to (2,3). This is equivalent to determining how many possible paths there are between (0,0) and (2,2), as shown on the right side of the figure. Likewise, determining the number of paths from (1,1) to (2,3) is equivalent to the number of paths from (0,0) to (1,2).

Figure 9 – Path Example



This same logic can be used to determine how many nets at arbitrary node (i,j) follow the transition edge, assuming that $0 \leq i \leq r$, $0 \leq j \leq c$, $r \leq c$, and $i \leq j$. P(r-i-1, c-j-1) nets will follow the transition edge, while P(r-i,c-j-1) will follow the non-transition edge. Notice that there are only P(r-i, c-j) nets at (i,j). If we define P(-1,j)=P(i,-1) = 0, we can use this information to form an iterative algorithm to determine exactly what paths are used to reach (r,c) from (0,0).

The codeword values are assigned by following the edges. For the nets following the transition edge, their next bit value will be the opposite of their current bit value. The other nets will have the same value in their next bit as their current bit. In our example, for instance, there would be three nets, each starting with a value of '0'. One net's codeword would become '00', while the two following the transition edge would become '01'. Eventually, the three codewords would be '0010', '0110', and '0100'.

To this point, we have been considering that P(r,c) nets are trying to get from (0,0) to (r,c). However, if the map's row is not full ($M_x(r) < P(r,c)$), the number of nets to split between the two outgoing edges, $M_x(r)$, would be less than P(r,c). In this case, the number of nets following the transition edge will be limited by three factors.

7.1 Number of Nets Limit

First, the number of nets at node (i,j) will be less than P(r-i, c-j), so the number following the transition edge will need to be capped at the number of nets that are at (i,j) to start with. This is because we can't have more nets follow the outgoing edge than there are nets in the node.

7.2 Transitions Allowed Limit

Second, each column will not necessarily have the exact same number of transitions. When all the rows were full, that was equivalent to all available transitions being used (in every column). However, if the rows are not full, then there are columns that will use fewer than P(r,c) transitions.

This means that before we can actually determine the path each node will take, we need to determine how many transitions each map will use for each column (from 0 to c) of each row. The number of transitions used by a map's row r is $T_x(r) = r*M_x(r)$. $T_x(r)/c$ is the average number of transitions that are used between each test pattern (for a given map and row). The remainder, $T_x(r) \bmod c$, tells us how many of those columns will be rounded up from the average for map x and row r. We want to find a way to spread out the columns that get rounded up (over all the rows and maps) so that we can use all the transitions available to us without violating the switching limit.

Now that we know how many columns will get rounded up in a given row and map, $T_x(r) \bmod c$, we want to use that information to determine how many columns will get rounded up over all rows and maps, U_T. Since U_T is the sum of all rounded up columns for each row of each map, $U_T = \sum_x \sum_r (T_x(r) \bmod c)$.

To determine how many columns are rounded up from average irrespective of the row and map, we then perform the same sort of calculation we did for $T_x(r)$. Specifically, we create a new array called U(cols), indicating how many roundups (over all rows and maps) there are for the given column col. We then say that for the first (U_T mod cols) columns

$U(cols) = \lceil U_T/cols \rceil$, and for the other columns $U(cols) = \lfloor U_T/cols \rfloor$.

An example of how this is beneficial is when there are two maps that both round up half of the columns (c/2). When combined, on average every column would be rounded up (2*c/2). We want to make sure that rather than both maps rounding up the first c/2 columns, one map rounds up the first c/2 columns while the other rounds up the last c/2 columns. This will cause the transitions for each column to be closer to the average, preventing any of them from exceeding the switching limit. In this case, $U_T = c$, so U(cols)=c/c=1 for each column. This indicates that each column will only allow one map's row to round up, forcing a balance.

We assign the number of actual transitions per row and column for a given map, $RQ_x(r,c)$, based on this information. $RQ_x(r,c)$ will have ($T_x(r)$ mod c) columns that are rounded up to $\lceil T_x(r)/c \rceil$, with the rest rounded down to $\lfloor T_x(r)/c \rfloor$.

There is always one current round-up column, which begins at column 0. This column is the one that $RQ_x(r,c)$ begins being filled from. So the first ($T_x(r)$ mod c) columns, beginning at the current column (which will wrap back around to column 0 if the last column is reached), are rounded up. The current column then points to the column after the last rounded up column. In this way, all roundups in RQ are spread evenly regardless of which map or row is being used. The pseudocode for this step is shown in Figure 10.

Now that we have determined how many transitions each column of each map and row are allowed, we have addressed our second limiting factor in determining how many nets in each row to allow to transition from one test pattern to the next.

Figure 10 – Transitions Per Column Pseudocode

make_RQ(){
  $U_T = \sum_x \sum_r (Tx(r) \bmod c)$;
  Make the first (U_T % cols) columns in $U_{max}(col)$ have $\lceil U_T/cols \rceil$ transitions, then rest of columns have $\lfloor U_T/cols \rfloor$ transitions;
  for each map
    for each row
      Starting at current column, set first ($T_x(r)$ mod c) columns of $RQ_x(r,c)$ to $\lceil T_x(r)/c \rceil$ then the rest have $\lfloor T_x(r)/c \rfloor$;
      Move current column up by ($T_x(r)$ mod c);
}

7.3 Proportion Limit

The third limiting factor is that we want the proportion of the number of transitions to the number of non-transitions to remain the same. If there are only 8 nets at node (i,j), but P(r-i,c-j)=10 (meaning there could be 10 nets in that node), then that same proportion should be used as the desired number of transitions from the node. In that case, instead of having P(r-i-1, c-j-1) nets follow the transition edge, there would be .8* P(r-i-1, c-j-1) nets that follow it (assuming they were not capped by the other two limiting factors).

7.4 Codeword Generation

We now cap the number of transitions we would normally allow, P(r-i-1, c-j-1), by 1) the number of nets in (i,j), 2) $RQ_x(r,c)$, and 3) a factor of the number of nets at (i,j) divided by P(r-i,c-j). We will use the same procedure as before, even though not all P(r-i, c-j) nets are being used. A graph algorithm is used to move from the root node (0,0) to (r,c), based on the number of nets taking transition edges from each node.

As the number of nets to transition is determined, the nodes are sorted based first on i and then on the number of nets represented by the node. This allows for the nets requiring more transitions to be able to get them before other nets. The pseudocode for the codeword generation step of the algorithm is given in Figure 11.

Figure 11 – Codeword Generation Pseudocode

input k_{y0}, k_{y1}, s_y, for each map pair;
calculate master k_0, k_1, s;
c =max($log_2(k_0)$, $log_2(k_1)$);
do {
    create_maps(); // make M_{M0} and M_{M1}
} while (multi_maps() OR morph_analysis())
make_RQ();
make_cword();

Make_cword(){
  for each map
    for each row
      Make graph root node with $M_x(r)$ nets
      For each col
        For each leaf node
          Make children nodes based on how many follow the transition edge;
          Update the codewords based on edge taken;
          Sort nodes;
}

8. ENHANCEMENTS

This algorithm offers many features not available in other algorithms, efficiently producing good results. Some of the advantages are due to the software implementation, while most are from the algo-

rithm itself. Even with these extra features and efficient approaches, additional research could yield an even better solution.

The morph reduction analysis currently only considers two possibilities – not expanding by any columns, and expanding by however many is necessary to eliminate morphing entirely. It may be possible that the optimal number of true/complement vectors lies somewhere in between (expanding, but not to the point of eliminating the morph vectors entirely).

This algorithm could easily be modified to set a given number of vectors to expand by. In fact, it would even be reasonable to have the algorithm iteratively check increasing column sizes until it found the minimal number of test patterns (assuming quadratic behavior) or to try every possibility and select the lowest number of overall test patterns required.

The device switching limit algorithm is not proven to produce optimal test patterns. There are some variations that could possibly improve the algorithm. For instance, the algorithm assumes that when assigning map pairs, it should always start with M_{y0}, then go to M_{y1}. It may be better to try to split the extra transitions evenly between the two maps rather than giving first to one and then giving whatever is left to the other. In addition, it may be better to try to split the transitions between the map pairs at the same time, rather than giving one map pair all it can handle and then moving on to the next pair.

9. RESULTS

The following results were based on results of running a software implementation of the algorithm (except where otherwise noted). The machine used to obtain the results was a 1.7GHz Pentium with 512 MB of RAM running Windows 2000.

When using a global SSOL constraint, no morph reduction, and k_0=k_1, the number of test patterns generated by this algorithm is the same as that generated using the Hollmann method. This is not surprising, since they both use the fewest overall transitions possible and therefore have the same number of required morph vectors. However, with morph reduction turned on, this algorithm offers a slight improvement over the Hollmann method. Morph elimination generally results in larger pattern sets. When it does not, the morph elimination vectors may be preferable since they are the same size but have no morphing. These results are shown in Table 1.

Not only were the test pattern sizes very small, the runtimes were also very good. The algorithm lent itself to some software optimizations that improved runtime, as well. Even with switching limits as low as 2% (of up to 15000 nets), the program ran nearly instantaneously (less than a second). When using global SSO limits, the time complexity of the software is O(r*c*k). The "c*k" portion corresponds to the total number of test bits. With the small time complexity constants, this is very reasonable.

It is difficult to compare the runtime efficiency of this algorithm to the Hollmann method. With regard to actually generating the codewords, the Hollmann paper simply describes the number of transitions that the codewords will have and the number of test patterns over which those transitions will occur ("take as code words all words of length *b* with at most *t*-1 transitions…"). It does not provide an explicit means of generating those codewords, only a description of them. While this is not conceptually difficult, different generation algorithms could have different runtime characteristics and varying efficiency. Likewise, it is difficult to determine the efficiency of other portions of their algorithm since they are only stated as mathematical equations (that appear to be costly in terms of software runtime, but again that depends on the implementation).

Table 1 – Data for k_0=k_1=k/2 (Global SSOL Constraint)

morph elimination / morph reduction [no morph analysis=Hollmann Method]

SSOL (% of nets)	8000 nets	7000 nets	6000 nets	5000 nets
50 %	28 / 28 [29]	28 / 28 [29]	26 / 26 [26]	26 / 26 [27]
45 %	28 / 28 [29]	28 / 28 [29]	26 / 26 [27]	26 / 26 [27]
40 %	30 / 29 [29]	28 / 28 [29]	28 / 28 [29]	28 / 28 [28]
35 %	30 / 30 [31]	30 / 30 [30]	30 / 30 [31]	28 / 28 [29]
30 %	32 / 32 [33]	32 / 32 [33]	32 / 32 [33]	32 / 31 [31]
25 %	38 / 37 [38]	38 / 37 [38]	36 / 36 [36]	36 / 35 [37]
20 %	44 / 43 [43]	44 / 42 [43]	42 / 42 [43]	42 / 40 [41]
15 %	52 / 51 [51]	50 / 50 [51]	50 / 49 [50]	48 / 47 [48]
10 %	60 / 60 [61]	60 / 60 [61]	60 / 60 [61]	60 / 60 [61]
5 %	120 / 114 [114]	120 / 113 [114]	120 / 112 [112]	120 / 111 [111]

Table 2 shows the results for varying k_0, k_1, and s when k=5000. It is evident that k_0 and k_1 had little effect in the overall pattern sizes, even when they were quite different from each other. This implies that when we create patterns with safe initial vectors, there will be little negative consequence in terms of number of test patterns required. It is advantageous that this algorithm produces a pattern set with an arbitrary initial pattern and does so without incurring a significant penalty in the number of test patterns.

Table 2 – Data for $k_0{}^{1}k_1$, with morph reduction

SSOL (% of nets)	k_0=2500 k_1=2500	k_0=1750 k_1=3250	k_0=1000 k_1=4000	k_0=250 k_1=4750
50 %	26	26	26	28
45 %	26	26	28	28
40 %	28	28	28	29
35 %	28	29	30	30
30 %	31	31	32	32
25 %	35	35	35	37
20 %	40	40	40	43
15 %	47	47	50	52
10 %	60	60	60	62
5 %	111	111	112	114

Table 3 – Data for $k_0{}^{1}k_1$, device switching limits

# of Devices (y)	k_0,k_1,s	# of Patterns (local)	# of Nets	# of Patterns (global)
2	12,4,8	10	32	12
	8,8,8	8		
3	11,11,8	11	60	14
	10,10,8	10		
	9,9,8	10		
4	10,5,5	10	60	15 (one morph vector)
	10,5,5	10		
	15,0,5	12		
	5,10,6	10		

Table 3 presents data obtained from applying the device switching limit algorithm to some small sample problems (not generated by software). If the algorithm did not consider the combination of the devices in determining the test patterns, it would have to then combine the independent results somehow. This is inefficient, because there will be duplicate codewords between devices, requiring the patterns for each device to not be applied at the same time.

By analyzing the devices with one master map pair, the overall test patterns are optimized over all devices. The codewords are made unique over all map pairs using these master maps. The codewords are made to not violate the switching limits by then considering each device's map pair separately.

10. CONCLUSION

This algorithm produces an efficient means of generating safe and effective test vectors. The primary advantage of this algorithm is that it makes provisions for several previously unaddressed concerns - namely safe initial vectors, reduced morphing, and constraints by device. Another advantage is that it does all this with good runtime efficiencies.

By using two separate maps (one for codewords beginning with '0' and one for those beginning with '1'), the algorithm is flexible enough to provide arbitrary first vectors that do not violate the switching limit. It optimizes over the whole true/complement codeword, not just the true half of the codeword, allowing for reduced morphing. The use of map pairs for each constrained set of nets allows for multiple constraints to be made for specific groups of nets (rather than one global constraint).

11. Acknowledgements

The work of Mehrdad Nourani was supported in part by the National Science Foundation CAREER Award # CCR-0130513.

12. REFERENCES

[1] IEEE Std 1149.1-1993, "IEEE Standard Test Access Port and Boundary-Scan Architecture".

[2] A. Kabbani, A. Al-Khalili, "Estimation of Ground Bounce Effects on CMOS Circuits", *IEEE Transactions on Components and Packaging Technologies*, Jun 1999, vol.22, no. 2, pp. 316-325.

[3] E. Marinissen, B. Vermeulen, H. Hollmann, B. Bennetts, "Minimizing Pattern Count for Interconnect Test under a Ground Bounce Constraint", *IEEE Design & Test of Computers*, Mar-Apr 2003, vol. 20, no. 2, pp. 8-18.

[4] H. Hollmann, E. Marinissen, B. Vermeulen, "Optimal Interconnect ATPG Under a Ground-Bounce Constraint", *Proceedings of International Test Conference 2003*, pp. 60-69.

[5] P. Wagner, "Interconnect Testing with Boundary Scan", *Proceedings of International Test Conference 1987*, pp. 52-57.

[6] K. Baker, "Constructive Pattern Generation Heuristic for Meeting SSO Limits", *Proceedings of International Test Conference 2003*, pp. 50-59.