Zero-Aliasing for Modeled Faults

Mody Lempel and Sandeep K. Gupta

CENG Technical Report 94-12

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-2251

April 1994

Zero-Aliasing for Modeled Faults *

Mody Lempel
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782

Sandeep K. Gupta
Department of EE-Systems
University of Southern California
Los Angeles, CA 90089-2562

Abstract

When using Built-In Self Test (BIST) for testing VLSI circuits the circuit response to an input test sequence, which may consist of thousands of bits, is *compacted* into a *signature* which consists of only tens of bits. Usually a linear feedback shift register (LFSR) is used to compact the response and the compaction function is polynomial division. The compacting function is a *many-to-one* function and as a result some erroneous responses may be mapped to the same signature as the good response. This is known as *aliasing.*

In this paper we deal with the selection of a feedback polynomial for the compacting LFSR, given a set of modeled faults, such that an erroneous response resulting from any modeled fault is mapped to a different signature than the good response. Such LFSRs are *zero-aliasing* LFSRs.

In particular, for irreducible and primitive feedback polynomials we present (1) upper bounds on the smallest degree zero-aliasing LFSR; (2) procedures for selecting a zero-aliasing LFSR with the smallest degree; (3) procedures for determining whether a zero-aliasing LFSR of a pre-specified degree exists, and if so, finding one; and (4) procedures for fast selection of a zero-aliasing LFSR. We also show that for all practical applications, a LFSR of degree less than or equal to 53 will always achieve zero-aliasing and we expect that a LFSR of degree less than 21 can always be found.

We analyze the worst case as well as expected time complexity of all the proposed procedures.

Experimental results are presented for practical problem sizes to demonstrate the applicability of the proposed procedures.

Key words: Built-In Self-Test, response compaction, zero-aliasing

*This work was supported by the Advanced Research Projects Agency and monitored by the Federal Bureau of Investigation under Contract No. JFBI90092. The views and conclusions considered in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

1 Introduction

Built-In Self-Test (BIST) is the capability of a circuit to test itself. The idea behind BIST is to create *pattern generators (PGs)* to generate test patterns for the circuit and *response analyzers (RAs)* to *compact* the circuit response to the inputs that are applied. The circuit response, which may consist of thousands of bits, is compacted into a *signature* which consists of only tens of bits. The compacting function is a *many-to-one* function and as a result some erroneous responses might be mapped to the same signature as the good response. This is known as *aliasing.*

When all erroneous responses are mapped to a different signature than the good response, we have *zero-aliasing.* There are two previous schemes to achieve zero-aliasing, that take into account all possible error sequences. The first is by Gupta et al. [7] [14]. In this scheme the RA is a linear feedback shift register (LFSR) and the compacting function is polynomial division of the good response by the feedback polynomial. The scheme requires the *quotient* of the good response to be periodic. This is achieved by proper selection of the LFSR feedback polynomial once the good response is known. They give a bound of $n/2$ on the length of the required register, for a test sequence of length n. The second scheme, due to Chakrabarty and Hayes [5], uses non-linear logic to detect any error in the response. The number of memory cells in their RA is $\lceil \log n \rceil$ but they have no bound on the extra logic required to implement their scheme.

The major difference between our scheme and the aforementioned zero-aliasing schemes is that we target a specific set of possible faults and try to achieve zero-aliasing for the error sequences resulting only from these faults. We *do not* try to recognize all possible error sequences, mainly because most of them will *never* occur. The fault model lets us focus on the *probable* error sequences. As a result, we use less hardware than the aforementioned schemes.

A previous method for finding zero-aliasing feedback polynomials for modeled faults was presented by Pomeranz et al. [13]. Different heuristics for finding a zero-aliasing polynomial are suggested, but these heuristics will not necessarily find an irreducible or primitive polynomial, which is very important if the register is also to function as a PG. They do not give bounds on the resulting or necessary degrees of their zero-aliasing polynomials, nor do they present results on the complexity of their methods.

The PGs and RAs are usually implemented from existing registers. Some registers are configured as PGs to generate tests for some bodies of logic and reconfigured as RAs to test other

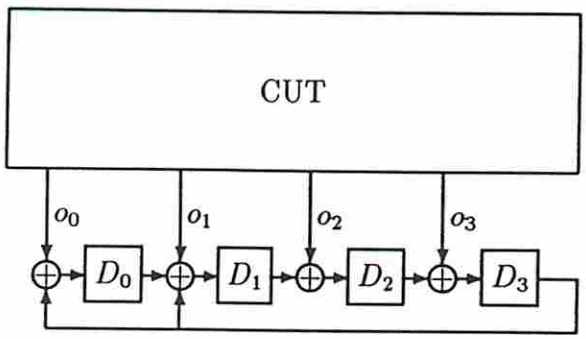

Figure 1: A MISR-based RA for a 4 output CUT. The feedback polynomial is $f(x) = x^4 + x + 1$.

bodies of logic. When the same design serves both purposes, the overhead of a reconfigurable design is saved. In such a scheme a LFSR is used as a PG and a *multiple input shift register (MISR)* is used as a RA. An example of a MISR-based RA is shown in Figure 1. The register is configured as a shift register where the input to each cell is an XOR function of the previous cell, an output bit of the *circuit under test (CUT)* and, depending on the *linear feedback function*, a feedback bit. Number the cells of a k stage MISR $D_0, D_1, \ldots, D_{k-1}$, with the feedback coming out of cell D_{k-1}. The feedback function is represented as a polynomial $f(x) = x^k + \sum_{i=0}^{k-1} f_i x^i$ and the feedback feeds cell D_i iff $f_i = 1$. The feedback polynomial of the MISR in Figure 1 is $f(x) = x^4 + x + 1$. The difference between a LFSR and a MISR are the extra inputs connected to the outputs of the CUT. If both the PG (LFSR) and the RA (MISR) use the same feedback polynomial, then the overhead of reconfigurable polynomials is saved. In a previous paper [11] we showed how to select the feedback polynomial for a PG; in this paper we deal with selecting the feedback polynomial for a RA.

The compacting function of a MISR is polynomial division over $GF[2]$. The *effective output polynomial* is divided by the feedback polynomial. The signature is the remainder of the division. If the CUT has k outputs, it has k output sequences. Denote these sequences by $o_0, o_1, \ldots, o_{k-1}$, where o_i feeds D_i. If the input sequence is of length n, then each o_i can be viewed as a polynomial $O_i = \sum_{j=0}^{n-1} o_{i,j} x^{n-1-j}$, where $o_{i,j}$ is the output value of the i-th output at time j. The effective polynomial is then

$$O = \sum_{l=0}^{k-1} O_l x^l.$$

Our objective is to select a feedback polynomial for the compacting MISR, given a set of modeled faults, such that an erroneous response resulting from any modeled fault is mapped to

a different signature than the signature of the good response.

For a CUT with few outputs, the available register might be too short to achieve zero-aliasing. In this case we need to lengthen the register by adding flip-flops. To keep the hardware overhead at a minimum, we want to add as few flip-flops as possible, hence we are interested in a feedback polynomial of smallest degree that achieves our objective. When a register is to serve both as a PG and a RA, it is advantageous to have the feedback polynomial of the same degree as the available register, hence we are interested in a feedback polynomial of a pre-specified degree. At times, we might want to find a feedback polynomial fast, even if the resulting MISR requires extra flip-flops over the optimum.

We assume the following test scenario. The input sequence to the CUT has been designed so that the effective output polynomial due to any target fault is different from the effective polynomial of the good response. Let r be the effective polynomial of the good response, then the effective polynomial due to fault i can be represent as $r + h_i$. By the linearity of the remaindering operation, we get a different remainder for this erroneous polynomial iff h_i is not divisible by the feedback polynomial. We assume we are given the error polynomials for each of the target faults.

The problem we deal with in this paper is the following: given a set of polynomials $H = \{h_1, h_2, \ldots, h_{|H|}\}$ find a polynomial that is relatively prime to all the polynomials of H. Such a polynomial will be referred to as a *non-factor* of H. If a non-factor is used as the feedback polynomial for the compacting MISR, zero-aliasing is achieved for the set of target faults. In particular, for irreducible and primitive feedback polynomials we present (1) upper bounds on the smallest degree zero-aliasing MISR; (2) procedures for selecting a zero-aliasing MISR with the smallest degree; (3) procedures for determining whether a zero-aliasing LFSR of a pre-specified degree exists, and if so, finding one; and (4) procedures for fast selection of a zero-aliasing MISR. We analyze the worst case as well as expected time complexity of the proposed procedures.

A note on notation. When using logarithmic notation, $\ln x$ will denote the *natural logarithm* of x and $\log x$ will denote the *base* 2 *logarithm* of x. The polynomials $\{h_i\}$ represent the error polynomials. The degree of h_i is represented by d_i. The product of the polynomials in H is denoted by h, and the degree of h is d_h. For each h_i, the product of the distinct, degree j, irreducible factors of h_i is denoted by $g_{i,j}$, with $d_{i,j}$ being the degree of $g_{i,j}$. The product, over all i, of the polynomials $g_{i,j}$ is denoted by g_j. The non-factor we seek will be referred to as a

with d_a representing the degree of a.

The rest of this paper is organized as follows. In Section 2 we establish upper bounds on the degree of a non-factor. In Section 3 we review polynomial operations over $GF[2]$ and their complexities. Section 4 presents procedures for finding a non-factor of smallest degree for the set H. Section 5 presents procedures for finding a non-factor of a pre-specified degree and for finding a non-factor fast. We also discuss the effectiveness of conducting an exhaustive search for a least degree non-factor. Section 6 presents some experimental data. We conclude in Section 7.

2 Bounds on the least degree non-factor of a set of polynomials

Consider the following problem.

Problem 1: Let H be a set of $|H|$ polynomials $h_1, \ldots, h_{|H|}$ with $deg(h_i) = d_i$ and for all $1 \leq i \leq |H|$, $d_i \leq n$. Let $h = \prod_{i=1}^{|H|} h_i$. Then $deg(h) = \sum_{i=1}^{|H|} d_i = d_h \leq |H|n$. Give an upper bound $s(d_h)$ on the degree of an irreducible polynomial and an upper bound $p(d_h)$ on the degree of a primitive polynomial that does not divide h, i.e. there exists an irreducible (primitive) polynomial of degree at most $s(d_h)$ ($p(d_h)$) that does not divide h. □

Similarly, let $es(H)$ ($ep(H)$) be the *expected* degree of an irreducible (primitive) polynomial that is a non-factor of H.

The bounds $s(d)$ and $p(d)$ will be referred to as the *worst case bounds* while the bounds $es(H)$ and $ep(H)$ will be referred to as the *expected bounds.* We first establish the worst case bounds and then proceed with the expected bounds.

2.1 The worst case bounds

For the bound on $s(d)$ we follow [10]. Let $I_2(j)$ denote the number of irreducible polynomials of degree j over $GF[2]$. The degree of the product of all irreducible polynomials of degree j is $jI_2(j)$. Let $s(d)$ denote the least integer such that $\sum_{j=1}^{s(d)} jI_2(j) > d$. Let $Q_{s(d)}$ be the product of all the irreducible polynomials of degree less than or equal to $s(d)$. The degree of $Q_{s(d)}$ is greater

than d. Replacing d with d_h, $Q_{s(d_h)}$ has at least one root that is not a root of h, hence $Q_{s(d_h)}$ has at least one irreducible factor that is not a factor of h. Thus, $s(d_h)$ is an upper bound on the degree of an irreducible polynomial that is relatively prime to all the polynomials in the set H. The following lemma provides a bound on $s(d_h)$.

Lemma 1: [10, Lemma 4, p.293]

$$s(d) \leq \lceil \log(d+1) \rceil$$

□

We turn to find the bound on $p(d)$. The number of primitive polynomials of degree m over $GF[2]$ is

$$\frac{1}{m}\phi(2^m - 1)$$

where $\phi(q)$ is the *Euler function* denoting the number of integers less than and relatively prime to q and ([12, p. 37])

$$\phi(q) = q\prod_{i=1}^{l}(1 - 1/p_i)$$

where the p_i's are all the distinct prime factors of q.

Lemma 2: [16, p. 173]

$$\phi(q) \geq e^{-\gamma}\frac{q}{\ln\ln q + \frac{5}{2e^{\gamma}\ln\ln q}}$$

for all $q \geq 3$ with the only exception being $q = 223,092,870$ (the product of the first nine primes), for which $\frac{5}{2}$ is replaced by 2.50637. $\gamma = 0.577215665\ldots$ is *Euler's constant.* □

For $q > 65$ we have

$$\phi(q) > \frac{q}{3\ln\ln q} > \frac{q}{2.08\log\log q}. \tag{1}$$

To help us derive the bound on $p(d)$ we introduce the value $\tau(t)$. Let $\tau(t)$ denote the least integer such that the ratio between $\tau(t)$ times the number of primitive polynomials of degree $\tau(t)$ and t times the number of irreducible polynomials of degree t is greater than 1, i.e.

$$\frac{\phi(2^{\tau(t)} - 1)}{tI_2(t)} > 1.$$

Lemma 3: [10, Lemma 3, p. 293] For $t \geq 3$

$$2^{t-1} < tI_2(t) \leq 2^t - 2.$$

□

Lemma 4: For $t > 6$, $\tau(t) \leq t + \lceil 1 + \log\log 2t \rceil$

Proof: For $q \geq 7$, the function $\frac{q}{2.08 \log\log q}$ is an increasing function. Also,

$$\frac{q}{2.08\log\log q} > \frac{q-1}{2.08\log\log q} \quad \text{and} \quad \frac{q-1}{2.08\log\log(q-1)} > \frac{q-1}{2.08\log\log q}$$

hence,

$$\frac{q-1}{2.08\log\log(q-1)} + 1 > \frac{q}{2.08\log\log q}.$$

Let $\tau'(t)$ be the least integer such that $\frac{2^{\tau'(t)}}{2.08\log\log 2^{\tau'(t)}} > 2^t$. Thus, using the above relation,

$$\frac{2^{\tau'(t)} - 1}{2.08\log\log(2^{\tau'(t)} - 1)} > 2^t - 1.$$

By Equation (1) we get

$$\phi(2^{\tau'(t)} - 1) > 2^t - 1$$

and due to Lemma 3

$$\phi(2^{\tau'(t)} - 1) > tI_2(t).$$

Thus, by the definition of $\tau(t)$, we have that $\tau'(t) \geq \tau(t)$. To bound $\tau(t)$ from above, we solve for $\tau'(t)$.

By definition, $\tau'(t)$ must satisfy

$$\begin{aligned}
\tfrac{2^{\tau'(t)}}{2.08\log\tau'(t)} &> 2^t && \Rightarrow \\
2^{\tau'(t)-t} &> 2.08\log\tau'(t) && \Rightarrow \\
\tau'(t) - t &> \log 2.08 + \log\log\tau'(t)
\end{aligned}$$

By setting $\tau'(t) = t + \lceil 1 + \log\log 2t \rceil$, we have

$$\tau'(t) - t = \lceil 1 + \log\log(2t) \rceil$$

and for $t > 6$

$$\lceil 1 + \log\log(2t) \rceil > \log 2.08 + \log\log \tau'(t)$$

□

Lemma 5: Let $p(d)$ denote the least integer such that $\sum_{j=1}^{p(d)} \phi(2^j - 1) > d$, then for $d > 65$

$$p(d) \leq \lceil \log(d+1) \rceil + \lceil 1 + \log\log(2\lceil \log(d+1) \rceil) \rceil.$$

Proof: By the definition of $\tau(t)$ and Lemma 4

$$\sum_{j=1}^{j=\lceil \log(d+1) \rceil + \lceil 1+\log\log(2\lceil \log(d+1) \rceil) \rceil} \phi(2^j - 1) > \sum_{j=1}^{j=\lceil \log(d+1) \rceil} j I_2(j).$$

By Lemma 1 and the definition of $s(d)$

$$\sum_{j=1}^{j=\lceil \log(d+1) \rceil} j I_2(j) \geq \sum_{j=1}^{s(d)} j I_2(j) > d.$$

□

Example 1: In Table 1 the values of $\phi(2^m - 1)$ (the degree of the product of all the primitive polynomials of degree m) and $\sum_{i=2}^{m} \phi(2^i - 1)$ (the degree of the product of all the primitive polynomials of degree $2 \leq i \leq m$) are tabulated for $2 \leq m \leq 53$. As long as d is less than the maximum value in the table, $p(d)$ can be read from the table, instead of using Lemma 5. For example, if the number of modeled faults in the CUT is $|H| = 10^4$ and the length of the test sequence is $n = 10^6$, then $d_h \leq 10^{10}$. The degree of the product of all primitive polynomials with degree less than or equal to 33 is the first which is greater than 10^{10}, hence $p(d_h) \leq 33$. Thus, a zero-aliasing LFSR with a primitive feedback polynomial, of degree at most 33, exists for the CUT. On the other hand, using the bound of Lemma 5 we get $p(d_h) \leq 38$. □

A closer look at Table 1 shows that the product of all the primitive polynomials of degree less than or equal to 53 has degree D greater than $1.4 \cdot 10^{16}$. Thus, as long as the product of the number of faults and the test sequence length is less than D (which is the case for all practical test applications) a zero-aliasing MISR of degree less than or equal to 53 exists.

m	$\phi(2^m-1)$	$\sum_{i=2}^{m}\phi(2^i-1)$	m	$\phi(2^m-1)$	$\sum_{i=2}^{m}\phi(2^i-1)$
2	2	2	28	132,765,696	343,973,802
3	6	8	29	533,826,432	877,800,234
4	8	16	30	534,600,000	1,412,400,234
5	30	46	31	2,147,483,646	3,559,883,880
6	36	82	32	2,147,483,648	5,707,367,528
7	126	208	33	6,963,536,448	12,670,903,976
8	128	336	34	11,452,896,600	24,123,800,576
9	432	768	35	32,524,320,000	56,648,432,576
10	600	1,368	36	26,121,388,032	82,769,820,608
11	1,936	3,304	37	136,822,635,072	219,592,455,680
12	1,728	5,032	38	183,250,539,864	402,842,995,544
13	8,190	13,222	39	465,193,834,560	868,036,830,104
14	10,584	23,806	40	473,702,400,000	1,341,739,230,104
15	27,000	50,806	41	2,198,858,730,832	3,540,597,960,936
16	32,768	83,574	42	2,427,720,325,632	5,968,318,286,568
17	131,070	214,644	43	8,774,777,333,880	14,743,095,620,448
18	139,968	354,612	44	8,834,232,287,232	23,577,327,907,680
19	524,286	878,898	45	28,548,223,200,000	52,125,551,107,680
20	480,000	1,358,898	46	45,914,084,232,320	98,039,635,340,000
21	1,778,112	3,137,010	47	140,646,443,289,600	238,686,078,629,600
22	2,640,704	5,777,714	48	109,586,090,557,440	348,272,169,187,040
23	8,210,080	13,987,794	49	558,517,276,622,592	906,789,445,809,632
24	6,635,520	20,623,314	50	656,100,000,000,000	1,562,889,445,809,632
25	32,400,000	53,023,314	51	1,910,296,842,179,040	3,473,186,287,988,672
26	44,717,400	97,740,714	52	2,338,996,194,662,400	5,812,182,482,651,072
27	113,467,392	211,208,106	53	9,005,653,101,120,000	14,817,835,583,771,072

Table 1: Number of primitive elements in $GF[2^m]$ and accumulated number of primitive elements in $GF[2^2]\ldots GF[2^m]$

2.2 The expected bounds

In deriving the expected bounds we assume that the polynomials $\{h_i\}$ are random polynomials. Denote the product of the distinct irreducible factors of degree j of h_i by $g_{i,j}$. Denote the number of distinct irreducible factors of h_i, of degree j, by v. The value of v can range from 0 to $min\{\lfloor d_i/j \rfloor, I_2(j)\}$.

Lemma 6: For $j \geq 2$, the *expected value* of v (the number of irreducible factors of $g_{i,j}$) is less than or equal to $\frac{1}{j}$.

Proof: Let $IR_2(j) = \{p_i\}_{i=1}^{I_2(j)}$ be the set of irreducible polynomials of degree j over $GF[2]$. For a given polynomial q, of degree greater or equal to j, define the *indicator function* $d(p_i, q)$ to be one if p_i divides q and zero otherwise. The probability that a polynomial of degree j divides a random polynomial of degree greater or equal to j is 2^{-j}, hence the probability that $d(p_i, q) = 1$ is equal to 2^{-j}. Thus

$$\begin{aligned} E[v] &= E\left[\sum_{i=1}^{I_2(j)} d(p_i, q)\right] \\ &= \frac{I_2(j)}{2^j} \\ &< \frac{1}{j} \end{aligned}$$

□

The same type of analysis can be used to bound $Var[v]$, the variance of v, and σ_v, the standard deviation of v.

Lemma 7: For $j \geq 2$, the *variance* of the number of irreducible factors of $g_{i,j}$ is less than $\frac{2}{j}$. The *standard deviation* is less than $\left(\frac{2}{j}\right)^{\frac{1}{2}}$.

Proof: The variance of v is given by $Var[v] = E[v^2] - E[v]^2$.

$$\begin{aligned} E[v^2] &= E\left[\left(\sum_{i=1}^{I_2(j)} d(p_i, q)\right)^2\right] \\ &= E\left[\sum_{i=1}^{I_2(j)} d(p_i, q)^2 + 2 \sum_{\substack{p_i, p_k \in IR_2(j) \\ i<k}} d(p_i, q) d(p_k, q)\right] \\ &= \sum_{i=1}^{I_2(j)} E\left[d(p_i, q)^2\right] + 2 \sum_{\substack{p_i, p_k \in IR_2(j) \\ i<k}} E\left[d(p_i, q) d(p_k, q)\right] \\ &< \frac{I_2(j)}{2^j} + \frac{2I_2(j)^2}{2^{2j}} \end{aligned}$$

$$< \frac{1}{j} + \frac{2}{j^2}.$$

For $j \geq 2$ we have

$$E[v^2] < \frac{2}{j}$$

and,

$$Var[v] = E[v^2] - E[v]^2 < E[v^2] < \frac{2}{j} \text{ and } \sigma_v < \left(\frac{2}{j}\right)^{\frac{1}{2}}.$$

□

Having computed the mean and variance of the number of irreducible factors of degree j per polynomial, we can compute a *confidence measure* for these results.

Lemma 8: For $j \geq 8$, the expected number of polynomials $g_{i,j}$ with more than 5 (50) factors is less than $|H|/100$ ($|H|/10,000$).

Proof: Using the *Chebyshev inequality* [8, p. 376]

$$Pr(|v - E[v]| \geq c\sigma_v) \leq 1/c^2$$

for $j \geq 8$ the probability that v is greater than 5 is less than 0.01. Using this result we can define a second random process in which the random variable x is 1 iff v is greater than 5 and 0 otherwise. This process is a *Bernoulli experiment* [6, Sec. 6.4]. The expected number of $g_{i,j}$'s with more than 5 factors is upper bounded by $|H|/100$, as is the variance. Similarly, the probability that v is greater than 50 is less than 0.0001 and the expected number of $g_{i,j}$'s with more than 50 factors is bounded by $|H|/10,000$. □

Lemma 9: The expected degree of the smallest irreducible non-factor of the set of polynomials H is bounded from above by $\lceil \log |H| \rceil + 1$.

Proof: Denote the product of the polynomials $g_{i,j}$, $1 \leq i \leq |H|$, by g_j. By Lemma 6, the expected number of (not necessarily distinct) factors of g_j is less than $|H|/j$. The smallest j for

which $I_2(j)$ exceeds this value is an upper bound on the expected degree d_a of a non-factor of H. Thus, the upper bound δ on d_a satisfies

$$\frac{|H|}{\delta} < \frac{2^{\delta-1}}{\delta} < I_2(\delta) \;\Rightarrow\; |H| < 2^{\delta-1}$$

and $d_a \leq \lceil \log |H| \rceil + 1$. □

By applying Lemma 4 on the result of Lemma 9, we have

Corollary 10: The expected degree of the smallest primitive non-factor of the set of polynomials H is bounded from above by $2 + \lceil \log |H| \rceil + \lceil \log \log(2 + 2\lceil \log |H| \rceil) \rceil$. □

Example 2: Using the numbers of Example 1, let $|H| = 10^4$ and $n = 10^6$. The first j for which $\phi(2^j - 1)/j$ exceeds $|H|/j$ is $j = 14$ (Table 1), hence we expect to find a zero-aliasing MISR with a primitive feedback polynomial of degree less than or equal to 14, as opposed to the worst case of 33. Corollary 10 would give us an upper bound of 19. □

As the expected bound is a only a function the number of faults and not the length of the test sequence, the expected degree of a zero-aliasing MISR will *never* exceed 53. In fact, as long as the number of faults is less than 1 million, we expect to find a zero-aliasing MISR of degree less than or equal to 21.

3 Polynomial operations in $GF[2]$

In search for a (least degree) non-factor of H we use procedures that sift the factors of the same degree from a given polynomial. These procedures are based on the following lemma.

Lemma 11: [12, Lemma 2.13, p.48] $x^{2^m} - x$ is the product of all irreducible polynomials of degree l, where l is a divisor of m. □

Thus, a basic step in finding the distinct irreducible factors of a polynomial $b(x)$ is the computation of

$$g(x) = gcd(b(x), x^{2^m} - x).$$

The result of this operation is the product of all the irreducible factors of degree l, where $l|m$, of $b(x)$. For most polynomials $b(x)$ of interest to us, $2^m >> deg(b(x))$. Therefore, we first compute

$$r(x) \equiv (x^{2^m} - x) \bmod b(x)$$

and then

$$g(x) = gcd(b(x), r(x)).$$

We first discuss the complexity of polynomial operations in $GF[2]$ and then review a well known approach to reduce x^{2^m} modulo $b(x)$.

3.1 Polynomial multiplication, division and gcd

The complexity of a polynomial *gcd* operation is $O(M(s) \log s)$ [1, pp. 300-308], where s is the degree of the larger polynomial operand and $M(s)$ is the complexity of polynomial multiplication, where the product has degree s. The complexity of polynomial division is also $O(M(s))$ [1, Ch. 8]. Hence, it is crucial to find an efficient multiplication algorithm.

We consider two multiplication algorithms. Both algorithms are based on FFT techniques [1, Ch. 7], [6, Ch. 32]. For these algorithms to work they need a root of unity whose order has small prime factors. In most cases, when the product polynomial has degree s, a root of order $2^m > s$ is used. This poses a problem, since fields of characteristic 2 do not contain such roots.

The first algorithm is due to Schönhage [17]. It uses roots of order 3^{m+1} to multiply polynomials of degree $s < 3^m$. Its complexity is $O(s \log s \log \log s)$.

The second algorithm is suggested by Cormen et al. [6, p. 799]. To multiply polynomials of degree $s/2 < 2^{m-1}$ they suggest working in the field $GF[p]$ where p is a prime of the form $\beta \cdot 2^m + 1$. The multiplication is done over $GF[p]$ and the coefficients of the product are reduced modulo 2 to give the correct result over $GF[2]$. The question that naturally arises is how big is p. The best provable bound on the size of p is that it is less than $2^{5.5 \cdot m}$ [9]. It is widely believed that $\beta = O(m^2)$ [16, p. 221]. The complexity, per multiplication, of the Cormen algorithm is $O(s \log s)$ operations in $GF[p]$. If the word size of a machine is greater than $\log p$, then word operations can be performed in $O(1)$ machine instructions. To further cut down on time, we construct logarithmic tables relative to a primitive element, α, of $GF[p]$ for multiplication and addition. With these tables, multiplication modulo p is addition modulo p of the logarithms.

m	p	α	ω	m	p	α	ω	m	p	α	ω
6	193	5	11	11	12289	11	7	16	65537	3	3
7	257	3	9	12	12289	11	41	17	786433	10	8
8	257	3	3	13	40961	3	12	18	786433	10	5
9	7681	17	62	14	65537	3	15	19	5767169	3	12
10	12289	11	49	15	65537	3	9	20	7340033	3	5

Table 2: Least prime p, of the form $\beta 2^m + 1$, with smallest generator α and 2^m-th root of unity ω.

The addition table stores the *Jacobi logarithm* $Z(i)$ [12, p. 69], i.e. $Z(i) = \log_\alpha(\alpha^i + 1)$ where $\log_\alpha(0)$ is defined as ∞. Thus, adding $\alpha^\mu + \alpha^\nu$, for $\mu \leq \nu$, is actually the multiplication operation $\alpha^\mu(1 + \alpha^{\nu-\mu})$ and the logarithm of the result equals $\mu + Z(\nu - \mu)$.

Table 2 shows the smallest p for $m = 6, 7, \ldots, 20$, along with the smallest primitive element α and the smallest $2^m - th$ root of unity ω.

In the sequel we shall use the notation $O(M(s))$ for the complexity of polynomial multiplication. Whenever possible it will mean $s \log s$, otherwise it should be taken as $s \log s \log\log s$. Similary the notation $L(s)$ will denote either $\log s$ or $\log s \log\log s$, as appropriate.

3.2 x^{2^m} modulo $b(x)$ and x^t modulo $b(x)$

We review a well known approach [3] [15] to find the remainder of x^{2^m} when divided by $b(x)$ without actually carrying out the division.

Let $s = deg(b(x))$ and let $R^{(j)}(x) = \sum_{i=0}^{s-1} R_i^{(j)} x^i \equiv x^{2^j} \bmod b(x)$. Then

$$R^{(j)^2}(x) \equiv (x^{2^j})^2 \bmod b(x) \equiv x^{2^{j+1}} \bmod b(x) = R^{(j+1)}(x).$$

Squaring over $GF[2]$ is easy ($r(x)^2 = r(x^2)$), thus by repeatedly squaring and reducing modulo $b(x)$, in time $O(mM(s))$ we can compute $R^{(m)}(x)$. Note that the maximum degree $R^{(j)^2}(x)$ can have is $2(s-1)$. Once we have $R^{(m)}(x)$, we can compute $g(x) = gcd(b(x), R^{(m)}(x) - x)$ in time $O(M(s)\log s)$. Overall time needed to compute $g(x)$ is $O(mM(s) + M(s)\log s)$.

Let $t = \sum_{j=0}^{m} t_j 2^j$. To compute $r(x) = x^t \bmod b(x)$ we compute $R^{(m)}(x)$ as described above. This costs $O(mM(s))$. We initialize $r(x)$ to equal 1. As we compute $R^{(m)}(x)$, for each inter-

mediate value $R^{(j)}(x)$ for which $t_j = 1$, we set $r(x)$ to the product $r(x)R^{(j)}(x) \bmod b(x)$. Each such computation costs $O(M(s))$, hence the cost of computing $x^t \bmod b(x)$ is $O(mM(s))$.

4 Finding a non-factor of smallest degree for a given set of polynomials

After establishing the bounds on the least degree non-factor of H in Section 2, this section addresses the question of finding a least degree non-factor for H.

Problem 2: Given a set of polynomials $H = \{h_1(x), h_2(x), \ldots, h_{|H|}(x)\}$ with $deg(h_i) = d_i \leq n$, let

$$h(x) = \prod_{i=1}^{|H|} h_i(x) \ , \ deg(h) = \sum_{i=1}^{|H|} d_i = d_h. \tag{2}$$

Find an irreducible (primitive) polynomial $a(x)$, with $deg(a) = d_a$, such that

1. For all $1 \leq i \leq |H|$, $h_i \not\equiv 0 \bmod a$ (equivalently, $h \not\equiv 0 \bmod a$).
2. For all irreducible (primitive) polynomials $b(x)$, with $deg(b) < d_a$, $h \equiv 0 \bmod b$ (or equivalently, there exists an i for which $h_i \equiv 0 \bmod b$).

□

One way of solving the problem is by factoring the polynomials of H. This would require too much work, since we do not need to know all the factors in order to find a non-factor. We only need to know the "small" factors.

In this section we present algorithms for solving Problem 2 and analyze their complexity. The complexity is given in two forms. The first is with *worst case complexity bounds*, referred to as the *worst case complexity*. The second is with *expected complexity bounds*, referred to as the *expected complexity*. The expected complexity is a refinement of the worst case complexity based on the *expected size* of the results from our procedures.

By Lemmas 1 and 5 (Section 2), we have an upper bound $u = s(d_h)$ or $u = p(d_h)$ on d_a, depending on whether we are looking for an irreducible or a primitive non-factor. Using this bound, we begin our search process, which is made up of three phases.

1. For all $h_i \in H$, find $g_{i,j}(x)$, the product of all distinct irreducible (primitive) factors of h_i, of degree j.

2. Having found the polynomials $g_{i,j}$, determine whether all irreducible (primitive) polynomials of degree j are factors of H.

3. If not all irreducible (primitive) polynomials of degree j are factors of H, find one that is not.

The worst case complexities of the three phases for the irreducible case are $O(|H|u^2M(n))$, $O(|H|^2M(n)\log n)$ and $O(|H|^2n^2u^2M(u))$. The dominant term is $O(|H|^2n^2u^2M(u))$. The worst case complexities of the three phases for the primitive case are $O(|H|u^3M(n))$, $O(|H|^2 \cdot M(n)\log n)$ and $O(|H|^2n^2u^3M(u)\log\log u)$. The dominant term is $O(|H|^2n^2u^3M(u)\log\log u)$.

The expected complexity of the first two phases are $O(|H|u^2M(n))$ and $O(|H|\log|H|u^2 \cdot L(n)\log n)$. The expected complexity for the third phase is $O(|H|\log|H|d_aM(d_a))$ to find an irreducible non-factor and $O(|H|\log|H|d_a^2\log\log d_aM(d_a))$ to find a primitive non-factor. The dominant term is $O(|H|u^2M(n))$.

The worst case complexity is a function of $|H|^2n^2$ multiplied by terms that are logarithmic in $|H|$ and n whereas the expected complexity is a function of $|H|n$ multiplied by terms that are logarithmic in $|H|$ and n.

4.1 The product of all distinct factors of the same degree for a given polynomial

Given the polynomial $h_i(x)$ and the upper bound u, we wish to compute $g_{i,j}$, the product of all distinct factors of h_i of degree j, for $1 \leq j \leq u$. The procedure for computing the polynomials $g_{i,j}$ is given in Figure 2. The polynomials $g_{i,j}$ are computed in three steps. First, for $u/2 < j \leq u$, compute $g_{i,j} = gcd(h_i(x), x^{2^j} - x)$. Each $g_{i,j}$ is a product of all the distinct irreducible factors of $h_i(x)$ of degree j and of degree l, where $l|j$.

When j is less than or equal to $u/2$, we have $2j \leq u$. By Theorem 11, $g_{i,2j}$ contains the product of all irreducible factors of degree l, where $l|j$, of h_i. Since the degree of $g_{i,2j}$ is (much) less than the degree of h_i, it is more efficient to compute $g_{i,j}$ from $g_{i,2j}$ than from h_i. Thus, in Step 2, for $1 < j \leq u/2$ compute $g_{i,j} = gcd(g_{i,2j}, x^{2^j} - x)$.

Procedure 1: *distinct_factors*(h_i)

1. For $(j = u\ ; j > u/2\ ; j - -)$
   (a) $g_{i,j} = gcd(h_i(x), x^{2^j} - x)$
2. For $(j = \lfloor u/2 \rfloor\ ; j > 0\ ; j - -)$
   (a) $g_{i,j} = gcd(g_{i,2j}, x^{2^j} - x)$
3. For $(j = 2; j \leq u; j + +)$
   (a) For all $l|j$
      i. $g_{i,j} = g_{i,j}/g_{i,l}$

Figure 2: Procedure *distinct_factors*(h_i). Computes the product of all distinct factors of degree j, for $1 \leq j \leq u$, of the polynomial h.

At the end of Step 2, each $g_{i,j}$ contains *all* the factors of degree $l|j$ of h_i. To sift out the factors of degree less than j from $g_{i,j}$, we need to divide $g_{i,j}$ by $g_{i,l}$, where l ranges over the set of divisors of j. This is carried out in Step 3.

Procedure *distinct_factors*() is not enough when we are looking for a primitive non-factor. At the end of the procedure, each $g_{i,j}$ is the product of all distinct irreducible polynomials of degree j, that are factors of h_i. From $g_{i,j}$ we need to sift out the non-primitive factors. Before describing this aspect, we introduce the notion of *maximal divisors*.

Definition 1: Let $q = \prod_{i=1}^{r} p_i^{e_i}$, with $p_1, \ldots, p_r$ being the distinct prime factors of q. The set of *maximal divisor* of q is the set $md(q) = \{m_i\}_{i=1}^{r}$ where $m_i = q/p_i$. □

For example, $20 = 2^2 \cdot 5$, hence $md(20) = \{10, 4\}$. $16 = 2^4$, therefore $md(16) = \{8\}$. Since 7 has only one prime factor, $md(7) = \{1\}$.

A polynomial over $GF[q]$ of degree m is irreducible iff it divides $x^{q^m-1} - 1$ and does not divide $x^{q^k-1} - 1$ for all divisors k of m. It is primitive of degree m iff it is irreducible and does not divide $x^l - 1$ for all l in $md(q^m - 1)$ [12, Ch. 3]. Procedure *distinct_primitives*(), shown in Figure 3, sifts out the non-primitive factors of $g_{i,j}$.

Lemma 12:

Procedure 2: *distinct_primitive*$(g_{i,j})$
/* sifts out the non-primitive factors of $g_{i,j}$ */

1. For all l in $md(2^j - 1)$
   (a) $q_l = gcd(g_{i,j}, x^l - 1)$
   (b) $g_{i,j} = g_{i,j}/q_l$

Figure 3: Procedure *distinct_primitive*$(g_{i,j})$. Sifts out the non-primitive factors of $g_{i,j}$.

1. The complexity of Procedure *distinct_factors*() is $O(u^2 M(n))$.
2. The complexity of Procedure *distinct_primitive*() is $O(u^3 M(n))$.
3. The complexity of the first phase is $O(|H|u^2 M(n))$ for the irreducible case and $O(|H|u^3 \cdot M(n))$ for the primitive case.

In the above expressions $u = s(d_h)$ for the irreducible case and $u = p(d_h)$ for the primitive case.

Proof:

1. The worst case complexity of Procedure *distinct_factors*() is as follows. In Step 1, the procedure performs $u/2$ *gcd* computations involving h_i. The complexity of each *gcd* computation is $O(jM(d_i) + M(d_i)\log d_i)$. Thus the total work for the first stage is

$$\sum_{j=\lceil u/2 \rceil}^{u} O(jM(d_i) + M(d_i)\log d_i) = O\left(\frac{u}{2} M(d_i)\left(\frac{3u}{4} + \log d_i\right)\right) = O(u^2 M(n)).$$

   In Step 2 the procedure carries out $u/2$ *gcd* operations. The work required for this step is

$$\sum_{j=1}^{\lfloor u/2 \rfloor} O(jM(d_{i,2j}) + M(d_{i,2j})\log d_{i,2j}) = O(u^2 M(n)).$$

   In Step 3, for every element of the sets of divisors, the procedure performs a division operation. The cost expression is

$$\sum_{j=1}^{u} \sum_{l|j} O(M(d_{i,j})) < \sum_{j=1}^{u} O(jM(d_{i,j})) = O(u^2 M(n)).$$

2. The complexity of Procedure *distinct_primitive*() is as follows. Each iteration of Procedure *distinct_primitives*() reduces $(x^k - 1)$ modulo $g_{i,j}$ and performs one *gcd* and one division operation. The cost of each iteration is $O(jM(d_{i,j}) + M(d_{i,j})\log(d_{i,j}))$. There are $md(2^j - 1)$ iterations, with $md(2^j - 1) < j \leq u$, and we run the procedure u times. Therefore, the additional work for the primitive case is bounded by $O(u^3 M(d_i))$.

   In most cases, the values $d_{i,j}$ will be (much) less than n, hence the actual work will be much less then $O(u^2 M(n))$ and the dominant factor will be Step 1 of Procedure *distinct_factors*().

3. Over the set H, based on 1 and 2, the complexity of the first phase is $(|H|u^2 M(n))$ for the irreducible case and $(|H|u^3 M(n))$ for the primitive case. The value of u is either $s(d_h)$ or $p(d_h)$ corresponding to either the irreducible or primitive case.

□

Lemma 13: The expected complexity of the first phase is $O(|H|u^2 M(n)))$ with u equal to either $es(H)$ or $ep(H)$.

Proof: The expected complexity of Procedure *distinct_factors*() is dominated by the complexity of Step 1, which is $O(u^2 M(n))$. The difference in the complexity of the other steps, over the worst case, comes from using the expected size of the $d_{i,j}$s, instead of their worst case size, which is equal to n. The expected complexity of the procedure (including Procedure *distinct_primitive*()), over the set H is, thus, $O(|H|u^2 M(n))$, with u equal to either $es(H)$ or $ep(H)$. □

4.2 The number of all distinct factors, of the same degree, for a set of polynomials

After the first phase, for all degrees $1 \leq j \leq u$, we have $|H|$ polynomials $g_{i,j}$, each a product of the distinct irreducible (primitive) factors of degree j of h_i. Some of the $g_{i,j}$'s might equal 1 while some pairs might have factors in common. Our goal is to find a least degree non-factor of H. The first thing we must determine is whether all irreducible polynomials of degree j appear in $g_j = \prod_i^{|H|} g_{i,j}$. This is the second of our three phases (page 15). A simple test is to compare

$\frac{deg(g_j)}{j} = \frac{\sum_i deg(g_{i,j})}{j}$ with $I_2(j)$. If $\frac{deg(g_j)}{j} < I_2(j)$ then there is a non-factor of degree j. For the primitive case we compare with $\frac{\phi(2^j-1)}{j}$.

If $\frac{deg(g_j)}{j} \geq I_2(j)$, the only way to determine whether all irreducible (primitive) polynomials of degree j are factors of g_j is to find those factors that appear in more than one of the $g_{i,j}$'s and to eliminate all their appearances except for one.

We considered two methods for removing repeated factors. The first is referred to as the *lcm method* and the second is referred to as the *gcd method.* The *lcm method* will be shown to be faster, but it also requires more space, which might not be available.

In the *lcm method* we first sort the $g_{i,j}$s according to their degrees and then place them in the sets s_k, where $g_{i,j} \in s_k$ iff $2^{k-1} < deg(g_{i,j}) \leq 2^k$. The sets $\{s_k\}$ are ordered according to their index, in increasing order. We then begin computing *lcm*s of two polynomials taken from the first set. If this set has only one polynomial we take the second polynomial from the next set. The resulting *lcm* polynomial is placed in the set corresponding to its degree. This process ends when we are left with one polynomial, representing the *lcm* of all the polynomials $g_{i,j}$.

In the *gcd method* the polynomials $g_{i,j}$ are sorted by their degrees. In each iteration the polynomial with the highest degree is taken out of the set and and all pairwise *gcd*s between itself and the other polynomials are taken. If the *gcd* is greater than 1, the other polynomial is divided by this *gcd.* At the end of the iteration none of the remaining polynomials in the set has a factor in common with the polynomial that was taken out. Thus, when the procedure ends, no factor appears in more than one of the $g_{i,j}$s.

Lemma 14:

1. The complexity of the second phase is $O(|H|^2 M(n) \log n)$.

2. The expected complexity of the second phase is $O(|H| \log^3 |H| L(n) \log(|H|n))$.

Proof:

1. We can bound the work required for the *lcm method* as follows. First assume $|H|$ and $d_{i,j}$ are powers of 2 (if they are not, for bounding purposes increase them to the nearest power of 2). Also, assume the polynomials are leaves of a binary tree. All the polynomials in the

same level have the same degree (each level corresponds to a different set s_k). Assume that in every *lcm* step, the degree of the *lcm* is the sum of the degrees of its two operands (i.e. the operands are relatively prime). The maximum degree the final *lcm* can have is $|H|n$ and computing this *lcm* costs $O(M(|H|n)\log(|H|n))$. Computing the two *lcm*'s of the next to last level costs at most $O(2\cdot M(|H|n/2)\cdot\log(|H|n/2))$. In each lower level there are at most twice as many *lcm*'s being computed but each costs less than half the cost of the level above it, hence the total cost is bounded by $O(\log(|H|n)M(|H|n)\log(|H|n)) \leq O(u^2M(|H|n))$.

To use the *lcm* method we need enough memory to store the final *lcm*. If we do not have the required memory, we use the *gcd method*. The work required is $O(|H|^2M(n)\log n)$.

2. When taking into account the expected size of the polynomials $g_{i,j}$, factorization becomes practical. The factoring algorithm used is that of Cantor and Zassenhaus [4]. The complexity for factoring a product of r distinct irreducible polynomials of the degree j is given by $O(rM(rj)(j+\log(rj))$. By Lemma 8, the expected number of polynomials $g_{i,j}$ that have more than $5\cdot 10^k$ factors is less than $|H|/10^{2k+2}$. If we take the number of polynomials with $5\cdot 10^k$ factors to be $\frac{99|H|}{10^{2k+2}}$ (i.e. all polynomials with at most 5 factors are assumed to have 5, all polynomials with $6\ldots 50$ factors are assumed to have 50, etc.), then the expected work required to factor all the polynomials is bounded by

$$O\left(\sum_{k=0}^{\frac{1}{2}\log_{10}|H|-1}\frac{99|H|}{10^{2k+2}}5\cdot 10^k M(5j\cdot 10^k)(j+\log(5j\cdot 10^k))\right).$$

By using the fact that $5j\cdot 10^k < n$ and by writing $M(5j\cdot 10^k)$ as $5j\cdot 10^k\cdot L(n)$, we can bound the sum by

$$O\left(\sum_{k=0}^{\frac{1}{2}\log_{10}|H|-1} 25|H|jL(n)(j+\log n)\right) = O(|H|\log|H|\cdot j\cdot L(n)(j+\log n)). \qquad (3)$$

When the factorization is completed, all the irreducible factors can be sorted in time $O(|H|\cdot\log|H|)$ and the unique factors can be counted.

Summing over $j = 1\ldots u(= es(H))$ we get $O(|H|\log|H|u^2L(n)(u+\log n))$. Since $u\approx\log|H|$, the expression becomes $O(|H|\log^3|H|L(n)\log(|H|n))$.

□

4.3 Finding a non-factor

We are now at the third phase, where we know the smallest degree d_a for which there exists a non-factor for h. We also have, $m \leq |H|$ polynomials g_{i,d_a} that are products of distinct irreducible (primitive) factors of h, all g_{i,d_a}'s are pairwise relatively prime and every irreducible (primitive) factor of degree d_a of h is a factor of one of these polynomials. We want to find an irreducible (primitive) polynomial of degree d_a that is a non-factor of H.

One approach is to divide the product of all irreducible (primitive) polynomials of degree d_a by the product of all m polynomials and find a factor of the result. This might pose a problem if we do not have the product at hand, i.e. only the polynomials g_{i,d_a}, or if the product is too large to handle as one polynomial.

Another way is to randomly select irreducible (primitive) polynomials and check whether they are factors or non-factors. The only way to check is by doing the actual division. This division, however, will be regular long division, and not FFT division, whenever the divisor has very small degree compared to the degree of the dividend. If an irreducible (primitive) polynomial is relatively prime to all of the g_{i,d_a}'s, it is a non-factor. If it divides at least one of the polynomials, we can keep the result of the division and reduce our work in upcoming trials. This reduction requires that polynomials do not repeat in the selection process.

Lemma 15:

1. The complexity of finding a non-factor once d_a is known is $O(|H|^2 n^2 d_a^2 M(d_a))$ for the irreducible case and $O(|H|^2 n^2 d_a^3 M(d_a) \log\log d_a)$ for the primitive case.

2. The expected complexity is $O(|H| \log |H| d_a M(d_a))$ for the irreducible case and $O(|H| \cdot \log |H| \cdot d_a^2 \log\log d_a M(d_a))$ for the primitive case.

Proof:

1. The procedure generates random polynomials, checks them for irreducibility (primitivity) and whether they are factors or not. The expected number of random polynomials that are tested for irreducibility (primitivity) before an irreducible (primitive) polynomial of degree d_a is found is $d_a/2$ ($\frac{d_a}{2} \log\log d_a$) [15]. The work required to test each polynomial

for irreducibility is $O(d_a M(d_a))$ ($O(d_a^2 M(d_a))$) [15]. The sum of the $d_{i,j}$'s cannot exceed $|H|n$, therefore after at most $\frac{|H|n}{d_a}$ irreducible polynomials are tried, a non-factor is found. The work involved with each try is $|H|n \cdot d_a$ (long division). Thus, the expected work required to find a non-factor is $O(|H|^2 n^2 \cdot d_a^2 M(d_a))$. For the primitive case the work is $O(|H|^2 n^2 d_a^3 M(d_a) \log\log d_a)$.

2. If the polynomials $g_{i,j}$ were factored (see proof of Lemma 14,(2)), once d_a is known, we draw irreducible (primitive) polynomials until a non-factor is found. We expect no more than $|H|/d_a$ factors. When an irreducible (primitive) polynomial is drawn, it takes $O(\log |H|)$ to check whether it is a factor or not. Hence, the expected work required to find a non-factor, once d_a is known, is bounded by $O(|H| \log |H| d_a M(d_a))$ for the irreducible case and $O(|H| \log |H| d_a^2 M(d_a) \cdot \log\log d_a)$ for the primitive case.

□

5 Practical scenarios

In this section we discuss some practical scenarios for finding zero-aliasing polynomials. First, when we want a non-factor of a pre-specified degree. Second, when we want to find a non-factor fast. Third, we compare our algorithm for finding a least degree non-factor with an exhaustive search over all irreducible (primitive) polynomials in ascending degrees. In some cases, this type of search will be faster.

5.1 Finding a non-factor of a pre-specified degree

In cases where the register is required to function as both a RA and a PG, a non-factor of a pre-specified degree is needed. Thus

Problem 3: Given a set of polynomials $H = \{h_1, h_2, \ldots, h_{|H|}\}$, with $deg(h_i) \leq n$, find an irreducible (primitive) non-factor of degree t for H. □

This problem is exactly the same as finding the least degree non-factor, except that we only need to consider the case of $j = t$, instead of iterating over all $1 \leq j \leq u$. We first compute the polynomials $g_{i,t}$, then determine whether a non-factor of degree t exists, and if so find one.

Lemma 16:

1. The complexity of finding a non-factor of degree t is $O(|H|^2 n^2 t^2 M(t))$ for the irreducible case and $O(|H|^2 n^2 t^3 M(t) \log\log t)$ for the primitive case.

2. The expected complexity is $O(|H| M(n)(t + \log n))$.

Proof:

1. Computing the polynomials $g_{i,t}$ involves computing $g_{i,t} = gcd(h_i, x^{2^t} - x)$ and for each $l \in md(t)$ computing $f_l = gcd(g_{i,t}, x^{2^l} - x)$ and $g_{i,t} = g_{i,t}/f_l$. The cost of the first *gcd* computation is $O(tM(d_i) + M(d_i)\log d_i)$. The cost of the $|md(t)|$ subsequent *gcd* and divisions is bounded by $O(\log t(tM(d_{i,t}) + M(d_{i,t})\log d_{i,t}))$. Substituting n for d_i and $d_{i,t}$ we get $O(\log t \cdot M(n)(t + \log n))$.

   Once we have the polynomials $g_{i,t}$, we need to sift out multiple instances of the same irreducible polynomial. When using the *gcd method*, this has a worst case cost of $O(|H|^2 M(n) \cdot \log n)$.

   At this stage, we know whether a non-factor of degree t exists or not. If one exists, we carry out phase 3. This has a worst case complexity of $O(|H|^2 n^2 t^2 M(t))$. This is the dominant term for the whole process. The analysis is the same for the primitive case, hence the worst case complexity of finding an irreducible (primitive) non-factor of a given degree t for a set of polynomials H is $O(|H|^2 n^2 t^2 M(t))$ $(O(|H|^2 n^2 t^3 M(t) \log\log t))$.

2. We turn to analyze the expected complexity. For each h_i, we compute $g_{i,t} = gcd(h_i, x^{2^t} - x)$. This costs $O(|H| M(n)(t + \log n))$. The cost of sifting out the factors of degree less than t from the $g_{i,t}$'s, based on the expected number of factors for each degree, will be insignificant. Factoring and sorting the polynomials in the second phase has expected cost of $O(|H| \log |H| t \cdot L(n)(t + \log n))$ (Eq. (3)). The expected number of distinct irreducible factors of degree t of H is bounded by $|H|/t$. Thus, the cost of finding a non-factor at this stage which consists of drawing at most $\frac{|H|}{t}$ irreducible (primitive) polynomials, each at an expected cost of $O(\frac{t}{2} tM(t))$ $(O(\frac{t}{2} \log\log t \cdot t^2 M(t)))$, and checking it against the list of factors, is bounded by $O(\frac{|H|}{t} \frac{t}{2} tM(t) \log(|H|/t))$ for the irreducible case and $O(|H|(\log |H| - \log t) t^2 M(t) \log\log t)$ for the primitive case. Hence, the expected complexity of finding a non-factor of degree t for H is bounded by $O(|H| M(n)(t + \log n))$.

□

5.2 Finding a non-factor fast

Problem 4: Given a set of polynomials $H = \{h_1, h_2, \ldots, h_{|H|}\}$, with $deg(h_i) = d_i \leq n$, $\sum_{i=1}^{|H|} d_i = d_h$, find an irreducible (primitive) non-factor of H in less than 2^c tries. □

The sum of the degrees of all irreducible (primitive) polynomials of degree less than or equal to $s(d_h)$ $(p(d_h))$ is greater than d_h. If we look at $u = s\left(\frac{2^c}{2^c-1}d_h\right)$ $\left(u = p\left(\frac{2^c}{2^c-1}d_h\right)\right)$ then $\frac{\sum_{j=1}^{u} jI_2(j)-d_h}{\sum_{j=1}^{u} jI_2(j)} \geq 2^{-c}$ $\left(\frac{\sum_{j=1}^{u} \phi(2^j-1)-d_h}{\sum_{j=1}^{u} \phi(2^j-1)} \geq 2^{-c}\right)$ and if we draw uniformly from all irreducible (primitive) polynomials of degree u, after 2^c drawings we expect to find a non-factor. The expected work cost for this case is $O(2^c \cdot (u^2M(u) + u|H|n)) = O(2^c u|H|n)$ which is the cost of 2^c iterations of drawing a polynomial and testing for irreducibility, and once one is found dividing all $|H|$ polynomials by this candidate non-factor, using long division. For the primitive case this becomes $O(2^c \cdot (u^3M(u)\log\log u + u|H|n)) = O(2^c u|H|n)$.

Example 3: Using the numbers in Example 1 again, say we want to find a non-factor in no more than 8 tries. We compute the bound $p\left(\frac{8}{7}10^{10}\right)$ and draw from all the primitive polynomials up to the computed bound. If we use Table 1, we see that instead of looking at the polynomials of degree less than or equal to 33, we need to consider all primitive polynomials of degree up to 34. In general, $\frac{2^c}{2^c-1} \leq 2$, hence by Lemma 5, we only have to consider polynomials of degree greater by at most 2 than for the case when we want the minimum degree non-factor. □

We can also use the expected bounds $es(d)$ and $ep(d)$ to lower the degrees of the candidate non-factors.

5.3 Exhaustive search

In this subsection, we compare our algorithms with an exhaustive search for a least degree non-factor. We will look at the irreducible case.

Assume the least degree irreducible non-factor has degree d_a. Also, assume we have a list of all irreducible polynomials in ascending order. The number of irreducible roots of degree j is

less than 2^j. We can bound the work required to find the non-factor, by an exhaustive search, by $O(|H|n2^{d_a+1})$. Using the expected bound on d_a ($d_a = O(\log |H|)$), we can bound the work by $O(|H|^2 n)$. The expected work required to find the least degree non-factor, by our algorithms, is $O(|H|u^2 M(n))$, which becomes $O(|H| \log^2 |H| \cdot n \log n)$ when we substitute in the value of u. Not taking into account any of the constants involved with these two results, their ratio is

$$\frac{|H|}{\log^2 |H| \log n}.$$

Assuming $|H| = 1024$, this ratio is less than 1 for $n > 1210$. Assuming $|H| = 2048$, the ratio is less than 1 for $n > 124,500$. For $|H| = 4096$, the ratio is less than 1 for $n > 365,284,284$. This suggests that depending on the number of target faults and the length of the test sequence, an exhaustive search might be more effective. Assuming the number of faults is less than 4096, based on the expected bound on the degree of a non-factor, we would need to store all the irreducible polynomials of degree at most 13. The number of these polynomials is 1389.

6 Experimental results

The following experiments were conducted to verify our results. The experiments were conducted on a HP-700 workstation.

6.1 Random selections based on the absolute bounds

An experiment was set up as follows. We generated a set of 1000 random polynomials of degree at most 200,000. This corresponds to a CUT with 1000 faults, i.e. $|H| = 1000$, and a test length of 200,000, i.e. $n = 200,000$. The degree of the product of these polynomials (d_h) was less than or equal to 200,000,000. We wanted a probability greater than 1/2 of finding a non-factor with just one drawing of a primitive polynomial. By looking at Table 1, we can achieve this by selecting from the set of all primitive polynomials of degree less than or equal to 29. The polynomials were drawn in a 2 step process. The first step selected the degree of the primitive candidate, the second selected the candidate. In the first step we selected a 32 bit number and took its value modulo the number of primitive roots in the fields $GF[2]$ through $GF[2^{29}]$. The result was used to determine the degree of the primitive candidate, by looking at the first field $GF[2^d]$ such that the number of primitive roots in the fields $GF[2]$ through $GF[2^d]$ is greater

than the result. The selection of the actual polynomial was done by setting the coefficients of $x, x^2, \ldots, x^{d-1}$ by a LFSR with a primitive feedback polynomial of degree $d-1$ that was initialized to a random state. This guarantees that no candidate will be selected twice and all candidates will have a chance at being considered. The candidates were tested for primitivity and if they were primitive, they were tested for being non-factors. If at some point they were found to be factors, the search continued from the current state of the degree $d-1$ LFSR.

We ran 200 such experiments. In *all 200 experiments* the first primitive candidate turned out to be a non-factor. Of the non-factors that were found, 1 was of degree 21, 2 were of degree 22, 3 of degree 23, 2 of degree 24, 7 of degree 25, 13 of degree 26, 32 of degree 27, 35 of degree 28 and 105 were of degree 29.

The number of polynomials that were tested for primitivity before one was found ranged from 1 to 160. The average number was 16. The time it took to find a primitive polynomial ranged from 0.01 seconds to 0.79 seconds. The average time was 0.104 seconds. It took between 153.25 and 166.68 seconds to find a non-factor, with the average being 160.50 seconds.

These experiments show that given the error sequences for each of the faults of interest, it is *very easy* to find a zero-aliasing polynomial for a circuit.

6.2 Random selections based on the expected bounds

Based on our expected bounds, Corollary 10, we should be able to find a non-factor of degree at most 14. We ran 100 experiment as above, only this time, we selected only primitive polynomials of degree 11 (the expected bound based on Table 1). The first primitive candidate that was selected was a non-factor in 66 of the 100 experiments. 19 experiments found the non-factor with the second candidate, 11 with the third, 2 with the fourth, 1 with the fifth and 1 with the sixth. We ran 100 experiments selecting only primitive candidates of degree 9. The number of primitive candidates that were tried before a non-factor was found ranged from 1 to 28. The average number of candidates was 7.5.

To test the tightness of our expected bound, we ran 126 experiments. In which 1024 random polynomials of degree at most 200,000 were generated and an exhaustive search, in increasing order of degrees, was conducted to find the least degree non-factor. By our expected bound, this least degree should be less than 14. In one experiment, the least degree was 7. In 35 it was 8 and in the remaining 90 experiments, the least degree was 9.

From these experiments we conclude that when the error polynomials are in fact random polynomials, the expected bounds, based on the analysis of the expected number of factors of a certain degree for a random polynomial, are in fact upper bounds on the least degree non-factor for a set of polynomials. As expected, the bounds read from Table 1 are tighter than those from Corollary 10.

6.3 Experiments on benchmark circuits

We tried our worst case and expected bounds on error sequences of two circuits from the Berkeley synthesis benchmarks [2]. The first circuit was *in5*, the second was *in7*. We used a fault simulator that did not take into account any fault collapsing, hence the number of faults was twice the number of lines in the circuit (for *stuck-at-0* and *stuck-at-1* faults on each line).

For circuit *in5* there were 1092 faults, six of which were redundant, hence there were 1086 detectable faults. The circuit had 14 primary outputs. We used a test sequence of length 6530 that detects all the non-redundant faults and computed the *effective output polynomials* of all the faults. All were non-zero, hence there were no cancellation of errors from one output by errors of another output. Thus we had 1086 error polynomials of degree at most 6543. From Table 1, the worst case bound on the degree of a primitive non-factor is 23. To draw a primitive non-factor with probability greater than $\frac{1}{2}$ we need to consider all primitive polynomials of degree 24 or less. We conducted 20 experiments of drawing zero-aliasing primitive polynomials, based on our worst case bounds. In all experiments, the first candidate was a non-factor. We then conducted another 20 experiments, this time drawing primitive polynomials of degree 14, the size of the register available at the circuit outputs. In all experiments the first candidate was a non-factor. Based on our expected bounds (Table 1), we should find a non-factor of degree 11 or less. We tried finding non-factor of degree 11, 9 and 7. For the degree 11 experiments, in 17 of 20 cases, the first primitive candidate was a non-factor. Two experiments found the non-factor with the second try, one with the third. We conducted 15 degree 9 experiments before considering all 48 primitive polynomials of degree 9. Of the 48 primitive polynomials of degree 9, 33 were factors, and 15 were non-factors. The average number of candidates tried before a non-factor was found was $3\frac{1}{3}$. All 18 primitive polynomials of degree 7 were factors.

For circuit *in7* there were 568 faults, 567 of which were non-redundant. The circuit has 10 primary outputs and we used a test sequence of length 9280. Using the worst case bounds,

to ensure selection of a primitive non-factor with probability greater than $\frac{1}{2}$, we considered all primitive polynomials of degree 24 or less. All 20 experiments found a non-factor with the first candidate. The expected bound (Table 1) for the degree of a primitive non-factor was 10. We tried to find non-factors of degree 11 and 10 (the size of the register available at the outputs). All 20 degree 11 experiments found a non-factor with the first try. Of the 20 degree 10 experiments, 13 found a non-factor with the first try, 6 with the second and one with the third.

For both circuits we tried to find the least degree non-factor using an exhaustive search. Since the fault extractor we used did not do any fault collapsing, some of the error polynomials were identical. By summing the values of all non-zero erroneous output words for each simulated fault, we found at least 292 different error polynomials in *in7* and at least 566 different error polynomials in *in5*. This would make our expected bounds (Table 1) to be 9 for *in7* and 10 for *in5*. For both circuits the least degree non-factor had degree 8. It took 11 CPU minutes to find each of these polynomials.

The experiments on the two benchmark circuits show that the assumption that the error polynomials behave as random polynomials does not invalidate our analysis and results. The expected bounds, as was the case for the random experiments, were upper bounds on the least degree non-factor.

7 Conclusions

In this paper we presented procedures for selecting zero-aliasing feedback polynomials for MISR-based RAs. When both PGs and RAs are designed as LFSRs/MISRs, our scheme, combined with algorithms for selecting efficient feedback polynomials for pattern generation [11], enables the selection of one feedback polynomial that serves both tasks, thus reducing the overhead of reconfigurable registers.

We presented upper bounds on the least degree irreducible and primitive zero-aliasing polynomial for a set of modeled faults. We showed that in all practical test applications such a polynomial will always be of degree less than 53. In fact, by our expected bounds, when the number of faults is less than 10^6, this degree will be at most 21. In the experiments that were conducted, a zero-aliasing polynomial of degree less than the expected bound was always found.

We also presented procedures for finding a zero-aliasing polynomial, when the objective is to

worst case		
	irreducible	primitive
bounds (u)	$s(d_h) = \lceil \log(\lvert H \rvert n + 1) \rceil$	$p(d_h) = s(d_h) + \lceil 1 + \log\log(2s(d_h)) \rceil$
smallest non-factor	$\lvert H \rvert^2 n^2 u^2 M(u)$	$\lvert H \rvert^2 n^2 u^3 M(u) \log\log u$
degree t non-factor	$\lvert H \rvert^2 n^2 t^2 M(t)$	$\lvert H \rvert^2 n^2 t^3 M(t) \log\log t$
expected		
	irreducible	primitive
bounds (u)	$es(H) = \lceil \log \lvert H \rvert \rceil$	$ep(H) = 1 + es(H) + \lceil \log\log(2es(H)) \rceil$
smallest non-factor	$\lvert H \rvert M(n) u^2$	$\lvert H \rvert M(n) u^2$
degree t non-factor	$\lvert H \rvert M(n)(t + \log n)$	$\lvert H \rvert M(n)(t + \log n)$

Table 3: Summary of Results

minimize the degree, to have a specific degree or speed. We analyzed the computational effort that is required both under worst case conditions and expected conditions. A (partial) summary of the results is presented in Table 3. For both the worst case analysis and expected analysis, Table 3 shows the upper bounds on the smallest non-factor, the computational complexity of finding a smallest non-factor and the complexity of finding a factor of a given degree.

Based on our analysis and on our experiments, it is our conclusion that when the error polynomials of the modeled target faults are available, zero-aliasing is an *easily* achievable goal. Thus, to ensure high quality tests, a premium should be put on fault modeling, automated test pattern generator design and fault simulation. With these tools available, zero-aliasing is not a problem.

Acknowledgement: We wish to thank Profs. L. A. Adleman, D. J. Ierardi and L. R. Welch for many helpful discussions.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974

[2] R.K. Brayton, G.D. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, MA, 1984

[3] E.R. Berlekamp, *Factoring Polynomials Over Large Finite Fields*, Mathematics of Computation, Vol. 24, No. 111, pp. 713-735, July, 1970

[4] D.G. Cantor and H. Zassenhaus, *A New Algorithm for Factoring Polynomials over Finite Fields*, Mathematics of Computation, Vol. 36, No. 154, pp. 587-592, April, 1981

[5] K. Chakrabarty and J.P. Hayes, *Aliasing-Free Error Detection (ALFRED)*, Proc. 11th IEEE VLSI Test Sym., pp. 260-266, 1993

[6] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, McGraw-Hill, 1990

[7] S.K. Gupta, D.K. Pradhan and S.M. Reddy, *Zero Aliasing Compression*, Proc. 20th Int. Sym. on Fault-Tolerant Computing, pp. 254-263, 1990

[8] R.L. Graham, D.E. Knuth and O. Patashnik, *Concrete Mathematics*, Addison-Wesley Publishing Company, 1989

[9] D.R. Heath-Brown, *Zero-free regions for Dirichlet L-functions, and the least prime in an arithmetic progression*, Proc. London Math. Soc., Vol. 64, No. 3, pp. 265-338, 1992

[10] A. Lempel, G. Seroussi and S. Winograd, *Complexity of multiplication in finite fields*, Theoretical Computer Science, Vol. 22, pp. 285-296, 1983

[11] M. Lempel, S.K. Gupta and M.A. Breuer, *Test Embedding with Discrete Logarithms*, Proc. 12th IEEE VLSI Test Sym., pp. 74-80, 1994

[12] R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, 1986

[13] I. Pomeranz, S.M. Reddy and R. Tangirala, *On Achieving Zero Aliasing for Modeled Faults*, Proc. European Design Automation Conf., 1992

[14] D.K. Pradhan and S.K. Gupta, *A New Framework for Designing and Analyzing BIST Techniques and Zero Aliasing Compression*, IEEE Transactions on Computers, Vol. C-40, No. 6, June 1991.

[15] M.O. Rabin, *Probabilistic Algorithms in Finite Fields*, SIAM J. of Computing, Vol. 9, No. 2, pp. 273-280, May 1980

[16] P. Ribenboim, *The Book of Prime Number Records*, 2nd Edition, Springer-Verlag, 1989

[17] A. Schönhage, *Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2*, Acta Informatica, Vol. 7, 1977, pp. 395-398