Software Accelerated Functional Fault Simulation for Data-Path Architectures*

M. Kassab, N. Mukherjee, J. Rajski†, and J. Tyszer

Microelectronics and Computer Systems Laboratory
McGill University, Montreal, Canada, H3A 2A7

†Mentor Graphics Corporation, Wilsonville, OR 97070, USA

Abstract - This paper demonstrates how fault simulation of building blocks found in data-path architectures can be performed extremely efficiently and accurately by taking advantage of their simple functional models and structural regularity. This technique can be used to accelerate the simulation of those blocks in virtually any fault simulation environment, resulting in fault simulation algorithms that can perform fault grading in a very demanding BIST environment.

I. Introduction

Data-path architectures constitute a large portion of circuits manufactured by the ASIC industry, and are mainly used in high performance computing systems like DSP circuits. The proliferation of embedded systems and high-level synthesis is expected to further increase the number of circuits comprising data-paths with such regular blocks as adders, multipliers, multiplexers, shifters, register files, etc.

Recently, it has been shown that for circuits with data-path architectures, existing hardware on the chip, such as arithmetic and logic units (ALUs), can be used to successfully perform test pattern generation [1] and test response compaction [2]. Consequently, for a given circuit, a built-in self test (BIST) scheme can be devised such that the circuit tests itself with virtually no area overhead and no performance degradation [3]. The test is applied at-speed, which allows the application of a large number of cycles and increases the probability of detecting dynamic and unmodeled faults.

However, in order to assess the quality of a proposed BIST scheme, fault grading has to be used. This requires fault simulation to be performed for a relatively large number of vectors with no fault dropping, which can be very computationally intensive for most known fault simulation techniques.

Existing fault simulators fall into several categories. Most are gate-level simulators, such as PROOFS [4] and HOPE [5], with very efficient structural simulation algorithms and the flexibility of handling any circuit whose structural model is known. However, they do not exploit the functionality of the circuit and its building blocks to reduce simulation time; any circuit is treated like random logic. Gate-level simulators require the entire circuit to be modeled at the structural level. This precludes simulating circuits which contains blocks that are only modeled behaviorally. Some simulators [6] model faults functionally at a higher level of abstraction. This enhances performance at the expense of accuracy. Simulators like MOZART [7] allow multilevel simulation, such that different blocks have the flexibility of being modeled at different levels of abstraction. Blocks which are not to be fault-simulated, or do not have a gate-level representation, can thus be modeled at a higher level of abstraction. More recent developments, like FEHSIM [8], use enhanced scheduling techniques to dynamically switch between different levels of abstraction such as to maximize speed without losing accuracy.

The simulation approach presented in this paper exploits the fact that most modules in data-path architectures perform arithmetic operations for which fault-free simulation can be performed very efficiently. The regularity and functionality of many arithmetic structures makes it possible to compute the faulty output functionally, without resorting to structural simulation. This is done without loss of accuracy for any fault model. Hence, behavioral-level speed can be obtained with gate-level accuracy. Memory usage is also drastically reduced as no netlist has to be instantiated for the module, and no values internal to the netlist need to be stored for the different faulty machines.

II. Functional Fault Modeling In Regular Blocks

To analyze the fault coverage of a circuit, given a BIST scheme, fault simulation has to be performed without fault dropping [9], i.e., the entire fault list has to be simulated for all vectors, making the process very computationally intensive. The techniques presented in this paper exploit features of data-path building blocks to speed up their simulation, and hence make fault simulation of data-path circuits possible in a BIST environment. Data-path architectures mainly consist of building blocks, such as adders, subtractors, multipliers, comparators, etc. These blocks have regular structures and simple functionality. Hence, their faulty behavior can often be modeled and computed functionally with gate-level accuracy, as will be shown in this section. Fault simulation for blocks with functional fault models can thus be performed almost as fast as functional simulation of the fault-free model.

This section examines the modeling of some of those building blocks. The fault-free functionality can be represented by simple operations, which can be invoked for faults external to the module. For faults internal to the module, the faulty output of the module can be efficiently computed by superposing the effect of the fault on the fault-free output.

* This work was supported by a Cooperative Research and Development grant from the Natural Sciences and Engineering Research Council of Canada and Bell-Northern Research.

32nd ACM/IEEE Design Automation Conference ®

Two examples are covered in detail: a ripple-carry adder and an array multiplier for unsigned numbers. In a similar way, fault models were developed for a number of other building blocks, such as Booth multipliers, ALUs, and multiplexers.

A. Adders, Subtractors, and Comparators

The modeling of an adder is illustrated using a ripple-carry adder (Figure 1). Each bit-slice is implemented as a full-adder cell. This simulation technique can be applied to a variety of fault models. The single-stuck-at model will be used in this paper. The uncollapsed stuck-at fault set for the full-adder cell consists of 30 faults. Hence, an *n*-bit adder contains $30n$ faults (uncollapsed fault set).

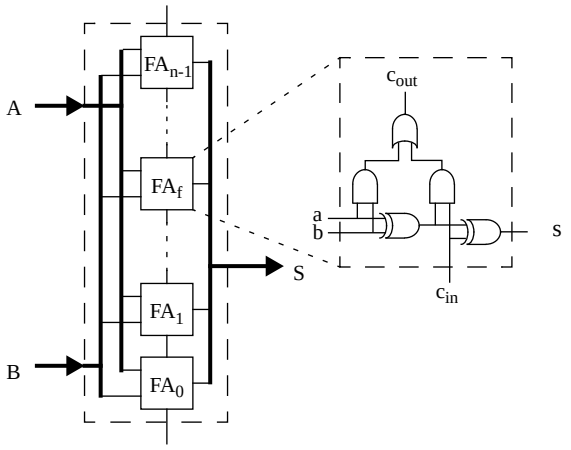

Fig. 1: Ripple-carry adder.

In the functional fault model, the faulty output due to an internal fault will be computed by superposing the fault effect on the fault-free sum. The faulty behavior of a full-adder cell is known. A lookup table is used to model the behavior of the cell by storing the outputs (sum and carry-out) of the full-adder cell for all faults (30) and all possible inputs ($2^3 = 8$). The table in this example models single stuck-at faults.

Consider the n-bit adder, with the fault-free sum output *S*, and an internal fault located in bit-slice f. Let s and c be the fault-free sum and carry-out values of bit-slice f, respectively, while s_f and c_f denote the faulty values of s and c. The faulty output of the adder can be computed according to the following theorem:

Theorem 1: The output of the faulty adder is $S + (s_f - s) \cdot 2^f + (c_f - c) \cdot 2^{f+1}$.

Proof: To superpose the effect of the fault on the output of the bit-slice, bit f of *S* has to change from to s_f. This is equivalent to adding $(s_f - s)$ to bit f, or adding $(s_f - s) \cdot 2^f$ to S. The carry-out from bit f is an input to the adder formed by bits $f+1$ to n-1. The difference in the output of the adder formed by bits $f+1$ to $n-1$, for the faulty adders, is equal to $(c_f - c)$. Hence, the fault effect is realized by adding $(c_f - c)$ to bit $f+1$, or adding $(c_f - c) \cdot 2^{f+1}$ to S. The two effects are superposed, such that $(s_f - s) \cdot 2^f + (c_f - c) \cdot 2^{f+1}$ is added to S. ■

Based on Theorem 1, calculation of the output of the faulty adder can be conducted as follows. First, variables s, c, s_f, and c_f have to be determined. To look up these values in the corresponding table, the three inputs to faulty cell f are calculated. The two input bits to the cell, A_f, and B_f, are directly extracted from primary inputs A and B, respectively. The carry-in bit to the cell is equal to the carry-out of the sum of bit-range 0 to $f-1$. This is illustrated in Algorithm 1.

```
adder(fault, x, y, c_in)
  case (fault location) of
    external to module:
      return (x + y + c_in)
    internal to module:
      f = index of faulty full-adder
      a = x[f]
      b = y[f]
      if (fault in least significant bit-slice)
        c = c_in
      else
        c = carry(x[f-1, 0] + y[f-1, 0] + c_in)

      fault-free sum of f = a ⊕ b ⊕ c
      fault-free carry of f = (a ∧ b) ∨ (b ∧ c) ∨ (a ∧ c)

      faulty sum of f = table_sum(fault,a,b,c)
      faulty carry of f = table_carry(fault,a,b,c)

      d_sum = (faulty sum of f) - (fault-free sum of f)
      d_carry = (faulty carry of f) - (fault-free carry of f)

      fault-free output = x + y + c_in
      correction = (d_sum · 2^f) + (d_carry · 2^(f+1))
      faulty output = fault-free output + correction

      return(faulty output)
```

Algorithm 1: Binary adder Functional fault model

The functional model shown by the algorithm computes the faulty output in constant time, independent of the adder size. The use of table lookup for the full-adder cells allows fast evaluation, as well as the flexibility to use different fault models.

Subtractors and comparators can be modeled as extensions of the adder. By inverting one of the adder's inputs and feeding a 1 to the carry-in of the least significant bit, the adder is transformed into a 2's complement subtractor. The functional fault model then needs to distinguish whether the fault lies on the adder or the input inverters, and inject the fault accordingly. The comparator, with inputs A and B, is required to check if $A > B$. This can be realized by feeding A and B to the negative and positive inputs of the subtractor, respectively. The result bit is the carry-out of the adder. The result bit is a 0 when $A \leq B$, and a 1 when $A > B$.

B. Multipliers

The modeling of a multiplier is illustrated by an array multiplier for unsigned numbers [10] (Figure 2). The multiplier uses an array of carry-save adders to add the partial products. The structure consists of an array of full-adders. The multiplier accepts an m-bit input x and an n-bit input y. The implementation shown contains $m(n-1)$ full-adders. Hence, the uncollapsed fault set consists of $30m(n-1)$ stuck-at faults.

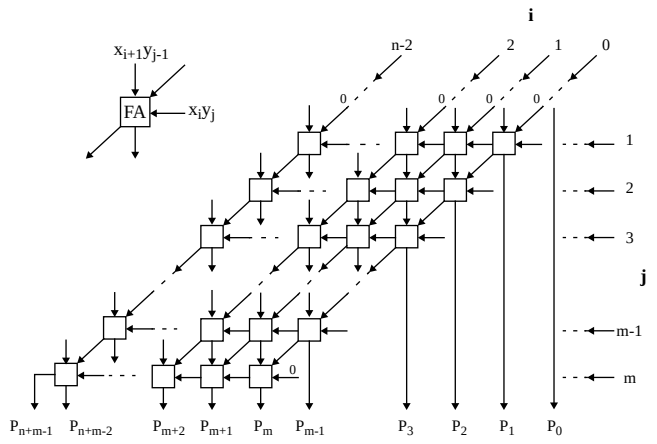

Fig. 2: Array multiplier

The fault-free model is a multiplication operation, which is invoked for faults external to the multiplier. For internal faults, the proposed fault model computes the faulty output of the multiplier in the following 3 steps:

1. Determination of the coordinates of the faulty cell.
2. Computation of the three inputs of the faulty cell.
3. Computation of the output of the faulty multiplier.

As with the adder, the faulty output is determined by superposing the fault effect on the correct multiplication result. Given the cell in which the fault is located, the faulty and fault-free outputs of the cell need to be determined before the superposition can be performed. However, for the outputs of the cell to be extracted from the full-adder lookup table, the inputs of the faulty cell first have to be determined.

Let $C_{i,j}$ denote the full-adder cell with coordinates i and j, where i and j are the column and row numbers, respectively. The coordinates of the faulty cell can be extracted from the fault identifier.

Inputs of faulty cell $C_{i,j}$

Each cell $C_{i,j}$ has two inputs $a_{i,j}$ and $b_{i,j}$, a carry-in input $c_{i,j}$, a sum output $s_{i,j}$, and a carry-out output $e_{i,j}$. Let the sum and carry-out outputs of $C_{i,j}$ for a fault f located in the cell be $s_{i,j,f}$ and $e_{i,j,f}$ respectively. We will also denote the result of the multiplication by p for the fault-free multiplier, and by p_f for the multiplier with fault f. Note that the structure of the circuit is very regular except for the last row, where the carry ripples horizontally. Hence, the analysis to determine the three inputs of $C_{i,j}$ is divided into two parts: the first is used for any cell located in any row but the last one, while the second is used for cells in the last row.

First, consider cells in any row except the last. The inputs x and y have masks applied to them, such that the value of the desired line can be observed directly on the output of the multiplier with the masked inputs. Hence, the appropriate masks are applied to the inputs, functional multiplication is performed, and the specific bit are extracted from the output.

Let $x[i]$ represent the value of the i'th bit of x. Also let x_m be the masked value of input x and y_m the masked value of input y, such that:

$$x_m[p] = \begin{cases} x[p]; & i+1 \le p \le m-1 \\ 0; & 0 \le p \le i \end{cases} \quad (1)$$

$$y_m[q] = \begin{cases} y[q]; & 0 \le q \le j-1 \\ 0; & q \ge j \end{cases} \quad (2)$$

The product of the masked inputs is $p_m = x_m \cdot y_m$. Bit $(i+j)$ of p_m, contains the first input $a_{i,j}$ to $C_{i,j}$. Hence, $a_{i,j} = p_m[i+j]$.

Theorem 2: The input $a_{i,j}$ to cell $C_{i,j}$ is the $(i+j)^{th}$ bit of the product of x_m and y_m, where x_m and y_m are defined by Equations 1 and 2, respectively.

Proof: First, we will prove that the value of $a_{i,j}$ is the same for inputs x_m and y_m as it is for the original inputs x and y. From the structure of the multiplier, it can be seen that $a_{i,j}$ is not affected by $\{x(q), q \le i\}$ or by $\{y(r), r \ge j\}$. Hence, the bits which are masked in x_m and y_m are not used in computing $a_{i,j}$. Now we have to show that $a_{i,j}$ is observed on bit $(i+j)$ of $x_m \cdot y_m$. $a_{i,j}$ propagates to the $(i+j)^{th}$ bit of p_m through a number of full-adder cells. The other two inputs of each of these full-adders is reduced to zero by the masks applied to x and y. Hence, the full-adders between $a_{i,j}$ and the primary output become transparent, and the value propagates to the output of the multiplier unchanged. ■

The output $s_{i,j}$ of $C_{i,j}$ can be obtained in a similar way by computing the input $a_{i,j+1}$ to $C_{i,j+1}$. The carry-in signal $c_{i,j}$ can then be deduced as follows: $c_{i,j} = s_{i,j} \oplus a_{i,j} \oplus b_{i,j} = a_{i,j+1} \oplus a_{i,j} \oplus b_{i,j}$. where $b_{i,j} = x[i] \wedge y[j]$.

Now consider cells in the last row. The output of cell $C_{i,j}$, where $j = m$, is: $s_{i,j} = p[i+j] = p[i+m]$. Inputs $a_{i,j}$ for any cell in the last row can be computed as previously described for cells in other rows.

To calculate $c_{i,j}$, the carry-out of cell $C_{i,j-1}$ is calculated as presented in the previous case, by computing the three inputs of $C_{i,j-1}$, then calculating the carry-out, where the carry-out is $(a_{i,j-1} \wedge b_{i,j-1}) \vee (a_{i,j-1} \wedge c_{i,j-1}) \vee (b_{i,j-1} \wedge c_{i,j-1})$.

The input $b_{i,j}$ is 0 for the first cell of the last row ($i = 0$), i.e. for $C_{0,m}$, $b_{0,m} = 0$. For any other cell in the last row ($i > 0$), $b_{i,j}$ is $s_{i,j} \oplus a_{i,j} \oplus c_{i,j}$.

Given $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$, the sum and carry-out can be determined from the full-adder output lookup table. Note that the table lookup technique is just one method of determining the cell output. If the cell involved is large, the memory requirements to store its outputs for all internal faults and all input combinations may be impractical. In that case, some other technique, like gate-level simulation of the cell, may be performed to determine the faulty and fault-free cell outputs.

Faulty output

Given the output and carry-out of the faulty cell, the faulty multiplier output is calculated by superposing the fault effect on the fault-free output. Let δ_s and δ_e denote the difference between the faulty and fault-free values of the sum and carry-out outputs of $C_{i,j}$, respectively, i.e.

$$\delta_s = s_{i,j,f} - s_{i,j}$$
$$\delta_e = e_{i,j,f} - e_{i,j}$$

Theorems 3 and show the effect of changes to $s_{i,j}$ and $e_{i,j}$ on the multiplication result, respectively.

Theorem 3: The difference δ_s between the output of the faulty and fault-free multiplier, due to the fault effect on the sum output of $C_{i,j}$, is $\delta_s \cdot 2^{i+j}$.

Proof: From the structure of the multiplier, it follows that the output of cell $C_{i,j}$ (or any input of $C_{i,j}$) is added to the final product, which is essentially a sum of the different rows, at position $(i + j)$. Due to the linearity of the circuit, the change from 0 to 1, or 1 to 0, can be superposed on the product by adding ± 1 at position $(i + j)$. That is, by adding $\pm 2^{i+j}$ to the fault-free result of the multiplication. ■

Theorem 4: The difference δ_e between the output of the faulty and fault-free multiplier circuits, due to the fault effect on the carry-out output of $C_{i,j}$, is $\delta_e \cdot 2^{i+j+1}$.

Proof: The carry-out from $C_{i,j}$ is an input of $C_{i,j+1}$. According to Theorem 3, the effect of the change on the line can be superposed on the product at position $i + (j + 1)$. That is, by adding $\pm 2^{i+j+1}$ to the fault-free multiplication. ■

The effects of changes on both $s_{i,j}$ and $e_{i,j}$, discussed in Theorems 3 and , respectively, can be superposed to compute the product of the faulty multiplier:

$$p_m = p + \delta_s \cdot 2^{i+j} + \delta_e \cdot 2^{i+j+1}$$

The multiplier functional fault model is shown in Algorithm 2 . As with the adder, the evaluation of the multiplier model requires constant time, i.e. the performance of the model is independent of the size of the multiplier. Hence, the efficiency of this simulation technique compared to gate-level simulation increases with larger structures.

multiplier(*fault, x, y*)
  case (*fault* location) **of**
    external to *module*:
      return $(x \cdot y)$
    internal to *module*:
      Determine coordinates i and j of faulty cell $C_{i,j}$
      $p_m = x[m-1, i+1] \cdot y[j-1, 0]$
      if (faulty cell in any row except the last)
        Input a of $C_{i,j} = p_m[i+j]$
        Input b of $C_{i,j} = x[i] \wedge y[i]$
        Output s of $C_{i,j}$ = input of $C_{i,j+1}$
        Input c_{in} of $C_{i,j} = a \oplus b \oplus s$
      else
        Input a of $C_{i,j} = p_m[i+j]$
        Output s of $C_{i,j}$ = Bit $(i+j)$ of $x \cdot y$
        Compute the 3 inputs of $C_{i,j-1}$, as done for $C_{i,j}$
        Output c_{out} of $C_{i,j-1}$ =
            $(a \wedge b) \vee (a \wedge c_{in}) \vee (b \wedge c_{in})$
        Input c_{in} of $C_{i,j}$ = c_{out} of $C_{i,j-1}$
        Input b of $C_{i,j} = a \oplus s \oplus c_{in}$

      f.f. sum of $f = a \oplus b \oplus c_{in}$
      f.f. carry of $f = (a \wedge b) \vee (a \wedge c_{in}) \vee (b \wedge c_{in})$

      faulty sum of f = table_sum(*fault,a,b,c*$_{in}$)
      faulty carry of f = table_carry(*fault,a,b,c*$_{in}$)

      d_{sum} = (faulty sum of $C_{i,j}$) - (f.f. sum of $C_{i,j}$)
      d_{carry} = (faulty carry of $C_{i,j}$) - (f.f. carry of $C_{i,j}$)

      fault-free output = $x \cdot y$
      correction = $(d_{sum} \cdot 2^{i+j}) + (d_{carry} \cdot 2^{i+j+1})$
      faulty output = fault-free output + correction

      return(faulty output)

Algorithm 2: Array multiplier functional fault model

III. Experimental Results

The experimental results presented in this section demonstrate the performance of the proposed fault simulation scheme and its applicability to typical data-path architectures. High-level synthesis benchmark circuits were simulated in a computationally-demanding BIST environment. The experiments are divided into two parts. First, some basic building blocks are analyzed in Section III.A. This involves simulation benchmarks using functional fault models, as well as testability results. In the second part, covered in Section III.B, a number of high-level synthesis benchmark circuits are used to evaluate the efficiency of the simulation technique.

High-level synthesis benchmark circuits are computing structures with data-paths comprising building blocks with regular structures. Each of these building blocks can consist of thousands of gates, making the circuits too computationally

intensive to simulate at the gate-level for a large number of vectors with no fault dropping.

A. Analysis of Building Blocks

Four arithmetic building blocks were simulated with pseudo-random test vectors: a multiplier, an adder, a subtractor, and a comparator. All simulations were run on a Sun SparcStation 5 with 32 MB of RAM and the results are summarized in Table I. A 16-bit data-path is used for all experiments. The simulation times are provided for the case in which detected faults are dropped from the list of active faults, as well as the case in which no fault dropping is performed; that is, all faults are simulated for all vectors. Complete fault coverage is obtained for all blocks.

TABLE I: Simulation of building blocks

Module	Observation	Faults	Vectors (100% FC)	CPU time (sec)	
				Dropping	No dropping
mul16	32-bit product	7103	280	3.2	14.3
	16 MSB (TRUNC)	7008	245096	857.2	12512
	16 MSB (XOR4)	7032	661	1.7	35.5
	16 MSB (XOR2)	7056	320	0.9	17.9
	16 MSB (XOR1)	7109	280	0.8	17.0
	16 MSB (ADD)	7103	280	0.7	14.8
adder16	16-bit sum (no carry)	456	26	0.003	0.040
sub16	16-bit difference	522	22	0.003	0.33
cmp16	1-bit (carry of adder)	445	178397	39.9	244.8
	1-bit (XOR4)	463	154	0.020	0.24
	1-bit (XOR2)	487	85	0.011	0.17
	1-bit (XOR1)	540	67	0.009	0.16

The simulation speed is also shown as the number of evaluations that can be performed per second for each of the building blocks. This is done for both the faulty and fault-free models, with the results shown in Table II. For example, the faulty 16×16 multiplier can be evaluated approximately 160,000 times per second. The time needed to evaluate a fault-free multiplier is approximately one order of magnitude less than that needed to evaluate the functional model of the faulty multiplier. The rate of equivalent gate evaluations refers to the performance achieved with the functional model, relative to structural simulation. That is, if the faulty multiplier can be evaluated 160,000 times per second, and its circuit consists of 1200 gates, then this is equivalent to simulating 1.92×10^8 gates per second.

TABLE II: Simulation performance of building blocks

Module	No. gates	Block eval/sec		Equiv. gate eval/sec	
		Fault-free	Faulty	Fault-free	Faulty
mul16	1200	1,700,000	160,000	2,040,000,000	192,000,000
adder16	80	2,400,000	353,000	198,000,000	28,200,000
sub16	96	3,022,000	409,000	290,000,000	39,300,000
cmp16	96	2,600,000	394,000	250,000,000	37,800,000

Memory requirements are drastically reduced when functional fault modeling is used. No netlist or internal values need to be stored for the module. Only one copy of the fault model needs to be kept, as well as copies of the memory elements in the circuits (e.g. registers) for all faults. The lookup table for the full-adder cell requires little memory. The cell has 3 single-bit inputs (8 possible input combinations) and 30 internal faults (uncollapsed fault set). For those 240 possible input-fault combinations, the values of the sum and carry-out bits need to be stored. If those 2 bits for every input-fault are stored in one byte of memory, the table requires 240 bytes. If the fault-free outputs are stored in the table as well (instead of being calculated using the boolean equations), then 248 bytes are required for the table. This memory can be reduced 4 times (to 62 bytes) by making use of all 8 bits in each byte of the array, instead of using only 2 bits.

Since a 16-bit data-path is being used, the 32-bit output of the multiplier is truncated by taking the most significant 16 bits. However, the truncation makes many faults in the circuit hard to observe as they have to propagate through many full-adder cells before reaching the observed outputs. This can be seen by the large number of input vectors that need to be applied to reach complete fault coverage. A number of modifications can be implemented to increase the observability of most faults in test mode, and hence decrease the test length. In the XOR1 scheme (Figure 3), the 16 least significant bits of the output are XORed together, and the result is fed to the carry-in of the adder chain in the last row of the multiplier, i.e. to the input of $C_{0,\,m}$ which is normally set to 0. The XOR2 and XOR4 schemes are the same as XOR1, except that the number of XOR gates is reduced as every second or fourth bit is XORed, respectively. For example, in the XOR4 scheme, the carry-in bit of $C_{0,\,m}$ is set to $P_4 \oplus P_8 \oplus P_{12}$. These modifications reduce the test length by three orders of magnitude. In the ADD scheme, the 16 LSBs are added to the 16 MSBs. The applicability of the ADD scheme is dependent on how the data-path is implemented, and whether it is feasible to perform the operation using existing hardware.

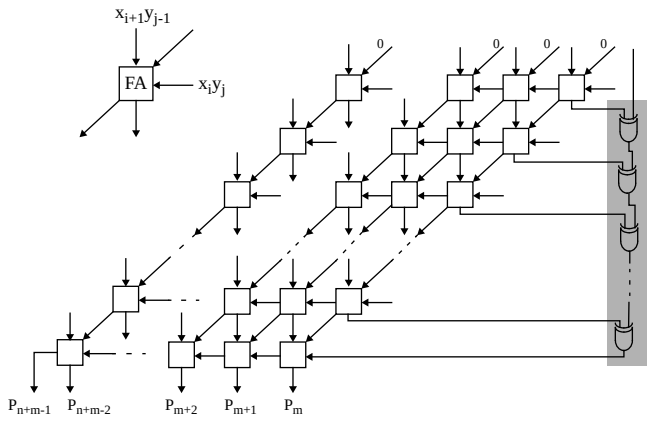

Fig. 3: Array multiplier with XOR gates

The comparator circuit suffers from an observability limitation similar to that of the multiplier. It is implemented as a subtractor, with the carry-out of the adder being observed. Hence, faults have to propagate through to the carry-out output of the adder to be observed. The modifications in this case consist of XORing selected output bits from the adder with the output bit. This leads to a significant reduction in the test length - more than three orders of magnitude.

In summary, a number of arithmetic building blocks were simulated with and without fault dropping. The simulation times illustrate the efficiency of the technique. Complete fault coverage can be achieved for all the blocks. However, some blocks feature limited observability that requires a large number of test vectors to achieve complete fault coverage. A number of modifications to these blocks can be devised to enhance the observability, and hence drastically decrease the test length required for complete fault coverage.

B. High-Level Synthesis Benchmarks

Two high-level synthesis benchmark circuits are analyzed in this section: an elliptical wave filter and a band-pass filter. The implementation of the elliptical wave filter contains 3 multipliers, 3 adders, and 17 registers. A total of 22677 faults are injected on the adders and multipliers. For each input vector applied, the circuit performs a total of 34 operations: 26 additions and 8 multiplications.

The input vectors are generated using an additive generator, which uses existing adders in the circuit. Compaction is done by converting the indicated addition operation to rotate-carry addition [3].

Table III indicates the fault coverage achieved and simulation time required to apply a given number of vectors. Note that for each vector applied, all operations shown in the data flow graph (DFG) are performed (26 additions and 8 multiplications). This is the equivalent of simulating 11680 gates for each of the 22678 machines - for every vector applied.

TABLE III: EWF benchmark simulation

No. vectors	CPU time (sec)	Fault coverage (%)				
		TRUNC	XOR4	XOR2	XOR1	ADD
10	10.9	71.896	84.513	89.772	93.043	95.251
100	105	87.924	99.283	99.902	99.943	99.921
1000	1050	94.377	99.982	100.00	99.996	100.00
10,000	10,485	98.214	100.00	100.00	100.00	100.00
100,000	10,9898	99.781	100.00	100.00	100.00	100.00

The fault coverage, when the 16 least-significant bits the multipliers' outputs are truncated, does not reach 100% due to the observability limitation of the multiplier. Full fault coverage is reached for each of the modified circuits within a reasonable test length.

The second benchmark circuit simulated is the band-pass filter. A total of 15640 faults are injected on the 2 multipliers, 2 adders, and 1 subtractor. There are 13 registers in the circuit. A total of 29 operations are performed for each input vector: 12 multiplications, 10 additions, and 7 subtractions. The simulation results are shown in Table IV. As with the elliptical wave filter circuit, complete fault coverage is achieved for all cases except when the multiplier output is truncated.

TABLE IV: BPF benchmark simulation

No. vectors	CPU time (sec)	Fault coverage (%)				
		TRUNC	XOR4	XOR2	XOR1	ADD
10	9.8	68.097	88.766	94.275	94.397	99.258
100	98	88.647	100.00	100.00	100.00	100.00
1000	974	95.573	100.00	100.00	100.00	100.00
10,000	10,739	99.592	100.00	100.00	100.00	100.00
100,000	$\approx$ 100, 000	99.987	100.00	100.00	100.00	100.00

IV. Conclusions

In this paper, it has been shown that the regularity of the structures of several building blocks commonly used in data-path architectures can be used to derive accurate functional fault models. The faulty response is typically computed by isolating the fault effect and superposing it on the fault-free result. This leads to very efficient fault simulation of these blocks, reducing simulation of hundreds or thousands of gates to a few instructions. The technique can be incorporated into a variety of simulation environments for accelerating the fault simulation of regular blocks that lend themselves to this modeling approach. Furthermore, memory usage is significantly reduced since no netlist needs to be instantiated for the blocks, and no internal values need to be stored for the different faulty machines.

References

[1] S. Gupta, J. Rajski, and J. Tyszer, "Test Pattern Generation Based on Arithmetic Operations," *Proc. of the ICCAD*, pp. 117-124, Nov. 1994.

[2] J. Rajski and J. Tyszer, "Accumulator-Based Compaction of Test Responses," *IEEE Trans. on Computers*, pp. 643-650, June 1993.

[3] M. Kassab, J. Rajski, and J. Tyszer, "Accumulator-Based Compaction for Built-In Self Test of Data-path Architectures," *1st Asian Test Symposium*, pp. 241-246, Hiroshima, Japan, Nov. 1992.

[4] T. M. Niermann and W. T. Cheng and J. H. Patel, "PROOFS: A Fast, Memory Efficient Sequential Circuit Fault Simulator," *IEEE Trans. on Computer-Aided Design*, pp. 198-207, Feb. 1992.

[5] H. K. Lee and D. S. Ha, "HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits," *Proc. 29th ACM/IEEE Design Automation Conference*, pp. 336-340, June 1992.

[6] S. Ghosh, "Behavioral-Level Fault Simulation," *IEEE Design and Test of Computers*, pp. 31-42, June 1988.

[7] S. Gai and P. L. Montessoro and F. Somenzi, "MOZART: A Concurrent Multilevel Simulator," *IEEE Trans. on Computer-Aided Design*, pp. 1005-1016, Sep. 1988.

[8] W. Meyer and R. Camposano, "Fast Hierarchical Multi-Level Fault Simulation of Sequential Circuits with Switch-Level Accuracy," *Proc. 30th ACM/IEEE Design Automation Conference*, pp. 515-519, 1993.

[9] N. Mukherjee, M. Kassab, J. Rajski, and J. Tyszer, "Arithmetic Built-In Self Test for High-Level Synthesis," *VLSI Test Symposium*, 1995.

[10] I. Koren, "Computer Arithmetic Algorithms," Prentice Hall, 1993.