

# On Built-In Self-Test for Multipliers

Mary D. Pulukuri, George J. Starr, and Charles E. Stroud  
 Department of Electrical and Computer Engineering  
 Auburn University  
 Auburn, Alabama 36849-5201 USA

**Abstract**—We evaluate some of the previously proposed test algorithms and approaches for various types of multipliers. We present methods to effectively test multipliers independent of their architecture and to achieve greater than 99% single stuck-at gate-level fault coverage with a simple 8-bit or 9-bit binary up-counter and some multiplexers. Finally, we discuss testing the multipliers present in most current Field Programmable Gate Arrays (FPGAs).<sup>1</sup>

## I. INTRODUCTION AND BACKGROUND

Multipliers are extensively used in digital signal processors (DSPs) [1]. Many current Field Programmable Gate Arrays (FPGAs) incorporate embedded cores such as memory and DSPs in addition to logic blocks. For example, Xilinx Virtex-2 and Spartan-3 series FPGAs incorporate 18×18K-bit random access memories (RAMs) with 18×18-bit multipliers associated with each RAM in addition to the configurable logic blocks (CLBs). Xilinx Virtex-4 and Virtex-5 series FPGAs incorporate DSP cores [2]. Since the multipliers are often part of larger complex circuits like DSPs they are less controllable and less observable and hence an effective Built-In Self-Test (BIST) approach is required [3]. While developing BIST approaches for the multipliers in DSP modules in Virtex-4 FPGAs, we investigated various previously proposed test approaches in an attempt to find an architecture independent test approach that effectively tests multipliers regardless of their architecture.

This paper presents the results of our investigation and evaluation. Section II gives an overview of the architectures of various multipliers. Section III overviews the prior BIST approaches proposed for testing multipliers. Section IV presents our fault simulation analysis of these previously proposed test algorithms for the multiplier architectures presented in Section II. Section V discusses the application of the results of our analysis to various FPGAs. Finally, Section VI summarizes and concludes the paper.

## II. MULTIPLIER ARCHITECTURES

In general an  $N \times M$ -bit multiplication is performed by multiplying each of the  $M$ -bits of the multiplier with all  $N$  bits of the multiplicand resulting in  $M$   $N$ -bit wide partial

products. These partial products are then added together to generate the final product. Faster multiplication can be achieved by reducing the number of partial products [4]. The multiplier inside the DSPs in Virtex-4 FPGAs is an 18×18-bit 2's complement multiplier that produces two 36-bit partial products [5]. The DSPs in Virtex-5 devices have larger 25×18-bit 2's complement multipliers that produce two 43-bit partial products [6]. These partial products are sign-extended to 48 bits and added using a separate 48-bit carry look ahead (CLA) adder as illustrated in Figure 1.

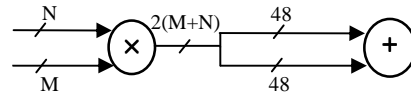


Fig. 1. Multiplication in Virtex-4 and Virtex-5 DSPs [5][6]

Knowing the architecture of the multiplier is important to design an effective test but the Xilinx data sheets for the multipliers and DSPs present inside the FPGAs provide little if any information about the architecture of these multipliers. Since there is no mention of clock cycle latencies in data sheets we can eliminate sequential logic architectures. Therefore, we performed fault simulations on various non-sequential multipliers to find an effective test approach that tests multipliers regardless of their architecture. The various choices for multipliers in our fault simulation analysis include array multipliers, signed array or Baugh Wooley multipliers, modified Booth multipliers, modified Booth Wallace tree multipliers and a custom multiplier obtained by making a modification to the modified Booth multiplier. These multipliers are summarized in Table I and described in more detail below.

Table I. Multipliers

#	Type	Notes
1	Unsigned Array	unsigned
2	Baugh Wooley	signed array
3	Modified Booth	carry look-ahead adder sums partial products
4	Modified Booth	carry look-ahead adder sums final partial products
5	Wallace Tree	carry-select adder sums final partial products
6		ripple-carry adder sums final partial products
7	Custom	carry look-ahead adder sums final partial products

Unsigned array multipliers (type #1 in Table I) are the simplest of all multipliers where the partial products generated using an array of AND gates are added using an array of full adders [4].

<sup>1</sup> This work was supported in part by the National Science Foundation Grant CNS-0708962.

Similarly, in *signed array* multipliers, also called *Baugh Wooley* multipliers (type #2 in Table I), the partial products are generated using an array of AND and NAND gates and are added using an array of full adders [7].

In *Modified Booth* multipliers (type #3 in Table I), partial products are calculated using the modified Booth algorithm [1] which uses a recoding technique to calculate partial products using a series of add, subtract and shift operations. The recoding technique reduces the number of partial products by half by recognizing patterns in the multiplier which results in faster addition and hence faster multiplication [1]. In this algorithm, the partial products are one bit larger than in the traditional array and Baugh Wooley multipliers. Therefore, each partial product will be  $N+1$  bits wide. Any sign extension bits will begin at the  $N+2$  position. We used CLA adders to add the partial products.

In *Modified Booth Wallace tree* multipliers (type #4 - #6 in Table I) addition of partial products that are generated using the modified Booth algorithm is done using a Wallace tree to reduce the number of addition steps [1]. Each column of partial products are added using a multi-stage setup of half and full adders. Each multistage adder circuit generates a sum and a carry which form the two final partial products [1][8]. We used three types of adders (CLA adder, carry select adder, and ripple carry adder) to add the final partial products in three different multiplier implementations indicated by types #4, #5 and #6, respectively, in Table I.

The *Custom* multiplier (type #7 in Table I) is a radix-4 Booth-encoded multiplier with Wallace-tree reduction throughout the partial products where the sign extension has been truncated using a unique reduction technique. The reduction technique is described in more detail in Appendix A at the end of this paper.

One Spartan-3 application note mentions that the multiplier is based on modified Booth architecture [9]. We believe that the modified Booth Wallace tree multiplier (multiplier #4 in Table I) is the most likely candidate based on area and performance analysis [10] and also due to the fact that the modified Booth Wallace tree architecture is the only architecture that results in two final stage partial products similar to the two final stage partial products generated by the multipliers in Virtex-4 and Virtex-5 FPGAs. However, since the architecture of the multiplier is not explicitly mentioned in the datasheets our goal is to find an architecture-independent test approach that effectively tests the multiplier in these devices and any multiplier in general whose architecture is not known.

### III. PRIOR TESTING APPROACHES

Test algorithms for testing modified Booth and modified Booth Wallace tree architectures have been proposed in [11] and [8], respectively. The test algorithms in [11] (referred to as the  $4 \times 4$  algorithm in this paper) and [8] (referred to as the  $5 \times 3$  algorithm in this paper) both use an 8-bit counter to generate 256 test vectors. This is illustrated in

Figure 2 where  $N$  indicates the number of bit-width of the multiplier. In [11] four bits of the counter are replicated and applied to one port of the multiplier and the remaining four bits of the counter are replicated and applied to the other port of the multiplier. In [8] (and later in [12]), five bits of the counter are replicated and applied to the multiplier port that has Booth encoding and the remaining three bits of the counter are replicated and applied to the other port of the multiplier. The authors in [8] do not explicitly mention in their paper that five bits of the counter are applied to the port with the Booth encoding but do illustrate this in a figure. However, the authors in [12] explicitly state that the five bits of the counter are applied to the port with Booth encoding. Both test algorithms can be used to test multipliers of any size and are reported to achieve gate-level single stuck-at fault coverage around 99.8%. A final alternative would be to apply three bits to the Booth encoding port and five bits to the remaining port. However, as in the case of the Xilinx FPGAs, one may not explicitly know which port, if any, incorporates Booth encoding.

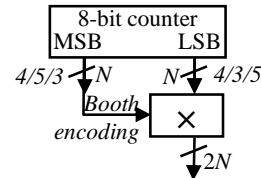


Fig. 2. Multiplier Test Algorithms

Since the  $5 \times 3$  test algorithm was claimed to be better than the  $4 \times 4$  test algorithm by the authors in [8], our original approach to test the multipliers in Virtex-4 devices was to apply both the  $5 \times 3$  and the  $3 \times 5$  test algorithms to ensure that the port with the Booth encoding receives five bits of the counter in either one of the two test sequences and then apply the  $4 \times 4$  test algorithm if the fault coverage improves [2]. This increases the test time but since the number of test vectors applied during each test algorithm is only 256 this is not a concern since download time is the dominant factor in FPGA testing.

### IV. ANALYSIS OF TEST ALGORITHMS

In an attempt to find an architecture-independent multiplier test algorithm we evaluated the  $4 \times 4$ ,  $5 \times 3$ , and  $3 \times 5$  test algorithms by building 8-bit gate-level models of multipliers summarized in Table I using Auburn Simulation Language (ASL) and running fault simulations using Auburn University Simulator (AUSIM) to determine the fault coverage for collapsed single stuck-at gate-level faults. In order to identify undetectable faults in each of our seven 8-bit multiplier implementations an exhaustive ( $2^{16}$ ) set of test patterns was applied. These undetectable faults were then removed from the fault list used to evaluate the fault coverage achieved by each of the three test algorithms. The fault simulation results are summarized in Table II. For each multiplier type, the total number of collapsed single stuck-at gate-level faults is given along with the total number of faults

detected by the exhaustive set of test patterns. Using the list of detectable faults for each multiplier implementation, each of the three test algorithms was applied individually and collectively to determine the fault coverage associated with each test algorithm and possible combinations. In each case the number of faults detected is reported along with the percent fault coverage in parentheses. The test algorithm(s) providing the highest fault coverage are highlighted in light gray and those providing the second highest fault coverage are highlighted in darker gray for each multiplier type.

**Table II. Fault Simulation Results**

Multi-plier	Total Faults	Test Algorithm Number faults detected (% fault coverage)					
		Exhaustive	4×4 [11]	5×3 [8][12]	3×5	5×3 & 3×5	5×3, 3×5 & 4×4
#1	1648	1644 (100)	1644 (100)	1644 (100)	1621 (98.60)	1644 (100)	1644 (100)
#2	1648	1644 (100)	1644 (100)	1644 (100)	1644 (100)	1644 (100)	1644 (100)
#3	2499	2196 (100)	2180 (99.27)	2168 (98.72)	2179 (99.23)	2182 (99.36)	2193 (99.86)
#4	2184	2090 (100)	2061 (98.61)	2068 (98.95)	2070 (99.04)	2071 (99.09)	2074 (99.23)
#5	2422	2243 (100)	2215 (98.75)	2217 (98.84)	2218 (98.89)	2222 (99.06)	2228 (99.33)
#6	2021	1962 (100)	1937 (98.73)	1944 (99.08)	1944 (99.08)	1944 (99.08)	1947 (99.24)
#7	1908	1805 (100)	1781 (98.67)	1787 (99.00)	1785 (98.89)	1791 (99.22)	1793 (99.34)

From the fault simulation results in Table II we observe that the 3×5 test algorithm provides the best fault coverage for five of the seven multipliers. It is interesting to note that this includes the target multiplier in references [8] and [12] that propose the 5×3 test algorithm as the best testing approach. Applying both the 3×5 and the 5×3 test algorithms provides better fault coverage as can be observed in second to last column of Table II. However, applying all three test algorithms (4×4, 5×3 and 3×5) provides the best fault coverage regardless of the architectural implementation of the multiplier as can be observed in the last column of Table II.

Multiple test algorithms can be easily applied by using multiplexers to change the inputs since the same 8-bit counter (approximately 80 gates) can be used to provide and apply all three test algorithms. In addition to the 8-bit counter, the maximum area overhead for applying one test algorithm to an  $N$ -bit multiplier is  $2N$  2:1 multiplexers (approximately  $8N$  gates total) to choose between system and test inputs. The maximum area overhead for applying two test algorithms to an  $N$ -bit multiplier is  $2N$  3:1 multiplexers (approximately  $12N$  gates total) and an additional counter bit (approximately 90 gates for a 9-bit counter) to control the application of the two test algorithm sequences. The maximum area overhead for applying all three test algorithms to an  $N$ -bit multiplier is  $2N$  4:1 multiplexers (approximately  $18N$  gates total) and two additional counter bits (approximately 100 gates for a 10-bit counter) to control the application of the test algorithms.

Therefore, the area overhead for applying one test algorithm is approximately 17% while the area overhead for

applying three test algorithms is approximately 29% for our 8-bit multiplier example. It should be noted that this is the worst case area overhead since the area overhead reduces with larger multipliers and since the synthesis tools may reduce the number and/or sizes of multiplexers. For example, the area overhead would be reduced in cases where more than one test algorithm is applied since some inputs of the multiplier ports receive the same counter bits. Since the area overhead is small and the size of the test pattern generator is the same regardless of the size of the multiplier, this is a very area-efficient BIST approach.

## V. APPLICATION TO MULTIPLIERS IN FPGAS

From the fault simulation results in Table II for the multiplier #4 in Table I (which is the most likely candidate for DSPs in Virtex-4 and Virtex-5 devices as discussed in Section II), we observed that the two additional faults the 3×5 test algorithm detects are in the CLA adder portion of the multiplier. Since the CLA adder is separated from the multiplier in Virtex-4 and Virtex-5 devices and a separate test algorithm is used to test the CLA adder [2][13], the 3×5 and 5×3 test algorithms provide the same fault coverage for the multiplier in these devices. Nonetheless, we apply both the 3×5 and the 5×3 test algorithms to test for bridging faults on the cascade routing between adjacent DSPs in the FPGAs. The three additional faults that the 4×4 test algorithm detects in addition to the 3×5 and 5×3 test algorithms for the multiplier #4 in Table I as illustrated in the last column of Table II are in the Wallace tree part of the multiplier. Therefore, a better approach to test the multipliers in Virtex-4 and Virtex-5 devices would be to apply the 3×5 and the 4×4 test algorithms.

This same approach can also be applied to multipliers in Xilinx Spartan-3ADSP, Spartan-6 and Virtex-6 FPGAs. The architecture of the multipliers in Spartan-3ADSP and Spartan-6 FPGAs is similar to the architecture of multipliers in the DSPs of Virtex-4 FPGAs and the architecture of multipliers in Virtex-6 FPGAs is similar to the architecture of multipliers in the DSPs of Virtex-5 FPGAs. Unlike the DSPs in Virtex-4 and Virtex-5 FPGAs, the adder is not separated from the multipliers in Spartan-3, Spartan-3E, Spartan-3A, Virtex-2 and Virtex-2 Pro FPGAs. For these multipliers applying the 3×5 and the 5×3 test algorithms is the most efficient test approach since these two test algorithms detect a good number of faults in the CLA adder portion of the multiplier as well. However applying all three test algorithms provides the best fault coverage with increased test time and area overhead. These same approaches can also be applied to multipliers in Altera Cyclone-II FPGAs as well as the multipliers in the DSPs in Altera Stratix, Stratix-II, Stratix-GX FPGAs [14].

## VI. SUMMARY AND CONCLUSION

An area efficient BIST approach was presented for effectively testing multipliers irrespective of their architecture. Using two previously proposed multiplier BIST

approaches, this paper analyzes the gate-level single stuck-at fault coverage for seven different multiplier architectures to find architecture independent combinations of these test algorithms. For multipliers whose architecture is not known applying both 5×3 and the 3×5 test algorithms is the most effective method in terms of fault coverage and number of test patterns. However, applying all three test algorithms (5×3, 3×5 and 4×4) is the best way to test multipliers. Since each of the three test algorithms apply 256 test vectors generated by a simple 8-bit binary up-counter for testing any size multiplier, the increase in area overhead and test time for applying more than one test algorithm is relatively small.

#### REFERENCES

- [1] A. Cooper, "Parallel Architecture Modified Booth Multiplier", *Proc. IEEE Electronic Circuits and Systems*, vol. 135, no. 3, pp. 125-128, 1998
- [2] M. Pulkuri and C. Stroud, "Built-In Self-Test of Digital Signal Processors in Virtex-4 FPGAs", *Proc. IEEE Southeastern Symp. on System Theory*, pp. 34-38, 2009
- [3] D. Gizopoulos, A. Paschalis, Y. Zorian, "An Effective Built-In Self-Test Scheme for Parallel Multipliers", *IEEE Trans. on Computers*, vol. 48, no. 9, pp 936-950, 1999
- [4] R. Stans, "The Testability of a Modified Booth Multiplier", *Proc. IEEE European Test Conf.*, pp 286-293, 1989
- [5] "XtremeDSP for Virtex-4 FPGAs," User Guide UG073 (v2.7), Xilinx Inc., 2008 (available at www.xilinx.com)
- [6] "Virtex-5 XtremeDSP Design Considerations," User Guide UG193 (v3.1), Xilinx Inc., 2008 (available at www.xilinx.com)
- [7] W. Wolf, *Modern VLSI Design*, 3<sup>rd</sup> Ed., Prentice Hall PTR, 2002
- [8] A. Paschalis, N. Kranitis, M. Psarakis, D. Gizopoulos and Y. Zorian, "An Effective BIST Architecture for Fast Multiplier Cores", *Proc. Design, Automation and Test in Europe Conf.*, pp. 117-121, 1999
- [9] "Using Embedded Multipliers in Spartan-3 FPGAs", Application note XAPP467 (v1.1), Xilinx Inc., 2003 (available at www.xilinx.com)
- [10] S. Shah, A. Al-Khalili and D. Al-Khalili, "Comparison of 32-bit Multipliers for Various Performance Measures", *Proc. IEEE Int. Conf on Microelectronics*, pp. 75-80, 2000
- [11] D. Gizopoulos, A. Paschalis and Y. Zorian, "An Effective BIST Scheme for Booth Multipliers", *Proc. IEEE Int. Test Conf.*, pp. 824-833, 1995
- [12] D. Bakalis, E. Kalligeros, D. Nikolos, H. Vergos and G. Alexiou, "Low Power BIST for Wallace Tree-based Fast Multipliers", *Proc. IEEE Int. Symp. on Quality of Electronic Design*, pp. 433, 2000
- [13] M. Pulkuri and C. Stroud, "On Built-In Self-Test for Adders," *J. Electronic Testing: Theory and Applications*, vol. 25, no. 6, pp. 343-346, 2009
- [14] "Implementing Multipliers in FPGA Devices," Application note 306 (v3.0), Altera Corp., 2004 (available at www.altera.com)

#### APPENDIX A: CUSTOM MULTIPLIER DESIGN

The basis of the reduction technique used in the custom multiplier (type #7 in Table I) is centered on the axiom that any 2's complement negative number added to its complement is equal to zero. Therefore, adding a number

and its 2's complements along with the partial products will be equivalent to just adding the partial products. To the set of generated partial products that come from the modified Booth algorithm we logically add the negative number  $-2^{N+1}$ . This number will appear as all 1s at and above the position  $N+1$  and all 0s below. For any positive sign extended number, 0s will be used to pad the upper bits. Adding a number of all 0s to all 1s will result in all of the padding bits and the sign bit becoming all 1s. Though if this number of all 1s is added to a negative number which pads with all 1s, the result will be that all of the padding bits will still be all 1s, but the lowest sign bit will have inverted from 1 to 0. If we keep the sign bit and the first padded bit from each stage in this process, then this algorithm can be simplified.

In this model the most significant bit (MSB), the sign bit, of each partial product is inverted and a binary 1 is added to the MSB+1 position. All higher order bits that were used for sign extension are modified to 0. As adding 0s will always yield 0, we can remove this extra logic from the circuit. This takes what would have been a long partial product with sign extension bits up to  $N+M$  bits long and reduces it to a fixed  $N+2$  bit partial product for all partial products. In an 8×8-bit modified Booth multiplier this can reduce the partial products by 12 bits. Larger multipliers will see an even larger reduction in sign extension bits as illustrated in Table III. For example, a 25×25-bit modified Booth multiplier will be reduced by 144 bits. This represents a 29% reduction in partial product bits from the original modified 25×25-bit Booth model. For symmetric multipliers, the number of reduced partial product bits is expressed in the equation below.

$$reduced\ Bits = \sum_{n=1}^{MultiplierSize/2} \frac{MultiplierSize}{2} - 2n$$

**Table III. Reduction in Partial Product Bits by Size of Multiplier**

Size of N in N×N Multiplier	8	9	10	11	12	13	14	15	25
Reduced bits	12	16	20	25	30	36	42	49	144

This reduction size of the partial product terms comes at a price. To reduce these higher order sign extension bits, we must add one more number to the partial product to compensate for adding in this previous value. This new number is simply the 2's complement of the previous value. This will always be a power-of-2 with the value of  $2^{N+1}$  in the modified Booth model. There are two solutions to resolving this issue. The first is to add this partial product in as a carry in bit within the Wallace tree structure. This will simply convert one half-adder into a full-adder. The second alternative is to modify the entity that produces the partial products to account for this number during the generating of the partial products. For simplicity in both size and speed, the first method was selected and this single bit was added into the Wallace tree structure in our model. When combined with the fact that the multiplier type is a modified Booth Wallace tree multiplier, the end result is an extremely compact model for a multiplier.