Using high-level synthesis to implement on-line testability

Petros Oikonomakos, Mark Zwolinski, *Senior Member, IEEE*

Abstract-- **On-line testing increases system reliability, which is essential in a number of applications. High-level synthesis, on the other hand, offers fast time-to-market and allows quick and painless design space exploration. In this work, we investigate on-line testing in the high-level synthesis context. Further, we propose a new technique (inversion testing) and demonstrate its potential benefits.**

Index terms–on-line testing, reliable systems, operational redundancy, high-level synthesis

1 INTRODUCTION

On-line testing targets physical failures, that is failures that occur while the system is operating, as opposed to fabrication errors or defects [1]. A physical failure may be due to environmental factors, like temperature, radiation or pressure. On the other hand, on-line testing resources can result in a significant hardware penalty or performance degradation to the original design, thus increasing cost. There are cases, however, where reliability is more important than cost or speed, or the environment is so hostile that we can expect relatively frequent physical failures – and this is where on-line testing fits in. These cases include (but are not restricted to) flight, space, automotive, medical and industrial electronics [2,7]. Further, VLSI technology is moving towards deeper submicron integration and reduced power supply voltages. These make systems more susceptible to physical failures and increase the need for on-line testing [6]. With the addition of self-recovering or self-repair techniques, extremely robust systems can be produced [8].

High-level (or behavioural) synthesis [9] provides a designer with the capability to consider several realisations of a conceptual design, in a fast and efficient manner. In this way, the designer can estimate the characteristics (area, performance, testability, power dissipation) of each realisation and choose the one that better accommodates the specification. Incorporating on-line testing in a high-level synthesis environment would let the designer estimate the cost and performance penalty associated with on-line testability, and finally make the decision to keep or drop that capability (at an early stage in the design process), according to the reliability requirements and target application.

Petros Oikonomakos and Mark Zwolinski are with the Electronic Systems Design Group, Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton SO17 1BJ, UK. Email: {po00r, mz}@ecs.soton.ac.uk.

2 FOUNDATION AND PREVIOUS WORK

Several on-line testing techniques have been proposed. Roughly speaking, they can be grouped into three main categories :

- self-checking design
- on-line built-in self-test (BIST)
- monitoring analogue characteristics.

We will not consider the last two categories further in this paper.

In *self-checking design* [7], the circuit under test (CUT) is augmented such that its output is encoded according to some error-detecting code. Provided that it falls within the detecting capabilities of the code, a failure results in the output being a non-code word, which is detected by a checker. Related to the self-checking design scheme are the well-established [7] *self-testing*, *fault-secure*, *totally self-checking* and *code-disjoint* properties that the CUT and checker (respectively) must satisfy in order to avoid fault escapes. Several error-detecting codes have been proposed [1], each one with its unique characteristics regarding error-detection capabilities, hardware overhead and checker

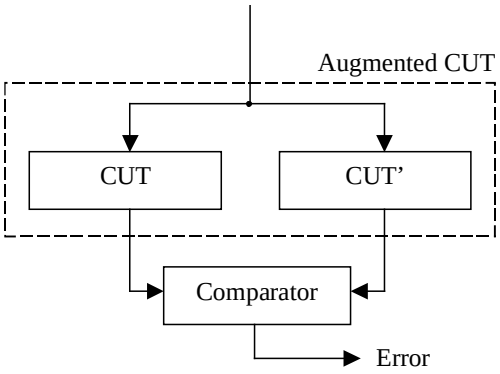

Figure 1. Duplication

complexity. Of particular interest for our work are duplication-related self-checking techniques. Figure 1 shows the *duplication testing* scheme. In this technique, the "augmented" CUT consists of two copies of the original

circuit. The second copy (CUT') must be functionally equivalent to CUT; if it is structurally equivalent as well, then we talk about *identical* duplication. Otherwise, we have the *diverse* duplication case. It has been demonstrated [4,5] that diverse duplication is preferable, due to the protection it offers against common mode failures. Note that duplication testing is fault secure by its very nature [7].

Clearly, simple physical duplication results in a hardware penalty of over 100%. When considering data paths (as opposed to isolated CUTs), a very attractive alternative is *algorithmic duplication*. Consider the example data flow

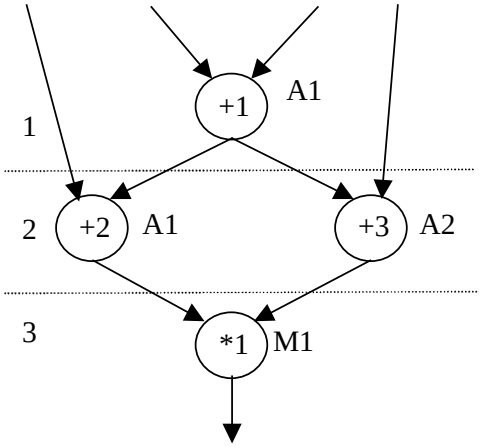

Figure 2. Example DFG

graph (DFG) of Figure 2. Four operations are performed over three control steps. Operations (additions) +1 and +2 are bound to module (adder) A1; +3 is bound to A2 and multiplication *1 is bound to multiplier M1. So the data path consist of 2 adders and 1 multiplier. Physically duplicating the data path obviously leads to 4 adders and 2 multipliers. However, note that adder A2 is idle during control steps (cs) 1 and 3, while adder A1 is idle* during cs 3. So A2 can be used during cs 1 to duplicate operation +1 (at the cost of an introduced multiplexer). Similarly, we can duplicate +2 and +3 during cs 3, binding the duplicate operations to modules A2 and A1 respectively. This way, we duplicate *operations* (as opposed to *operators*), saving hardware. Two recent examples of algorithmic duplication techniques are presented in [3,8]. In [8], algorithmic duplication is combined with *rollback* to provide error recovery, while in [3] module *differentiation* is exploited to provide fault identification.

Having demonstrated the scope for hardware savings that algorithmic duplication introduces, it is this particular form of self-checking design that we favour for incorporation in high-level synthesis. Furthermore, we supplement it with a new, allied technique, which we present in the next section.

*A functional module is considered to be *idle* during a control step, if it is not processing any useful data and is not expected to produce any valid result during that particular control step.

3 INVERSION TESTING

The proposed *inversion testing* scheme (Figure 3) is a variation of duplication testing. INV(CUT) stands for a circuit performing the *inverse* (arithmetic or logical) operation of CUT. Pairs of CUT/ INV(CUT) can be simple, for example adder/subtractor or left shifter/right shifter, or complex such as chains of simple modules. As shown in Figure 3, functional inputs are compared with the inverted functional outputs. In the fault free case, they will be equal. If either of the modules fails, the comparator detects and signals the failure.

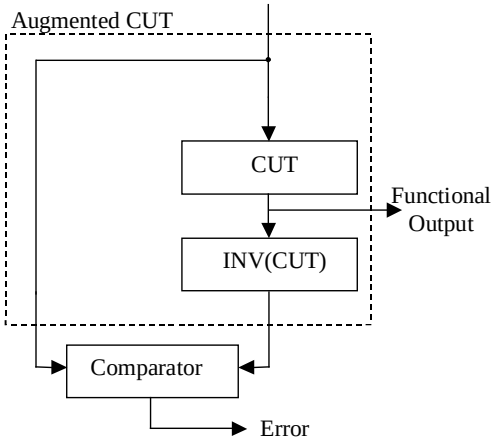

Figure 3. Inversion Testing

The inversion testing scheme is fault secure by its nature (under the common single fault assumption), just like duplication schemes. In general, there is no advantage in physically inverting a module for testing purposes, as opposed to physically duplicating it. However, if modules of type INV(CUT) are idle during particular clock cycles in a design, then *algorithmic* inversion can lead to more hardware efficient implementations. A simple case to

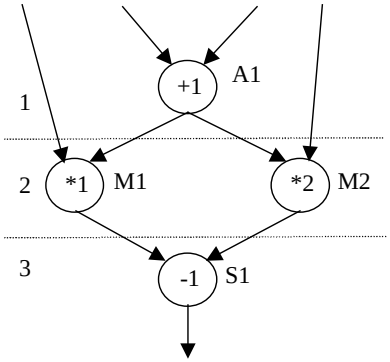

Figure 4. Motivating example

illustrate this concept is shown in Figure 4. This DFG consists of one addition, one subtraction and two multiplications. The circuit is implemented by one adder (A1), one sutractor (S1) and two multipliers (M1, M2). Adder A1 is idle during cs 2 and 3, while subtractor S1 is

idle during cs 1 and 2. Operation +1 can be inverted by subtractor S1 at cs 2, while operation –1 can also be tested during cs 2 by A1 (assuming that the operations are in a control loop so that after cs 3 control returns to cs 1). So, the result of each operation is verified without the need to introduce *any* additional adders or subtractors. Applying duplication to the same DFG would necessarily lead to physically duplicating both modules. In that particular case, applying inversion saves 50% of the functional modules used if duplication is applied.

Comparing Figures 1 and 3, we note that the inverse operation is performed after the original, while the duplicate can be performed simultaneously. However, given relaxed clock period requirements and a powerful synthesis tool, an operation and its inverse can be chained in the same control step. If an operation and its duplicate or inverse are scheduled in the same control step, then a potential failure is detected during that same control step. If the duplicate or inverse is scheduled n control steps later, then any failure is detected with an *error latency* of n clock cycles. Experimental results presented in the following section illustrate the effects of chaining and the effects of preventing chaining in system speed and in error latency.

4 Experimental results

In this section we present our first experimental results. We have been experimenting with three High-Level Synthesis Workshop benchmarks, namely *tseng* (1991), *diffeq* (1992) and *qrs* (1995). Tseng is an example design with no physical significance, consisting of 8 arithmetic and logical operations and used extensively for high-level synthesis experimentation and evaluation. Diffeq is a differential equation solver (also very popular for experimental purposes), while qrs is a more complex design representing a medical electronics application and consisting of more than 70 operations. All three benchmarks have been synthesized using the MOODS Behavioural Synthesis Suite [9,10], developed at the University of Southampton. Commercial tools have been used for lower level synthesis and implementation. Synplicity Synplify version 6.2 has been used for RT level synthesis, while Xilinx Design Manager version 3.1i has been used for implementation. For each benchmark, we have changed the high-level VHDL descriptions to synthesize and compare on-line testing methods. Tables 1 through 4 summarize our results. In all our examples, hardware usage figures are the ones reported by the implementer tool (Xilinx Design Manager). The implementer tool timing analysis reports are also used to determine the maximum achievable frequency. In all our experiments, 16-bit arithmetic has been used.

Tables 1 and 2 show implementation results for the tseng example. The *original* version has no testing strategy at all (that is why the error latency is infinite). In the *duplicated* version, we have applied the duplication testing strategy. We supplemented the VHDL description by adding duplicate and comparison operations to every operation of the original version. For example, a simple addition operation :

```
v8i := v3i + v5i;
```

is replaced by the series of operations :

```
v8i := v3i + v5i;
sc1 := v3i + v5i;
failed <= sc1 /= v8i;
```

and an additional port (**failed**) is added to the design. In the *inverted_1* version, we have applied the inversion testing scheme for those operations that can be inverted within the particular data path, ie whenever the “inverse” module is present in the design. Again, we achieved this by modifying the VHDL description in a manner similar to the duplication case shown above. Further, in the *inverted_2* version, we force the high-level synthesis tool to schedule inverse operations one clock cycle after valid operations, thus preventing chaining.

Comparing the results in Table 1 for the duplicated and the inverted_1 versions, we note that inverted_1 has a smaller hardware overhead. This is made clearer in Table 2, where functional module usage is shown. Further, Table 1 shows that errors are detected at the same control step as they occur (so error latency is 0) in both duplicated and inverted_1 cases and performance degradation (in terms of clock cycles) is also the same; however, chaining of operation/inverse pairs within the same control step results in the maximum achievable clock frequency being 7 times lower in the inverted_1 version. On the other hand, in the inverted_2 version an additional 4 cycles are needed, but the maximum achievable clock frequency is the same as in the duplicated version. The hardware overhead is more than for the inverted_1 version but is still less than the duplicated version. Functional module usage is the same as in inverted_1 (reported in Table 2); the extra hardware overhead is due to registers introduced to store values across clock cycle boundaries. Non-zero error latency is introduced; indeed, out of 8 operations, 4 are inverted and checked with an error latency of 1. Error latency is 0 for the other 4 (duplicated) ones, giving an average of 0.5.

Similar to the tseng benchmark are the results obtained with the diffeq example, summarised in Table 3. This benchmark consists of 12 operations, 4 of which are suitable for inversion. Version names have the same meaning as in the previous example.

In Table 4, our results for the qrs benchmark are presented. The qrs application is composed mainly of additions, subtractions and right shift operations. For this benchmark, we synthesized duplicated (*dupl_1*) and inverted (*inv_1*) versions exactly as before. This time, though, we tried two different target technologies. Further, we also tried the *dupl_2* and *inv_2* versions, where we used several **sc** and **failed** signals instead of one (as in the above code fragment) and ORed the **failed** ones to produce the **failed** single-pin output port.

In contrast to what we have seen up to now, comparing different implementations shows that there can be cases (ie

technologies) when duplication results in smaller overhead than inversion. Further, comparing _1 and _2 versions shows how unacceptably high performance degradation (133.3% in _1 versions) can be reduced considerably (about 30% in _2 versions), at the expense of some more hardware overhead (due to introduced registers for the additional signals). Thus, for a design of some realistic complexity, the well-known high-level synthesis concept of trading-off area for speed or vice versa applies equally to on-line test insertion.

5 CONCLUSION AND FUTURE WORK

In this work, we have presented a new self-testing technique (inversion testing), and justified it through examples as a competitive alternative or supplement to algorithmic duplication. Through our experiments, we have been able to identify factors that can be used to evaluate the quality of our testing scheme and direct the synthesis process towards the most beneficial one, according to the particular designer's needs and specifications. Namely, introduced hardware overhead and performance degradation, restrictions imposed to clock frequency and average error latency are among those factors. In addition, target technology has a significant role in the whole process, while traditional high-level synthesis dilemmas and trade-offs are still valid.

Our future work regards implementation of on-line testing within the MOODS synthesis system, in a way that it will be *transparent* to the user (no modification of VHDL code will be required). To achieve this, we are already working towards *quantifying* on-line testability so that a traditional high-level synthesis *cost function* can be enhanced to include the on-line testability criterion, in addition to the traditional ones (area, delay [9,10]). When these two tasks are finished, we expect to have developed the first (to our knowledge) high-level synthesis system to provide on-line testing, in a totally automated manner, according to the designer's requirements.

Version	Resource Usage		Speed Parameters		Testing Penalty		Average Error Latency (cycles)
	Slices	Tristate Buffers	Cycles	Maximum Frequency	Hardware Overhead (slices %)	Performance Degradation (cycles %)	
Original	137	400	7	50 MHz	-	-	∞
Duplicated	166	706	9	35 MHz	21.2	28.6	0
Inverted_1	158	754	9	5 MHz	15.3	28.6	0
Inverted_2	161	770	13	35 MHz	17.5	85.7	0.5

Table 1 : Tseng benchmark synthesis results (Target technology Xilinx Virtex XCV800 FPGA)

Version	adders	subtractors	OR gates	AND gates	left shifters	right shifters	comparators
Original	1	1	1	1	1	1	-
Duplicated	2	2	2	2	2	2	1
Inverted_1	1	1	2	2	2	2	1

Table 2 : Tseng benchmark functional module usage

Version	Resource Usage		Speed Parameters		Testing Penalty		Average Error Latency (cycles)
	Slices	Tristate Buffers	Cycles	Maximum Frequency	Hardware Overhead (slices %)	Performance Degradation (cycles %)	
Original	233	578	13	25 MHz	-	-	∞
Duplicated	322	964	15	25 MHz	38.2	15.4	0
Inverted_1	306	948	15	4 MHz	31.3	15.4	0
Inverted_2	316	996	18	25 MHz	35.6	38.5	0.33

Table 3 : Diffeq benchmark synthesis results (Target technology Xilinx Virtex XCV800 FPGA)

Version	Target Technology	Resource Usage		Cycles	Testing Penalty	
		Slices	Tristate Buffers		Hardware Overhead (slices %)	Performance Degradation (cycles %)
Original	Xilinx Virtex XCV800	465	2910	33	-	-
Original	Xilinx XC95288XV	548	2910	33	-	-
Dupl_1	Xilinx Virtex XCV800	589	4874	77	26.7	133.3
Inv_1	Xilinx Virtex XCV800	600	5000	77	29.0	133.3
Dupl_1	Xilinx XC95288XV	702	4874	77	28.1	133.3
Inv_1	Xilinx XC95288XV	671	5000	77	22.4	133.3
Dupl_2	Xilinx Virtex XCV800	654	5208	42	40.6	27.3
Inv_2	Xilinx Virtex XCV800	630	5441	43	35.5	30.3

Table 4 : Qrs benchmark synthesis results

6 REFERENCES

[1] M. Abramovici, M.A. Breuer, A.D. Friedman, "Digital Systems Testing and Testable Design", IEEE Press 1990.

[2] H. Al-Asaad, B.T. Murray, J.P. Hayes, "Online BIST for embedded systems", IEEE Design & Test of Computers, Vol. 15, No. 4, October–December 1998, p. 17-24.

[3] S.N. Hamilton, A. Orailoglu, "On-line test for fault-secure fault identification", IEEE Transactions on VLSI, Vol. 8, No. 4, August 2000, p. 446-452.

[4] S. Mitra, E.J. McCluskey, "Which concurrent error detection scheme to choose?", IEEE International Test Conference, 2000, p. 985-994.

[5] S. Mitra, N.R. Saxena, E.J. McCluskey, "Fault Escapes in Duplex Systems", IEEE VLSI Test Symposium, 2000, p. 453-458.

[6] M. Nicolaidis, L. Anghel, "Concurrent Checking for VLSI", Microelectronic Engineering, Vol. 49, No. 1-2, November 1999, p. 139-156.

[7] M. Nicolaidis, Y. Zorian, "On-line Testing for VLSI – A compendium of approaches", Journal of Electronic Testing – Theory and Applications, Vol. 12, No. 1-2, February-April 1998, p. 7-20.

[8] A. Orailoglu, R. Karri, "Automatic Synthesis of Self-Recovering VLSI Systems", IEEE Transactions on Computers, Vol. 45, No. 2, February 1996, p. 131-142.

[9] A.C. Williams, "A Behavioural VHDL synthesis system using data path optimisation", PhD Thesis, University of Southampton, 1997.

[10] A.C. Williams, A.D. Brown, M. Zwolinski, "Simultaneous optimisation of dynamic power, area and delay in behavioural synthesis", IEE Proceedings – Computers and Digital Techniques, Vol. 147, No. 6, November 2000, p. 383-390.