UNIVERSITY OF SOUTHAMPTON

SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

High-level Synthesis for On-line Testability

Petros Oikonomakos

December, 2004

A thesis submitted for the title of
Doctor of Philosophy.

UNIVERSITY OF SOUTHAMPTON

High-level Synthesis for On-line Testability

by

Petros Oikonomakos

A thesis submitted for the degree of
Doctor of Philosophy.

School of Electronics and Computer Science,
University of Southampton

December, 2004

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

High-level Synthesis for On-line Testability

by Petros Oikonomakos

On-line testing increases hardware reliability, which is essential in safety-critical applications, particularly in hostile operating conditions. High-level synthesis, on the other hand, offers fast time-to-market and allows quick and painless design space exploration. This thesis details the realisation of on-line testability, in the form of self-checking design, within a high-level synthesis environment. The MOODS (Multiple Objective Optimisation in Data and control path Synthesis) high-level synthesis suite is used for the implementation of this concept.

A high-level synthesis tool typically outputs controller / datapath hardware architectures. These two parts pose different self-checking problems that require different solutions. Datapath self-checking is realised using duplication and inversion testing schemes within the circuit data-flow graph. The challenge therein is to identify and implement suitable high-level transformations and algorithms to enable the automatic addition of self-checking properties to the system functionality. This further involves the introduction of an expression quantifying on-line testability and including it in the standard high-level synthesis cost function, thus materialising a three-dimensional design space, to be explored by the designer feeding the synthesis tool with the problem specifications and constraints.

In contrast, controller self-checking is not implemented within the synthesis process, but is rather the result of a post-processing synthesis step, directly applying an appropriate checker to the system control signals. Nevertheless, challenges include choosing suitable self-checking techniques, achieving the Totally Self-Checking (TSC) goal, and investigating ways to reuse any existing datapath self-checking resources for controller on-line testability. Solutions based both on parity-checking and on straightforward 1-hot checking are given, again providing the designer with enhanced opportunities for time-efficient experimentation in search for the best solution in every given synthesis project.

The self-checking structures are finally verified theoretically and experimentally, through fault simulation. Overall, the enhanced version of the MOODS system, produced as a result of this research work, enables the implementation of reliable electronics efficiently, so that reliability-critical applications can be accommodated in a mass production context.

Contents

List of Figures

List of Tables

Acknowledgements

This work was funded partly by the Engineering and Physical Sciences Research Council and partly by the School of Electronics and Computer Science of the University of Southampton. I am grateful to both organisations for their financial support. I am also grateful to my supervisor Dr Mark Zwolinski for guiding me when I needed guidance and trusting me when I did not. I acknowledge the contribution of Professor Bashir Al-Hashimi, who served as the internal examiner of both my first year report and my transfer viva, and on both occasions provided the very much appreciated third person's perspective of my work. I would also like to thank my current employers in the Computer Laboratory of the University of Cambridge, and in particular my immediate supervisor Dr Simon Moore, for providing the financial and professional stability, that were invaluable for the final completion of this thesis.

I am grateful to all my former colleagues in Southampton, for both the academic and the social support that they provided. This should include everybody who worked in rooms 3049, 3051 and 3053 of the Mountbatten Building anytime between October 2000 and October 2003; a special mention is needed for my good friend Manoj Gaur and his family.

I would also like to thank the numerous members of the international research community who exchanged ideas with me either in person or through email. Several of them provided copies of their research papers and clarifications on them. The "primacy of honour" among them belongs to Dr Steffen Tarnick who enormously helped me to understand and practically implement his previously published theoretical work.

This thesis is dedicated to all the members of my family, especially my father Eleftherios and my mother Chrysoula, who once upon a time denied herself the chance to pursue her own PhD in order to support the education and well-being of five other people. I am also particularly grateful to my brother George for introducing me to the art of Design Automation in the first place.

Finally, I would like to dedicate this thesis to all the people who have been patient enough to contribute to my spiritual growth, particularly by leading me to and keeping me in the Holy Church of Christ.

Chapter 1

Introduction

Hardware reliability is an area of electronic design, attaining more and more importance in recent years. The typical solution for the increase of the on-field reliability of digital electronic components is *on-line testing*. As the term suggests, on-line testing targets and detects chip failures that occur while the system is operating, as opposed to fabrication errors or defects [1] that are detected during manufacturing tests. Typically, on-line detection is followed by corrective action, thus implementing *fault tolerance*. On-line testing should essentially be viewed as the first step towards fault tolerance.

In earlier days of computing [2], on-line testing solutions were devised primarily for protection against failures that were attributed to initially minor manufacturing imperfections in chips. Over time, aging, corrosion, electrical, thermal and mechanical stress exacerbated the effects of such imperfections, thus eventually developing permanent logic faults. Clearly, when such faults were anticipated in safety-critical applications during the expected lifetime of an electronic component, it was imperative that a detection and recovery mechanism be configured. As fabrication quality improved, the reliability risk associated with such phenomena decreased rapidly and on-line testing lost a lot of its significance in the 1980s; indeed, the testing literature is particularly poor in on-line testing techniques during that decade.

This situation began to change in the beginning of the 1990s and changed fundamentally around and after 1995, with the continuous shrinking in transistor sizes and the decrease in operating voltage levels (low-power computing). The push for ever-reducing geometries in order to meet the requirements of Moore's law [3] prompted engineers to look for reliability "workarounds", driven by the need to produce *operative* electronics out of *imper-*

fect fabrication lines. Fault tolerance was identified as such a workaround [4]. Putting aside this re-surfacing of fault tolerance for permanent faults, the real driving force for reliability in the last ten years has been the increasing number of problems with *single-event upsets* (*SEUs*) in modern electronics. A SEU is a transient fault that corrupts a logic value either in a memory or in functional logic only *once*; however, this one-off failure, or the superposition of multiple such failures, is enough to disturb the correct operation of the system. SEUs (also termed *soft* errors) are primarily attributed to environmental radiation effects, in principle alpha particle cosmic radiation or atmospheric neutrons. Such radiation can induce electrical charges at particular capacitive parts of a circuit; given the reduced voltage levels of modern low-power electronics, this charge is often comparable to the charge stored in the said parts during normal operation. As a result, the logic value determined by the amount of charge stored in the particular location is likely to change. Another explanation of radiation upsets is that particles that hit the body of transistors in the OFF state can induce enough energy to create a channel, thus unexpectedly turning the transistor ON and potentially corrupting the logic value at its drain.

In the light of this situation, on-line testing and fault tolerance have gained significant importance in modern electronics. Safety-critical or even life-critical applications cannot risk failures and thus require constant testing. These applications include space and aviation, automotive and medical electronics. The situation is particularly severe in high altitudes and in space, where the density of cosmic particles is higher than on sea level. Further, it is predicted that technology rapidly approaches the point where even everyday commodity applications will need some sort of protection against radiation upsets [5]. Interestingly, for all these reasons the industry experts of the consortium publishing the International Technology Roadmap for Semiconductors [4] have identified fault tolerance as one of the five major "crosscutting" challenges in semiconductor design. Moreover, on-line testing and its extension fault tolerance have been proved useful to straightforwardly enhance manufacturing yield, by providing protection even against manufacturing defects [6]. Finally, on-line testing in the form of self-checking has also been proposed as a countermeasure against optical tampering in security applications [7].

In this era of digital electronics that require more and more functions on a single chip, electronic design automation (EDA) tools are used throughout the whole process of chip design. Naturally, significant efforts are also invested in tool development, both in indus-

trial environments and in academic research groups. High-level synthesis is a particular trend within the EDA context, whereby electronic systems are produced automatically by a synthesis tool when the tool is fed merely by an algorithmic description of the desired *behaviour*, and *automatically* extracts all structural and timing information. The benefits are fast time-to-market, fast and efficient design space exploration, and optimisation at the highest level of abstraction. Clearly, mass production industrial environments can greatly benefit from such characteristics.

The Multiple Objective Optimisation in Data and control path Synthesis (MOODS) tool is a high-level synthesis suite, developed in the University of Southampton [8]. It is an example of academic research in the field of high-level synthesis, and its particular characteristic is automatically trading-off different system parameters (area, delay), in its attempt to simultaneously satisfy all (typically contradicting) designer requirements.

1.1 Objectives and thesis organisation

As on-line testing becomes more and more relevant to industry sectors that require high volumes of production, it becomes obvious that it would be beneficial to develop a high-level synthesis tool, capable of automatically producing on-line testable systems, while simultaneously optimising for the traditional synthesis goal of area and delay. No present synthesis tool offers this. It is this gap in the art of semiconductor electronic design that this work fills. The fundamentals of *high-level synthesis for on-line testability* are provided. The development part of the work enhances the existing MOODS system to provide on-line testability. The whole foundation and implementation are tested through numerous experiments, and the reliability of the overall produced solutions is assessed.

This thesis comprises eight chapters. Chapters 2 – 4 cover background material, as in the following.

Chapter 2 provides a thorough overview of electronic testing. Conventional off-line testing is briefly covered; however, overwhelming emphasis is naturally given to digital on-line testing techniques.

Chapter 3 describes high-level synthesis. The basic terminology and definitions are initially given, followed by a more detailed overview of the MOODS High-Level Synthesis Suite.

Chapter 4 gives elements of fault simulation. In this thesis, fault simulation is used for reliability evaluation purposes; therefore, the basics are given and a few recent representative techniques demonstrating the state-of-the-art are presented.

Chapter 5 – 7 describe original work, along the following lines.

Chapter 5 presents the work carried out in the direction of providing *datapath* self-checking design for controller / datapath pairs produced by high-level synthesis processes. The most appropriate on-line testing technique is identified, and details of the implementation with the MOODS system are given. Extensive experimental results are shown and commented on.

Chapter 6 focuses on the controller part of a controller / datapath architecture, and provides six alternative self-checking solutions for it, taking into account multiple communicating control units, and utilising existing datapath self-checking resources. These techniques are all implemented into MOODS, and more experimental results are presented.

Chapter 7 provides a theoretical and experimental evaluation of the reliability of the on-line testable system under the presence of single or multiple physical failures.

Finally, chapter 8 presents ideas for future research based on this thesis and concludes it by summarizing its most important contributions.

Three appendices are also included in this thesis. Appendix A is a brief "User's Guide" of the produced high-level synthesis for on-line testability variation of MOODS. Appendix B shows the benchmark designs used in the experiments of chapters 5 and 6. Finally, Appendix C shows the research papers written and unofficial presentations given as part of the work that lead to the production of this thesis.

Chapter 2

An Overview of Electronic Testing

This chapter provides background information on electronic testing theory and various practical testing techniques, most of them developed in the 1990s. The presentation herein begins with a very brief overview of off-line *scan-based Design-For-Testability* and *Built-In Self-Test (BIST)* in section 2.1, while section 2.2 describes various on-line testing techniques in detail. Finally, section 2.3 summarizes the chapter.

The behaviour of an electronic system under the presence of a logical fault can be evaluated using the *structural stuck-at* fault model assumption [1], under which a wire in a system is considered to retain a logical value ("0" or "1"), regardless of the value driving it, thus producing a logical error whenever the driving line assumes the opposite ("1" or "0" respectively) value. An alternative structural model is the *bridging fault model* [1], whereby an erroneous short circuit between two wires effectively gives rise to a new elementary logic function (AND or OR). Higher-level functional models also exist. For instance, given the functional *hardware description language* (*HDL*) code of an electronic system, a whole multi-bit variable can be modelled as being stuck at a particular arithmetic value. Another example of functional fault modelling are the *stuck-at-true / struck-at-false* faults, conceivable whenever a functional description contains conditional statements. Generally speaking, the structural bit-wise stuck-at fault model has most often been favoured over other models in the research literature, for its simplicity, representative power, and ease of use. It is also fully adopted in this chapter and generally throughout the whole of this thesis. By convention, a stuck-at-0 wire that fails to take the "1" value of the line driving it, is said to assume the *1/0* or *D* value. Likewise, a stuck-at-1 wire that fails to take the "0" value of its driving line, is said to assume the *0/1* or $\overline{D}$ value. These conventional notations, taken from [1], will be used hereafter.

2.1 Off-line testing

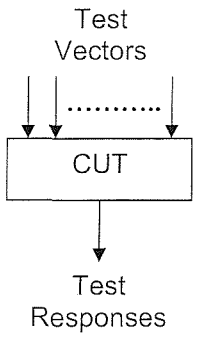
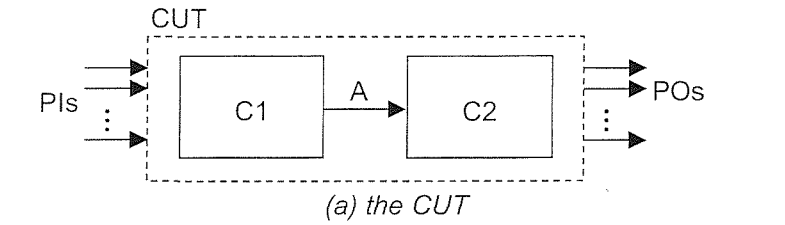
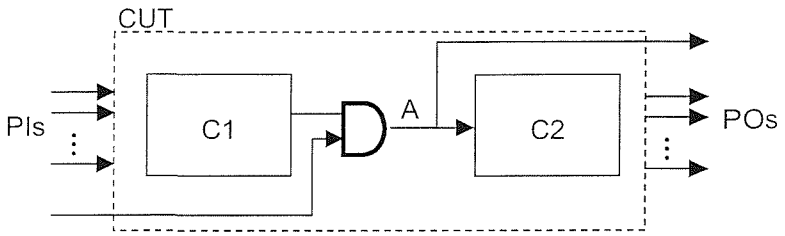


Figure 2.1. Off-line electronic testing

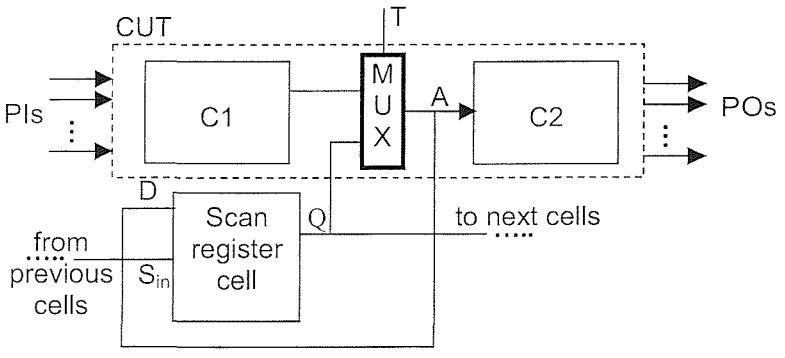
The general off-line testing scheme is depicted in Figure 2.1. The *Circuit Under Test (CUT)* is taken off-line (that is, its normal operation is suspended), *test vectors / test patterns* are applied to its inputs, and *test responses* are read at the output(s). The test responses are compared against the expected fault-free responses, and mismatches signify faulty situations. Test vectors are provided either externally, by *Automatic Test Equipment (ATE)*, or internally by dedicated hardware embedded in the system (chip or board). A comprehensive account of early electronic testing approaches can be found in [1]. Some elementary concepts are provided here, since they are needed for the foundation of this work; further and more recent advances are not covered because they exceed its scope.



(a) the CUT



(b) control and observation point insertion



(c) employing a scan register cell

Figure 2.2. DFT in an example CUT model

2.1.1 Scan-based Design-For-Testability

This subsection deals with testing using externally applied vectors. Consider Figure 2.2a. The CUT is assumed to comprise subcircuits C1 and C2, communicating through a single line A. The abbreviations *PIs* and *POs* refer to the system *Primary Inputs* and *Primary Outputs* respectively. In order for a stuck-at-x, $x \in \{0,1\}$ type fault at line A to be tested against, the test vector at the PIs and the initial conditions in C1 must be such that A is driven to the $\bar{x}$ value under fault-free operation. If such a vector can be found and such conditions reached, then line A is said to be $\bar{x}$*-controllable*. Further, in order for the effect of the considered fault to be observed, the test vector and conditions in C2 must be such that the erroneous value in line A corrupts one or more of the POs. Once again, if this is possible, then line A is *observable*. The term *Design-For-Testability (DFT)* refers to the family of design techniques that aim at increasing system controllability and observability, often trading-off chip area and / or performance.

Figure 2.2b shows a first approach towards DFT, namely *control and observation* (collectively *test*) *point insertion*. Line A is made directly 0-controllable through the insertion of an additional AND gate (shown in bold), controlled by an additional PI. It is also directly connected to an additional PO, thus made observable. 1-controllability can also be achieved using an OR gate, while simultaneous 0- and 1- controllability require a multiplexer. This approach can be very expensive in terms of additional I/O pins when several test points need to be inserted, which is typically the case.

An alternative approach commonly applied is based on *scan registers*. A scan register is a register that has both shift and parallel-load capabilities. An n-bit scan register is shown in Figure 2.3. Scan register cells in the figure are normal flip-flops, augmented with a control

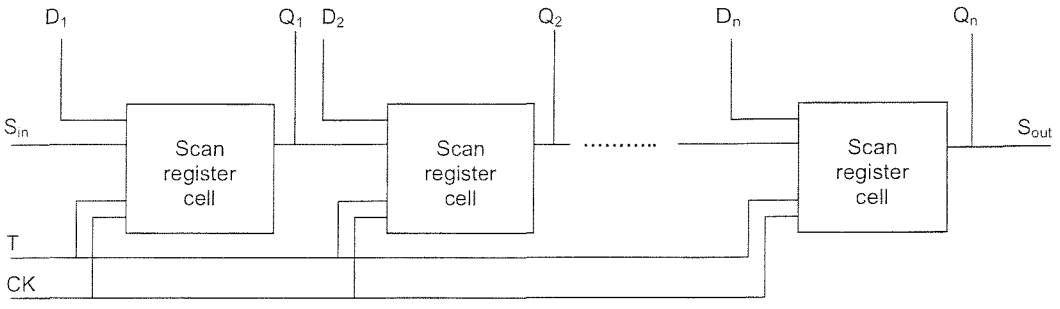


Figure 2.3. A scan register

input T, determining normal or test mode operation. In normal mode (T=0), the register is loaded with functional inputs through the parallel-load input ports $D_1 - D_n$. In test mode, data are shifted into the register through the S_{in} primary input port, and / or shifted out through the S_{out} primary output port.

Figure 2.2c depicts how a scan register cell can be utilised for DFT purposes in the example of 2.2a. It is assumed that the cell shown is actually part of an overall scan register, or chain of registers, providing test point functionality for the whole design. In normal mode operation, the multiplexer (MUX) propagates the functional value produced by C1. In contrast, in test mode, the value provided by the scan flip-flop is fed to C2 instead. Therefore all that is needed to directly control point A is to feed the scan chain with the appropriate bit value, and apply the appropriate number of clock pulses, so that this value reaches the relevant scan cell. Further, in normal mode, the value of A is always registered at the scan cell through port D. Therefore, in order to directly observe it, the scan chain can be clocked as many times as needed for the appropriate value to reach the scan output port S_{out} (Figure 2.3). This way, testability improvements are achieved using 2 or 3 primary input ports (S_{in}, T, and perhaps a dedicated scan clock, which can be different from the functional circuit clock), and only 1 primary output port (S_{out}). Scan registers can be pre-existing functional system registers, augmented to accommodate test mode shifting. If this is not possible for a particular system (e.g. due to the absence of enough functional registers), then dedicated scan registers can be added.

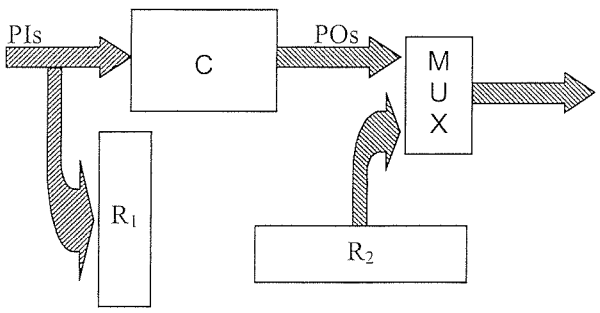


Figure 2.4. Boundary Scan

Using the scan-based DFT approach, systematic *boundary scan* architectures can be formulated, as Figure 2.4 shows. In the figure, block C represents a segment of the considered system, while R_1 and R_2 are scan registers. R_1 is used to observe the PIs of C (effectively the outputs of the previous segment), while R_2 controls the POs of C, through the multiplexer (in effect controlling the inputs to the next segment).

2.1.2 Built-In Self-Test

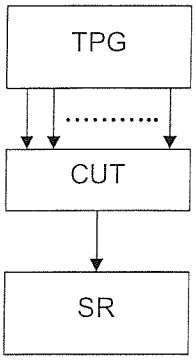


Figure 2.5. Built-In Self-Test

Built-In Self-Test (BIST) deals with the situation when test vectors are provided to the CUT by dedicated hardware, embedded into the system itself, while responses are also analysed and the decision characterising the system as fault-free or faulty is reached once more by hardware in the circuit. A typical BIST configuration employing a *Test Pattern Generator (TPG)* and a *Signature Register (SR)* is shown in Figure 2.5. In the following, properties of TPGs and SRs, and their realisation using *Linear Feedback Shift Registers (LFSRs)* will be briefly discussed.

An n-bit LFSR is presented in Figure 2.6. It is composed of normal flip-flops connected as the figure shows, while for the blocks denoted as c_i it is $c_i \in \{0,1\}$, $1 \le i \le n$. Effectively, the c_i blocks signify the presence or absence of a feedback connection at the relevant point. c_n is always 1. Associated with an n-bit LFSR is its *characteristic polynomial* $P(x)=1+c_1x+ +c_2x^2+\ldots+c_nx^n$. The LFSR of Figure 2.6 is *autonomous*, meaning that it has no inputs but the required clock signal.

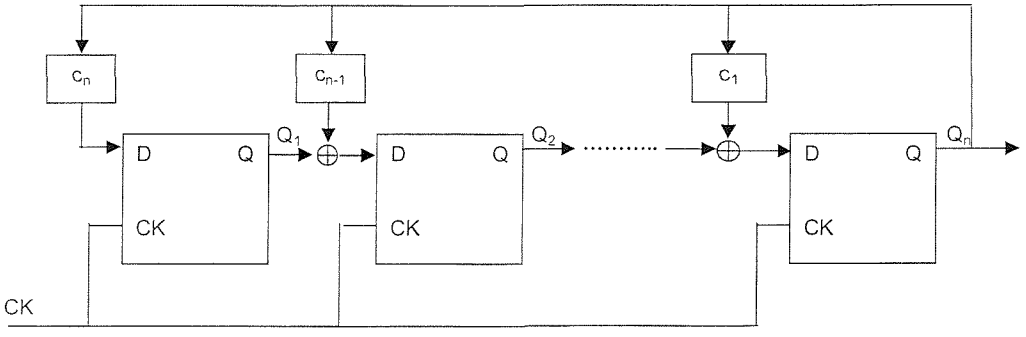


Figure 2.6. An autonomous n-bit LFSR

It can be shown that when P(x) is *primitive* [1], then all n-bit vectors except the all-zeros vector successively appear in the outputs Q_i of the LFSR, provided that it is initialised with a non-zero vector. This property can be exploited when *exhaustive* testing is desired for an n-input CUT, by feeding the CUT input ports through the LFSR Q_i outputs, thus applying all 2^{n-1} non-zero vectors to the CUT, effectively utilising the LFSR for TPG pur-

poses. Alternatively, non-exhaustive, deterministic test pattern sequences can be produced, by designing appropriate autonomous LFSRs with non-primitive characteristic polynomials and initialising them with appropriate vectors.

A slightly different LFSR structure that is used as a *Multi-Input Signature Register (MISR)*, is shown in Figure 2.7, where the clock signal is implied but not explicitly shown. This structure is not autonomous; rather, it is fed by the CUT outputs X_i, corresponding to

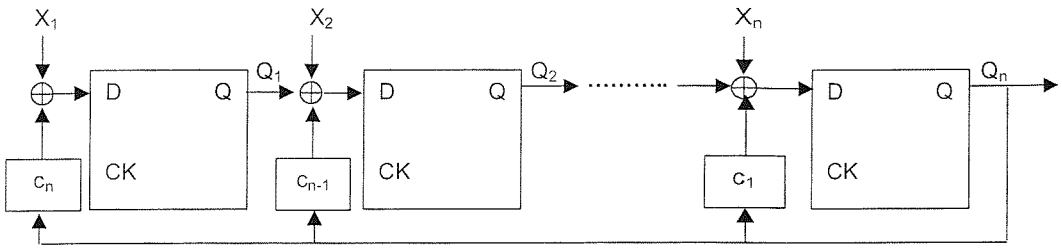


Figure 2.7. An n-bit LFSR configured as an MISR

responses of the circuit to TPG vectors. When all test vectors have been applied and the MISR has processed all test responses, then a unique pattern called a *signature* resides in the MISR. This pattern is then compared against a pre-computed fault-free signature, and any mismatch signifies a faulty situation. In the prevailing terminology, the test responses are often said to be *compressed* by the MISR.

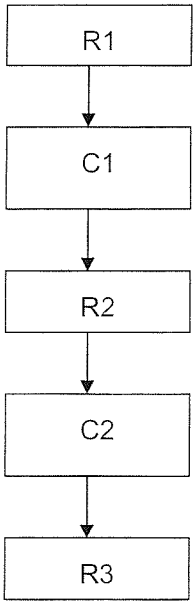


Figure 2.8. BIST in separate test sessions : the need for BILBO registers

In practice, when circuits of realistic sizes are considered, it is often possible and economical to configure functional registers into LFSRs and use them as TPGs or MISRs in test mode, while maintaining their normal functionality during functional mode. This often leads to situations when segments of large circuits are fed by the same TPG or have their test responses compacted by the same MISR. In such cases, BIST has to work in more than one *test sessions*, by partitioning the circuit in groups of segments that do not share BIST resources, and can therefore be tested concurrently. A more

complicated situation is depicted in Figure 2.8. C1 and C2 refer to segments of a large CUT, communicating with functional registers R1, R2 and R3 as shown. Clearly, R1 can be configured as a TPG for C1, while R3 can be an MISR for C2. R2 can be either a TPG for C2 *or* an MISR for C1. In either case, an additional LFSR needs to be introduced, to act as an MISR for C1 or a TPG for C2 respectively. Alternatively, it would be desirable to transform R2 into an architecture that would be able to provide *both* TPG *and* MISR functionality, so that no additional LFSR would be needed. A *Built-In Logic-Block Observation (BILBO)* register provides such dual functionality. A BILBO structure is given in [1] and not repeated here; for the purposes of the present work, it is enough to mention that a BILBO is an LFSR-based structure that can function as either a normal register, a shift register (§2.1.1), an LFSR-based TPG, or an LFSR-based MISR, depending on the values of two control inputs.

2.2 On-line testing

In this section, the state-of-the-art of on-line testing is presented. The discussion is much more thorough than in the off-line case of §2.1, since on-line testing is essentially the focus of this work. Generally speaking, on-line testing techniques can be classified into three main categories, namely :

- self-checking design
- on-line BIST or on-line scan-based DFT
- monitoring analogue electronic parameters (such as current)

Self-checking design consists of encoding module outputs using some error detecting code and then checking some code-specific invariant property (e.g. parity). On-line BIST and on-line scan-based DFT, on the other hand, attempt to use the concepts and structures of §2.1, in the on-line context. Usually existing (off-line) BIST or scan constructs are exploited to perform tests during certain time windows when normal operation is temporarily suspended, either globally for the whole system (*periodic BIST*), or locally (during subsystems' *idle* periods). Monitoring analogue characteristics is useful to detect errors in electrical properties of information signals that either manifest faults that are hard to detect otherwise, or will result in logical faults in the future.

It has to be noted that the above classification is by no means exhaustive. In fact, there are techniques that combine elements of the categories mentioned above. Moreover, there also exist some unique techniques that do not really fall into any of these categories.

2.2.1 Self-checking design

The basic self-checking design scheme is depicted in Figure 2.9 [9]. The functional circuit is such that the output signals it produces are *encoded*. This can be a natural property of the considered circuit; otherwise, the system has to be augmented and redesigned appropriately, according to the chosen *Error-Detecting and / or Correcting Code (EDC / ECC)*. Any circuit fault that corrupts the output bits, such that the output word does not belong to the given code, is detected by the checker. If the output bits are corrupted, but the output is still a code word, then the fault *escapes* detection, and it is said to exceed the *detecting capabilities* of the particular code.

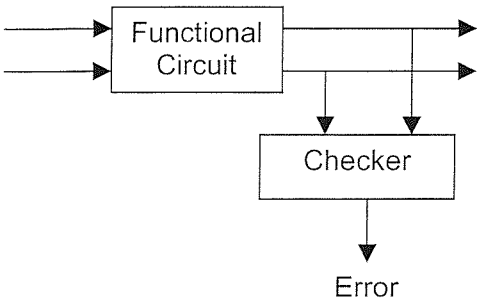


Figure 2.9. Self-checking design.

Before presenting the most important error detecting codes, some fundamental definitions are given. These constitute the theoretical foundation of self-checking design, and theoretically determine the efficiency of practical self-checking schemes. They first appeared in [10], and are repeated in practically every modern publication addressing the issue (for example [11]).

Let f be the Boolean function corresponding to a circuit C. Let X be the set of inputs that C receives and Y be the set of (encoded) outputs that it produces. Furthermore, let Φ be a set of modelled physical faults and φ a fault in Φ. The function of C in the presence of fault φ is denoted by $f(x,\varphi)$, while the fault-free function is denoted by $f(x,0)$.

Definition 2.1: A circuit is *self-testing* with respect to Φ if and only if $\forall \varphi \in \Phi \; \exists x \in X$: $f(x,\varphi) \notin Y$.

In other words, the circuit is self-testing, if for every fault in the specified set, there is at least one functional input that produces a non-code output.

Definition 2.2: A circuit is *fault-secure* with respect to Φ if and only if $\forall x \in X$, $\forall \varphi \in \Phi$: $f(x,\varphi) \notin Y$ or $f(x,\varphi)=f(x,0)$.
In fault-secure circuits, an output in the presence of a single fault is either correct, or a non-code word. That is, it cannot be an incorrect code word.
Definition 2.3: A circuit is *totally self-checking* (TSC) with respect to Φ if and only if it is both self-testing and fault-secure with respect to Φ.

The totally self-checking property is the usual goal when designing the functional circuit. It guarantees that erroneous outputs produced by faults will not be mistaken for correct ones (fault-secure), and that all modelled faults are detectable by the given set of input vectors (self-testing). The fault-secure property is relevant to the structure of the circuit, while the self-testing one is concerned both with the structure and with the set of inputs the circuit receives, and whether or not they are enough to detect all faults in the particular structural realisation of the circuit.

Definition 2.4: A circuit is called *code-disjoint* if and only if $\forall x \in X : f(x,0) \in Y$ and $\forall x \notin X : f(x,0) \notin Y$.
That is, in the fault-free case, a code-disjoint circuit maps code inputs to code outputs and non-code inputs to non-code outputs.
Definition 2.5: A circuit is called a *totally self-checking checker* if and only if it is both totally self-checking and code-disjoint.
In the case of a checker, a produced code word output corresponds to the fault-free indication, while a non-code word output is the error indication. Thus, a totally self-checking checker produces code or non-code outputs according to its inputs (functional circuit outputs) in a fault-free case, while under the presence of a fault it produces either the correct code output or a non-code output. In addition, there is at least one code input that leads to a non-code output under the presence of a fault.

Allied to the above definitions is the following hypothesis [12, 13] :
Hypothesis 2.1: Faults occur one at a time, and the time distance between the occurrences of two consecutive faults is long enough for all the *available* input code words to be applied to the circuit.

It is important to differentiate between the *available* code words, and *all possible* code words. The available code words are all the input code words applied during normal operation, i.e. the members of set X as defined above. However, there can be vectors that are code words, in the sense that they satisfy the characteristic invariant property of the EDC at hand, but do not appear at the circuit input ports during normal operation. In this case, the set X of inputs is said to be *incomplete*. In the rest of this work, Hypothesis 2.1 will be assumed, unless explicitly stated otherwise.

In practice, when designing checkers, it is clearly desirable that they be totally self-checking with respect to the targeted set of faults. In principle, the three properties that the checker must possess are considered separately, in each given situation. A general comment that can be made at this point though, is that the fault-secure condition cannot be fulfilled by a checker whose output is a single bit. Indeed, if $x \in \{0,1\}$ is the fault-free indication value of such a checker, and Z the single-bit output, then for the fault $\{\varphi : Z \text{ stuck-at-}x\}$, any erroneous (i.e. *non-code* word) checker input will produce the *code* single-bit word x [9]. For this purpose, double-output fault-secure checkers are typically used, where by convention the complementary values $\{01,10\}$ correspond to the fault free operation, while any of the remaining $\{00,11\}$ values indicates the presence of a fault.

Further, the code-disjoint property may not always be achievable (an example is considered in chapter 6). In such cases, the checker must be at least designed to achieve the self-testing goal with as few code words as possible, and it must receive as many code inputs as possible. Still, if the code inputs provided are not enough for the self-testing condition to be satisfied, the last resort is *self-exercising* checker design [9]. In such a configuration, the checker is armed with an internal TPG (§2.1.2) that provides the necessary code words. These designs tend to be expensive in hardware overhead; therefore, it can sometimes be tempting to trade-off strict coherence with self-checking design theory for a more hardware-efficient solution, also depending on the size and nature of the design and an analysis of the realistic possibilities of a failure. An example of such a situation is shown in chapter 5.

The most important EDCs and relevant self-checking design considerations are presented in the following subsections §2.2.1.1 - §2.2.1.7. Before that, two classes of EDCs are defined here [1, 9].

Definition 2.6: Given two n-bit words $x:=x_{n-1}x_{n-2}...x_1x_0$ and $y:=y_{n-1}y_{n-2}...y_1y_0$, *x covers y*, if and only if $\forall i,\ n-1 \geq i \geq 0$: if $y_i=1 \Rightarrow x_i=1$.

For example, if x=10111, y=10100, then x covers y, because x has a "1" in every bit position that y has a "1".

Definition 2.7: An EDC is *unordered*, if and only if there are no two different code words x and y, such that x covers y.

Obviously, the above x=10111 and y=10100 words cannot be code words of the same unordered code.

Definition 2.8: In a *separable* EDC, each bit in a given code word is either an *information bit*, or a *check bit*. If the characteristic invariant property of the code is embedded within a code word, so that such a classification is not possible, then the EDC is a *nonseparable* code.

Typically, when a separable code is used, the functional circuit (Figure 2.9) is partitioned into two parts, both fed by the functional input. These are the *functional* part, producing the normal functional output, and the *code prediction* part, independently producing a number of additional bits, ensuring adherence to a code-specific invariant property. By contrast, when a nonseparable code is used, no such partitioning can be conceived. Rather, the produced functional output adheres to the code-specific property by nature or by design.

The theory and definitions of this section are further demonstrated and clarified in the subsequent §2.2.1.1 - §2.2.1.7 through specific examples.

2.2.1.1 Parity codes

When a *parity* code is used, a single check bit is added to the information bits, and it is calculated such that the parity of each code word is constant (odd or even). Parity codes can detect all single or odd multiplicity errors. They are the cheapest possible EDCs, since the check bit is only one and parity checkers are relatively simple [9, 14].

The parity bit of a parity-encoded word is clearly separable from the information bits; therefore, parity codes are separable codes, and normal combinational circuits need to be augmented by a *parity prediction* part, in order to implement a parity self-checking scheme. In the case of an arithmetic functional block, the parity bit can be calculated as

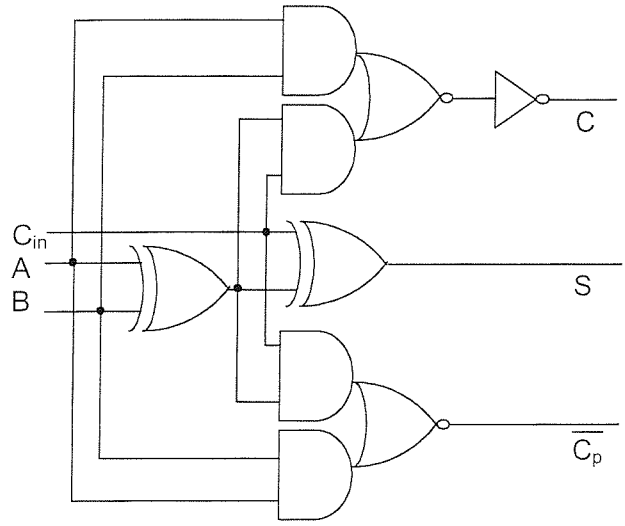


Figure 2.10. Fault-secure full-adder cell with a redundant carry used for parity prediction

the XOR function of the input operands, and of the internal "carry" signal bits of the sub-blocks that constitute the overall circuit [15]. However, in order for the fault-secure property to be achieved and maintained, it is essential that internal bit faults affect an *odd* number of primary output bits. Nicolaidis et al. [15, 16] considered this problem for ripple-carry adders, and for a collection of multiplier and divider structures and proved that fault security is in danger if the functional internal "carry" bits are used for parity prediction. They further came up with the full-adder logic cell with a redundant carry of Figure 2.10 [15], and used it as the basic building block for their designs. A, B and C_{in} in the figure, are the usual addition inputs and input carry, while S, C and C_p are the sum, the output carry and a redundant carry respectively. In a complex multi-bit arithmetic circuit, the redundant carries of internal full adders and the parity bits of input operands are all XORed together; the result of this XOR operation is the predicted parity bit of the circuit output. The authors of [15] analytically prove that this way any single fault in any internal full adder cell may corrupt either *none*, *a single* or *an odd number* of the circuit outputs. Therefore, under the presence of a fault the circuit either produces the correct output, or reverses its parity, hence producing a non-code output. Fault-security is thus achieved.

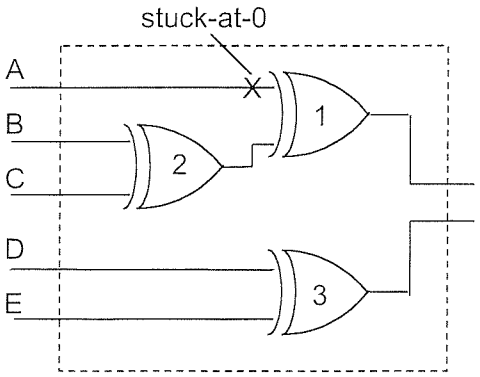


Figure 2.11. A 5-bit odd parity checker

Parity checkers are easily designed as "parity trees" composed of 2-input XOR gates. As stated in [9], splitting the code word in two groups and using two separate parity trees results in a two-output fault secure parity

checker. An example 5-bit odd parity checker is shown in Figure 2.11, fed by the 5-bit odd-parity encoded input word ABCDE, and consisting of two parity trees, composed of two (labelled 1,2) and one (labelled 3) XOR gate(s) respectively.

The simple example of Figure 2.11 is used here to clarify the importance of the self-testing property of §2.2.1. Let us first assume that under fault free operation, the circuit input word ABCDE can only take one of the three values in the following set X={01110, 01000, 00111}. As explained in §2.1.1, a value in the set Y={01,10} signifies correct operation; in the terminology of §2.1.1, Y is the set of code word outputs. Clearly the input words are all odd-parity encoded, and it can easily be confirmed that all three of them produce code word outputs. However, the checker receives only a small subset of all possible 5-bit odd-parity code words. It is not totally self-checking with respect to the set Φ of all stuck-at faults at its constituent gates, since it does not satisfy the self-testing property when fed by these three inputs only. This can easily be confirmed, since for the fault φ shown in Figure 2.11 representing an input of gate 1 to be stuck-at-0, there is no code word $\in X$ that produces a non-code word. φ is therefore undetectable by the particular set of functional inputs, and this potentially hinders the detecting capabilities of the checker. The significance of this can be appreciated if one takes into account that the input word ABCDE is normally the encoded output of a functional circuit, according to Figure 2.9. If the checker has already been hit by fault φ, and at a future point of time an additional fault in the functional circuit causes, for example, the non-code word ABCDE=11110 to appear in the checker input, it is easy to verify that the checker response will be the code word output 01, meaning that the functional circuit fault escapes detection.

From the above example, it is clear that the self-testing property for a checker is not a property of the checker alone; rather it is a property of the checker in the context of the overall system it is part of, since it is the system that provides the code words. Furthermore, it is a property that is strongly related to the actual internal structure of the checker (in this case, the particular arrangement of the 2-input XOR gates), since the set of modelled faults Φ is defined with respect to the structure [17]. Therefore, two behaviourally equivalent (in the fault-free case) parity checkers in the same context, receiving the same code words may not be both self-testing. Two converse problems can be formulated in this context :

- given a parity checker of known structure, it is desirable to identify the minimum number of code word inputs that ensure the self-testing property.
- given a set of parity code word inputs, it is desirable to determine whether or not they can ensure the self-testing property, for some checker structure(s), and, having secured that, to design the corresponding optimally-structured checker.

Regarding the first problem, all XOR gates in the checker should receive all four possible input combinations 00, 01, 10, 11 [18]. This guarantees that the checker will be self-testing, regardless of the actual XOR gate implementation. Khakbaz and McCluskey [17] propose a way to identify a set of code words ensuring this property. They show that it is enough for the two XOR gates that produce the final checker outputs (e.g. gates 1 and 3 in Figure 2.11) to receive these four combinations. These values can be traced back to the checker primary inputs, and thus determine the required code words. For the checker of Figure 2.11, it can easily be verified that {11100, 00010, 10101, 01011} is such a set, and it can also be seen that the remaining XOR gate 2 also receives all possible inputs. Interestingly, this limits the number of necessary code words to *only four* for every given checker structure, *regardless* of bit-width.

As far as the second issue is concerned, the following two lemmas apply (taken from [17, 18], where proofs can also be found) :
Lemma 2.1: Any n-bit parity checker realisation that receives more than 75% of its possible codeword inputs is self-testing.
Lemma 2.2: Consider a $4\times n$ Boolean matrix M, whose rows constitute a test set for an n-bit even (odd) parity checker realised with 2-input XOR gates only. Then M has distinct rows, all rows have even (odd) parity, and each column has exactly two 0s and two 1s.

In the light of these two lemmas, it can now be stated that, given a set of n-bit parity code word inputs, and taking into account that the total number of such possible code words is 2^{n-1}, if the number of code inputs is large enough (more than $3\times 2^{n-3}$), then any 2-input XOR gate realisation of the checker is a self-testing one. Otherwise, if four code words can be found within the given set that satisfy the conditions of Lemma 2.2, then there exists *at least* one 2-input XOR gate realisation of a parity checker that is self-testing. Analytical algorithms to design such checkers, and to optimise them for speed (by minimising

the number of logic gate levels within the checker), can be found in [17, 18], but exceed the purposes of this presentation.

There can be situations when the input set is so incomplete that neither the conditions of Lemma 2.1 nor those of Lemma 2.2 can be satisfied by the available code words. As mentioned in §2.2.1, self-exercising checker design provides a theoretically robust solution for

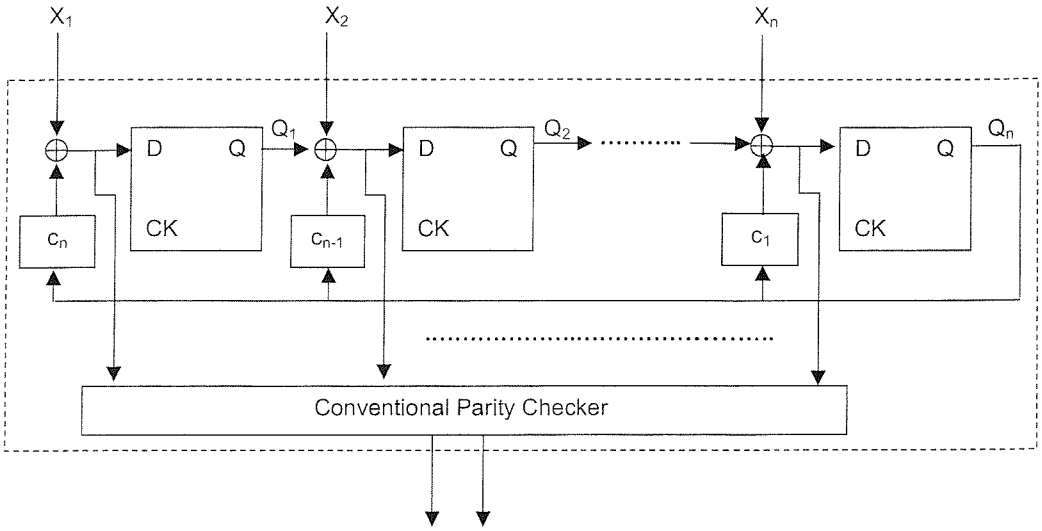


Figure 2.12. n-bit embedded TSC parity checker with error memorizing capability

this problem. The *embedded parity TSC checker with error-memorizing capability* of Figure 2.12 was presented in [12, 19, 20] for this purpose. In the figure, $X_1 \ldots X_n$ is the even-parity encoded checker input. The conventional even parity checker is supplemented by an LFSR structure (similar to the MISR of Figure 2.7). As usual in LFSR designs, it is $c_i \in \{0,1\}$, $1 \le i \le n$, and c_i signify the presence or absence of feedback at the particular point. The design is based on the observation that the even parity code is a *linear* code, that is when two even-parity encoded words are added modulo-2 (XORed), the parity of the resulting word is still even. Therefore, if the LFSR is designed so as to provide all even parity words, then the set of code words that the conventional checker receives is greatly enhanced. The technique applies equally to odd-parity encoding, by simply inverting an arbitrary bit of the input word. The problem of designing a proper LFSR (choosing suitable c_i values) is addressed by the following theory [12, 19, 20].

Definition 2.9: An EDC is called *cyclic* if, for every given code word, a "rotate" (cyclic shift) operation always results in another code word.

Observe that both even and odd parity codes are cyclic. For example, if the word 10001111 (which is an odd-parity 8-bit code word) is rotated left once, the resulting word 00011111 is still odd-parity encoded.

Definition 2.10: Given an n-bit code word $c=c_0c_1\ldots c_{n-1}$, the polynomial

$$c(x)=c_0x^{n-1}+c_1x^{n-2}+\ldots+c_{n-2}x+c_{n-1} \quad (2.1)$$

is called the *code polynomial*.

Definition 2.11: Given an n-bit cyclic code and a polynomial g(x) of degree n-k, g(x) is a *generator polynomial* of the code, if all code polynomials corresponding to all code words are divisible by g(x). The code is then particularly called an *(n,k)* cyclic code.

It can be shown [12, 19, 20] that g(x)=x+1 is a generator polynomial of the even parity code irrespective of the bit width, thus making it an (n,1) cyclic code.

Theorem 2.1: Let g(x) be a generator polynomial of an (n,k) cyclic code, and d(x) a primitive polynomial of degree k. Then the LFSR with the characteristic polynomial p(x)=g(x)d(x) generates all code words of the cyclic code, except for the all-zeros pattern.

Theorem 2.1 was initially introduced and proved by Hsiao et al [21].

It is now clear how the LFSR in Figure 2.12 can be designed. One simply needs to choose a primitive polynomial of degree n-1 and multiply it by the generator polynomial (x+1), to obtain the characteristic polynomial of an LFSR that produces all even parity code words, when seeded with any non-zero even parity encoded word. Tarnick has shown [12, 19, 20] that in order for the overall checker of Figure 2.12 to be totally self-checking, the functional circuit only needs to provide *two* different non-zero code words. The disadvantage of this technique is the hardware penalty that the introduction of the LFSR imposes, but it is the only available solution if normal operation provides only very few code words, and if strict adherence to self-checking theory is desirable.

An interesting application of parity error detecting codes is self-checking state machines. Zeng et al. [22] propose a state encoding and parity prediction technique to check the present state and primary output signals of state machines. The present state signals are checked using a single parity bit. The primary outputs either have a parity bit computed and attached to them, or are partitioned into groups, with a parity prediction scheme applied to each one of the groups. Hardware savings are achieved in the latter case, by allowing logic sharing between different groups. Lakshminarayana et al [23] mention parity prediction as a means to design self-checking controllers of controller / datapath designs,

but do not elaborate on their technique with respect to self-checking theory. This issue is revisited in chapter 6, with a theoretical evaluation, practical considerations, specific implementation examples, and the particular contribution of this thesis.

2.2.1.2 m-out-of-n codes

An *m-out-of-n* code word has exactly m 1s out of n total bits. m-out-of-n (also signified by *m/n*) codes are an example of unordered and nonseparable codes. They detect all single and multiple *unidirectional* errors (that is, errors resulting in corrupted signals where all erroneous bits have the *same* value, either D or $\overline{D}$). The fault-secure property using unordered codes in general, and m/n codes in particular can be achieved for a limited number of functions, and it is practically considered only when the function outputs are already encoded using such a code, by nature. Some attempts to design fault-secure arithmetic units using unordered codes have been reported, but they are not widely adopted, since they are much more expensive to implement than parity prediction schemes [15].

It can easily be verified that an m/n code has exactly $\binom{n}{m} = n!/m!(n-m)!$ code words. For any given n, this value is maximum for m=[n/2] [10]. Therefore, [n/2]-out-of-n codes, often considered for n=2k and referred to as k-out-of-2k, are of particular interest, since they have the maximum *capacity* (in code words) of all other m-out-of-n codes. 1-out-of-n (1/n, also referred to as *1-hot*) codes are another special case of particular interest. They have the minimum code word capacity (only n words), but they frequently appear in computer systems by nature, e.g. in memory address "select" lines.

A lot of work has been presented in the direction of designing totally self-checking checkers for k/2k, 1-hot, as well as generic m/n codes. Historically, the first attempt was reported in [10], also mentioned in [2]. Anderson and Metze [10] used *majority functions* for this purpose.

Definition 2.12: Consider the n_a-bit signal A, and let k_a be the number of bits of A that take the 1 value at a given point of time. Let i be an integer value. The *majority function* $T(k_a \geq i)$ is defined as follows :

$$T(k_a \geq i)=1, \text{ if } k_a \geq i \quad (2.2a)$$

$$T(k_a \geq i)=0, \text{ if } k_a < i \quad (2.2b)$$

A *majority detection circuit* is a circuit implementing a majority function, and is typically realised in a 2-level AND-OR form, by using all possible i-bit combinations of all n_a bits of A as inputs to respective AND gates, and ORing the AND gate outputs.

In [10], the realisation of TSC k/2k checkers is described, through either sum-of-products or product-of-sums combinations of the outputs of suitable majority detection circuits. The k/2k encoded signal that feeds the checker is partitioned into two signals A and B of bit widths n_a and n_b respectively, where $n_a=n_b=k$. Let k_a and k_b be the number of 1s in each signal. Then the logic functions F and G that produce the primary outputs of the checker are described (for example in sum-of-products form) by the following equations :

$$F=\sum_{i=0}^{k} T(k_a \geq i) \times T(k_b \geq k-i),\ i \text{ odd} \tag{2.3a}$$

$$G=\sum_{i=0}^{k} T(k_a \geq i) \times T(k_b \geq k-i),\ i \text{ even} \tag{2.3b}$$

Functions realising checkers for generic m/n codes are also provided, but it is shown that the designs are TSC in the k/2k case only. However, TSC m/n checkers can be implemented based on the k/2k ones, if the scheme of Figure 2.13 is applied. In this scheme, the generic m/n code is first decoded into an $1/\binom{n}{m}$ (1-hot) code (using a simple conventional decoder composed of AND gates only), and then a suitable totally self-checking code translator is used, to formulate a k/2k code, where k is selected such that $2^k \leq \binom{n}{m} \leq \binom{2k}{k}$. The code translator is shown in [10, 2] to be easily implementable using a single level of OR gates only. It is to be noted that this modular technique is not proved to be applicable for every given m/n code; in fact, some problematic codes for which the TSC goal is not achieved are already admitted in [10].

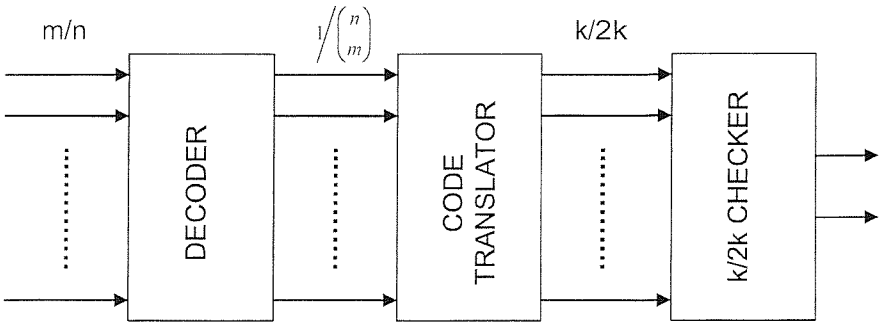


Figure 2.13. m/n checker by Anderson and Metze

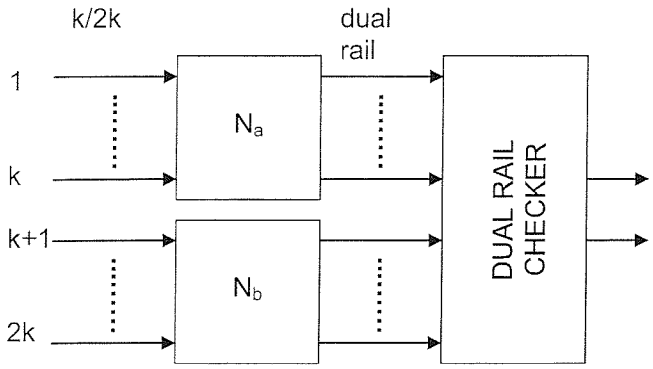


Figure 2.14. k/2k checker by Paschalis et al

This early work, however incomplete, clearly showed the importance of k/2k and 1-hot checkers, not only for the purpose of checking corresponding codes, but also in order to be used as building blocks for generic m/n checkers. Paschalis et al [24] presented an alternative modular design for a TSC k/2k checker, shown in Figure 2.14. The input signal is once again divided into two signals of equal widths; this time, however, subcircuits N_a and N_b are used instead of majority functions. These subcircuits produce m-bit wide outputs, where $m=\lceil \log k+1 \rceil$, that correspond to the binary representation of the number of 1s in their inputs, augmented by suitably calculated constants, so as to be complementary. These complementary signals are subsequently checked by an m-bit *dual-rail checker*. Dual-rail checkers are covered together with the dual-rail code in §2.2.2.2; for the time being, it is enough to mention that such a checker provides the fault-free indication if its input vectors are complementary, and signals an error otherwise. Further, the implementation of N_a and N_b, and the proper calculation of the mentioned constants, are discussed in detail in [24]; interestingly, the subcircuits are composed of full-adder and half-adder cells only. Tables comparing the implementations of [24] to these of [10] are also available in [24]; from them, it is obvious that the most efficient implementation strongly relies on the value of k. In principle, however, the adder-based approach becomes more and more hardware-efficient as k grows [25]. The work of [24] is further continued in [26], [27] and [25], where it is shown that the same or a similar technique can be used to design some (but still *not all*) m/n checkers with $n \neq 2m$, and sufficient conditions that m and n have to satisfy in order for this to be possible are derived. In principle, m always needs to be within a narrow range around n/2, in order for the checker design to be TSC.

The above presented works cover the issue (and reveal the limitations) of m/n checkers using logic gates as building blocks. Kavousianos et al [28] investigate the design of m/n checkers based on CMOS transistors. They ultimately propose the design shown in Figure 2.15. This design consists of two almost identical *m/m+1 programmable weight threshold circuits* L_0 and L_1, producing the checker primary output 2-bit word Q_0Q_1. Each one of

these subcircuits comprises a pull-up part of two PMOS transistors pm_m and pm_{m+1}, and a pull-down part of n NMOS transistors, nm_1-nm_n. $X \equiv X_1$-X_n is the checker input, while I is a control input signal. [28] shows that for suitable values of transistor sizes (given as functions of m), and for I=0, the pull-down part of L_1 drives Q_1 to a "strong 0", that prevails over the 1 that the pull-up part attempts to drive Q_1 to, *only* if the number of 1s in X is greater than m. Due to the inverter, pm_{m+1} has no effect in L_0, and Q_0 is driven to 0 only if the number of 1s in X is greater than *or equal to* m. Q_1 and Q_0 are therefore complementary *only* if the number of 1s in X is *exactly* equal to m, thus providing the fault-free indication. The operation can be analysed similarly and similar conclusions can be drawn when I=1. The authors further prove the TSC property of their checker, which is notably utilisable for arbitrary practical values of n and m, but has the limitation that it is totally technology-specific, therefore unsuitable when a high-level of abstraction design flow is adopted, or when independence of technology is desired.

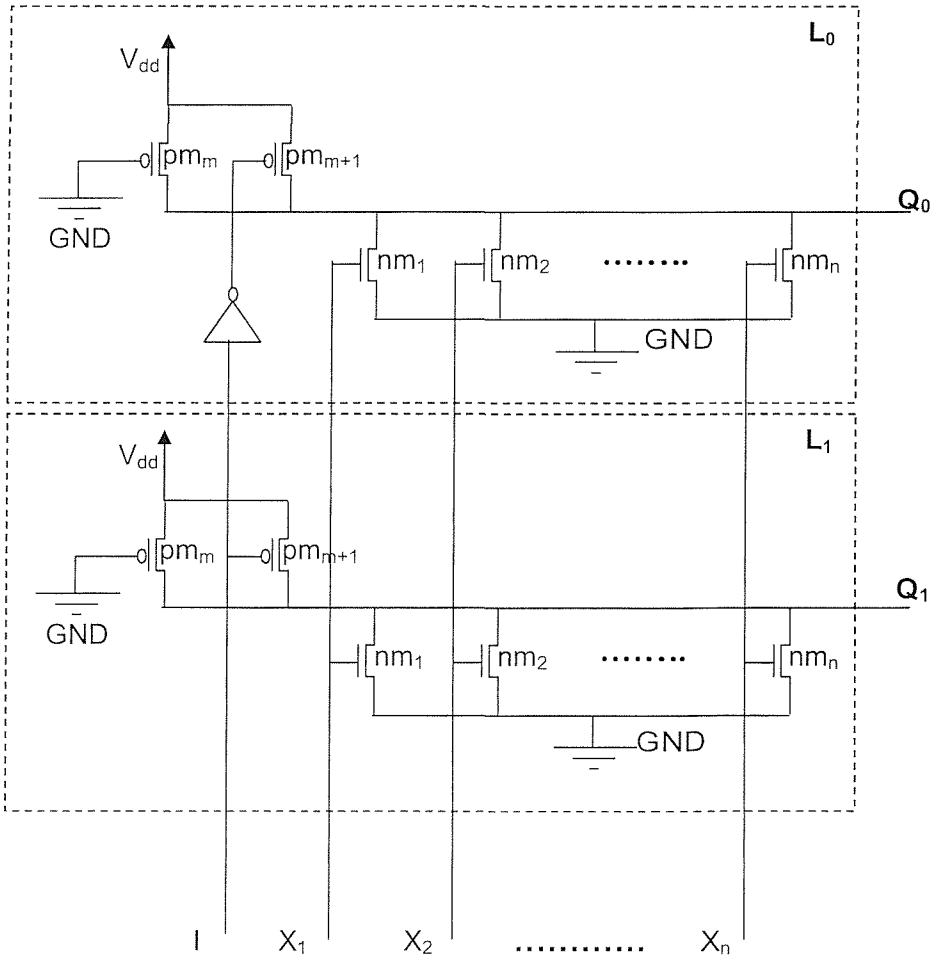


Figure 2.15. CMOS m/n checker by Kavousianos et al

Let us now focus exclusively on the design of TSC checkers for 1-hot codes. Such checkers can be used as building blocks for the design of generic m/n checkers (according to Figure 2.13). Further, as already mentioned, 1-hot codes often appear naturally in computer systems, and are therefore of particular importance.

A couple of choices for 1-hot code checkers have already been covered in this section. Firstly, Anderson and Metze's scheme (Figure 2.13, [10]) suggests that a code translator followed by a k/2k checker, implemented in any of the ways proposed in [10, 2, 24, 26, 27, 25] and mentioned here earlier, can serve this purpose. Secondly, the n/m CMOS checker of [28] is also utilisable for n=1. A third alternative is presented in [29] by Khakbaz, covered in [2] by Lala, and depicted here in Figure 2.16. The *n*-bit wide 1-hot code is first translated to a dual-rail code, consisting of $p=\lceil \log_2 n \rceil$ pairs of complementary bits. Subsequently a dual-rail checker (§2.2.2.2) produces the fault-detection or fault-free indication. The code translator is systematically implemented as follows :

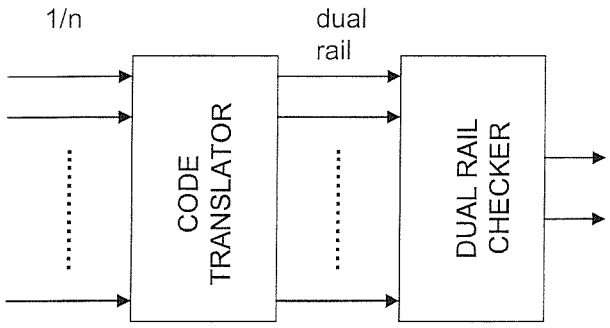


Figure 2.16. 1/n checker by Khakbaz

- Let x_1-x_n be the 1-hot encoded inputs to the checker. Further, let (J_1,K_1)-(J_p,K_p) be the p pairs of complementary code translator outputs.
- Consider the p-bit binary representation of all integers between 1 and *n* (inclusive).

The translator output pairs are produced by NOR gates, where input x_i is connected to the gate producing output J_j, if the binary representation of integer i has a "1" in bit position (p-j). Conversely, input x_i is connected to the gate producing output K_j, if the binary representation of integer i has a "0" in bit position (p-j). If I(k) denotes the k-position bit of the binary representation of integer i, the above idea can be formulated as in the following equations :

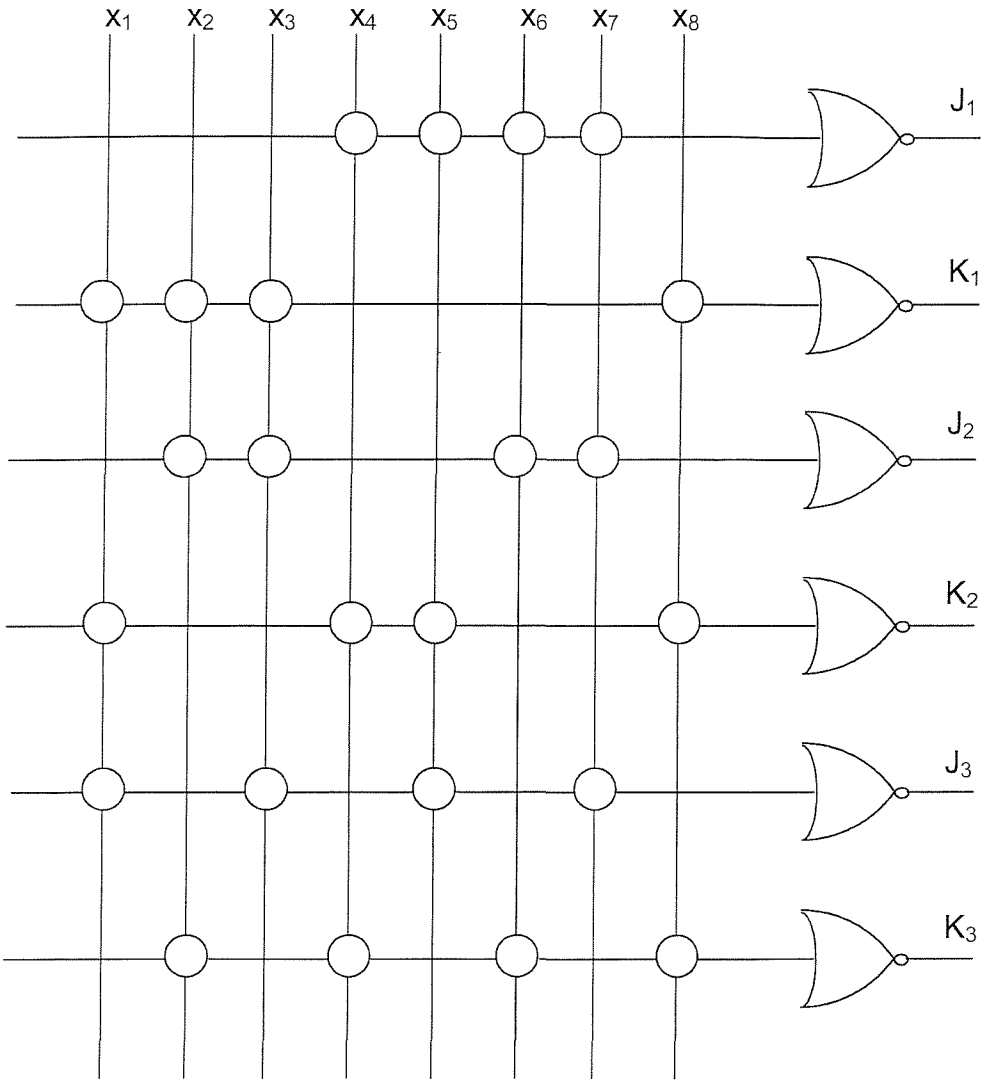


Figure 2.17. 1/8 to 3-pair dual-rail code translator

$$J_j = \overline{\sum_i x_i}\text{, for all i} : I(p-j) = 1 \tag{2.4a}$$

$$K_j = \overline{\sum_i x_i}\text{, for all i} : I(p-j) = 0 \tag{2.4b}$$

The translator construction process is further clarified through the illustrative simple example of Figure 2.17, taken from [2]. The example deals with the translation of a 1/8 code into a ($\lceil \log_2 8 \rceil$) 3-pair dual rail code. Vertical lines represent the x_i inputs, while horizontal lines signify the NOR gate inputs. A bubble where a vertical line meets a horizontal

one signifies a physical connection, i.e. the input corresponding to the vertical line is an input to the gate corresponding to the horizontal line. For example :

$$J_3 = \overline{(x_1 + x_3 + x_5 + x_7)} \tag{2.5}$$

Figure 2.17 is an elegant visualisation of equations (2.4a) and (2.4b). Indeed, consider e.g. input x_3. It is $3_{<10>}=011_{<2>}$, so there are 1s in bit positions "0" and "1". According to the above rule, this means that x_3 will be an input to gates producing J_3 and J_2, and the corresponding connections can be observed in the figure. There is a 0 only in bit position "2", so x_3 contributes to K_1 and again the connection appears in the figure.

In [29], Khakbaz further proves the TSC property for his design of Figure 2.16. This is achieved automatically if the bit width of the 1-hot code is a power of 2, since in this case the dual-rail checker of Figure 2.16 receives all possible code words. Otherwise, an implementation of the p-pair dual-rail checker using a combination of two 2-pair dual-rail

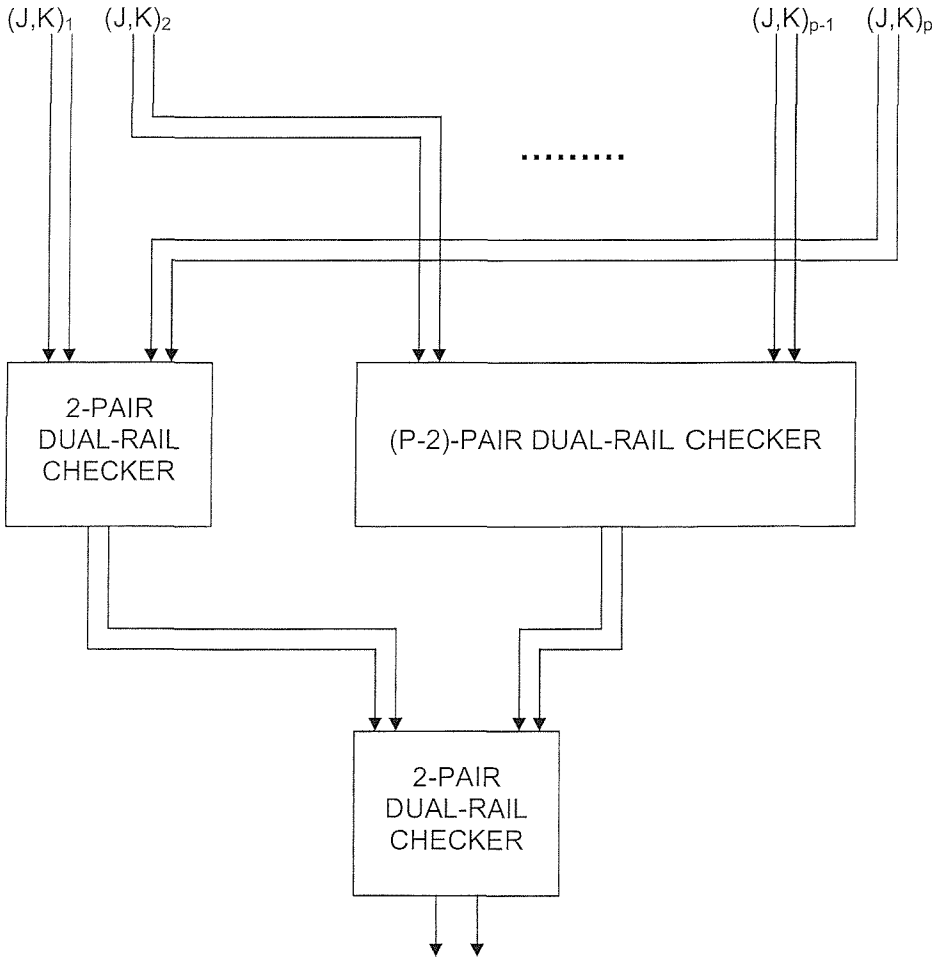


Figure 2.18. TSC dual-rail checker for the design of Figure 2.16, n<>2^p, n<>3

checkers and a (p-2)-pair dual-rail checker is proposed. This implementation is depicted in Figure 2.18 and achieves the TSC goal for any 1/n checker with n>3. $(J,K)_i$ pairs in the figure correspond to the outputs of the translator of Figure 2.17. When the code bit-width is not a power of 2 the dual-rail code is incomplete, and therefore the TSC property for the overall checker is not guaranteed by nature; it is, however, achieved by construction, since as shown in [29] all three constituent checkers of Figure 2.18 separately receive all possible code words during normal operation.

Khakbaz's 1-hot checker design was initially developed to target Programmable Logic Array (PLA) implementations. It is, however, based on elementary logic functions; it can, therefore, be realised in any technology. It is extensively used in this work (chapter 6) for alternative technologies, and that is why particular emphasis has been given to it here.

As the 1-hot bit width grows, equations (2.4) can become significantly long. Depending on the design flow and target technology, that can have serious impacts on the performance of the checker. Tao et al [30] propose yet another choice for the 1-hot checker. They revisit the classical approach of Figure 2.13 (1/n-to-k/2k code translator, followed by a k/2k checker), and propose NOR gate-based design solutions both for the translator and for the checker. Once more, the implementation targets a PLA device, but it can be utilised for other technologies. This technique is reported to experience minimum gate delay; it does not, however, yield totally self-checking solutions for all n. Unfortunately, some practically important values of n are among those not served by it (e.g. 7, 9, 11). Depending on the application at hand, this can be a prohibitive drawback.

Curiously, none of the techniques presented so far can be used to construct a TSC 1-out-of-3 checker. The most generic of them [29], is utilisable for all values of n, *except* n=3. The reason for this, is that the code translator (Figures 2.16, 2.17) in the 1/3 case, produces an incomplete dual-rail code (3 dual-rail code words, 1 missing), which is not enough to guarantee the TSC property for the subsequent dual-rail checker. In fact, it has been proved mathematically that no stand-alone TSC 1/3 checker composed of logic gates can be constructed [14]. This prompted the research community to look for alternative solutions. One such solution [31] considers the 1/3 code in the context of a full-scale self-checking system; it assumes that at least one totally self-checking checker (of any arbitrary code) exists in the system, and combines it with the output of the translator of Figure 2.17,

as shown in Figure 2.19. As the figure shows, the 2-pair output of the code translator is broken in two. The first pair is combined with the (2-bit, §2.2.1) output of the arbitrary code checker through a 2-pair dual-rail checker. The output of this checker is further combined with the second pair through another dual-rail checker, to give the overall error or error-free indication. Both 2-pair dual-rail checkers now receive all possible 4 input code words, so the overall scheme is totally self-checking.

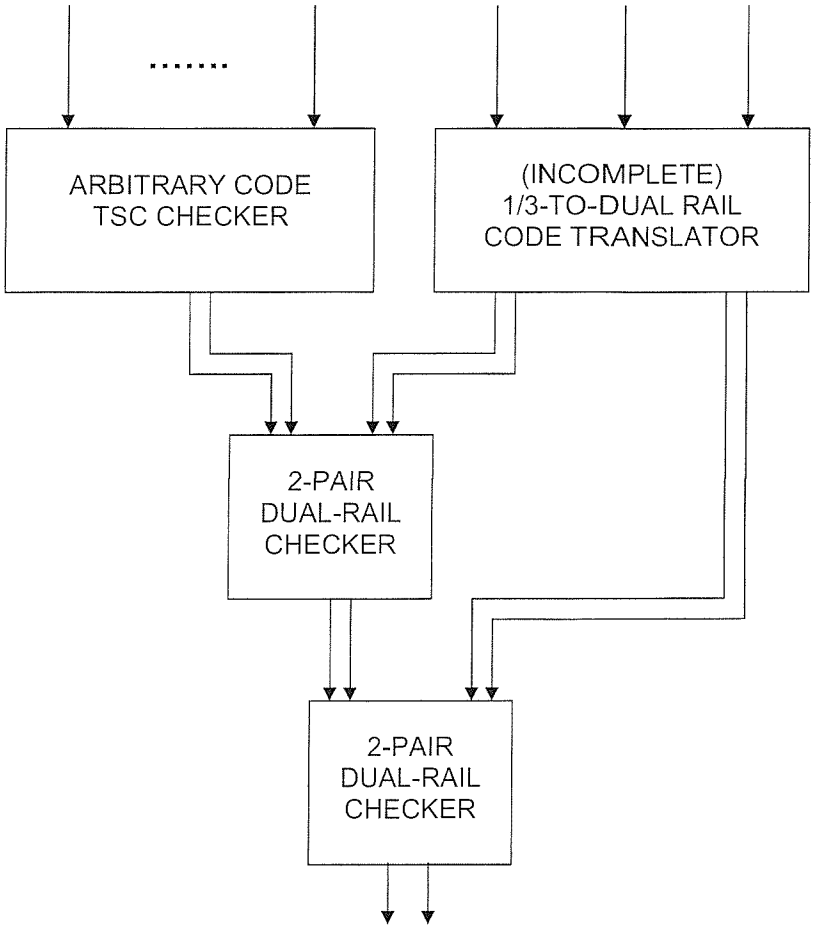


Figure 2.19. A 1/3 code translator combined with an arbitraty TSC checker

Another family of techniques look for transistor-level TSC implementations for the problematic 1/3 code checker. Lo and Thanawastien [32] propose a very compact checker, consisting of 11 NMOS transistors only. The design is only *partially* self-checking (that is, totally self-checking for only a subset of the faults of interest). Metra et al [33] present a generic 1/n TSC checker, utilisable for the 1/3 case, and, like [28], based on threshold circuits (Figure 2.15). Of course, the m/n checker of [28] can in itself be used in the 1/n case, including 1/3.

As a final note on the m/n checker issue, Figure 2.20 shows an "out of the mainstream" sequential configuration that can provide checker functionality for m/n codes. It is based

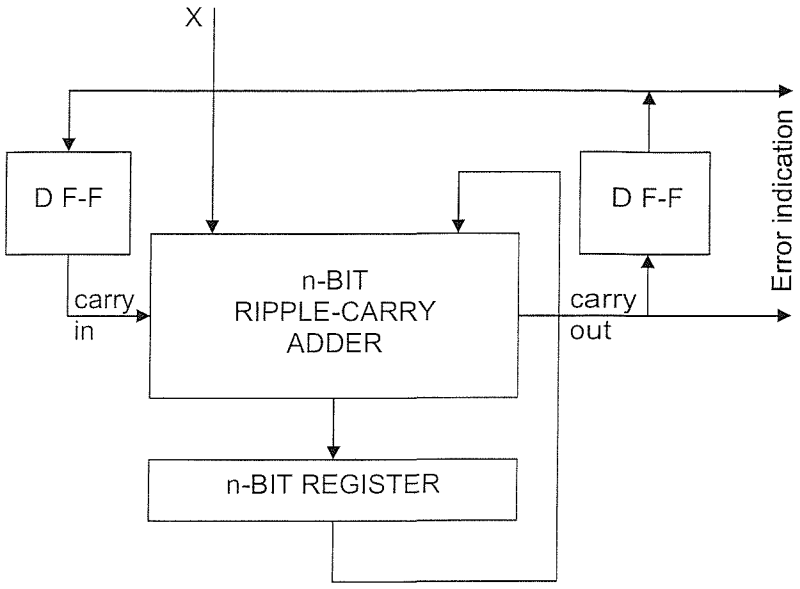


Figure 2.20. Programmable embedded self-testing checker for an m/n code

on the combination of an n-bit ripple-carry adder with an n-bit register, with the register output being fed to one of the adder inputs. Such a configuration is often referred to as an *accumulator*. The m/n encoded signal X is fed to the other input. Two D flip-flops are also used, connected to the adder carry-in and carry-out ports as the figure depicts. The error indication is produced at the carry-out end of the adder as shown. Stroele and Tarnick propose this design in [34], and provide an analytical proof and explanation of its fault detection capabilities, and a description of its properties. Interestingly, the same n-bit design can be used for any given m/n code, provided that the register is initialised with a code word belonging to the code at hand. This property makes it *programmable*. Its main advantage is that it is self-testing by construction as proved in [34]. On the other hand, it sometimes experiences error latency of a few clock cycles (i.e. errors are sometimes detected a few clock cycles after they occur). It is, therefore, not code-disjoint in the strict sense of the term. Unfortunately, error latency increases as the value of n increases; its usefulness is thus restricted to rather low bit-widths. It is also reported [34] that faults can totally escape detection, albeit with a low probability.

The extensive discussion of this section reflects the variety of choices available for checker designs for m/n codes, including the special cases of k/2k, 1/n, and even dedicated pieces of work to address the 1/3 case. Chapter 6 of this thesis provides a critical evaluation of these choices in the context of this present work, and describes the associated implementation and experimental results. As a final remark, further more options for n/m checkers have notably been presented, most of them historical and / or out of the scope of this thesis. These are further covered in [2] and [9].

2.2.1.3 Berger codes

An n-bit word encoded according to a Berger code scheme consists of a k-bit information part I and an r-bit check part I_c, the latter being the binary representation of the number of 1s in the information part (clearly n=k+r). Variations exist, wherein I_c is either the 1's complement of the number of 1s in I, or the number of 0s in I. Without loss of generality these variations are ignored in this discussion. In any case, a Berger code is a separable as well as an unordered code [9]. As already mentioned in §2.2.1.2, it is not always possible to achieve the fault-secure property using unordered codes; Berger codes are no exception to this rule.

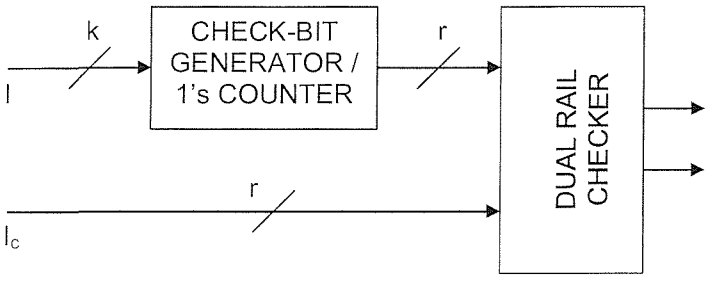


Figure 2.21. A general Berger code checker

A generic implementation of a Berger code checker is shown in Figure 2.21. As is clear from the figure, the information part is fed to a check-bit generator, that effectively reproduces the check part – or typically the complement of the check part, so that a dual-rail checker can subsequently be applied to produce the erroneous or error-free indication. In practice, the check-bit generator is a 1s counter with inverted outputs, composed of full- and half-adder cells only. Issues related to the totally self-checking goal arise here as well, resulting in modified versions of the general scheme of Figure 2.21, often involving suitable constants added both to the output of the check-bit generator and to the check part (analogous to the k/2k checker design of Figure 2.14), or using potentially existing checker outputs (in line with the scheme of Figure 2.19). A recent account of such approaches can be found in

[35], but will not concern this thesis any further. It can also be noted that accumulator-based designs (similar to that of Figure 2.20) have been configured in [34] for Berger codes.

2.2.1.4 Codes based on Hamming distance

The *Hamming distance* of any two n-bit words is the number of bits in which they differ. The Hamming distance concept has been used for error detecting and correcting purposes. In particular, if a code is defined such that any two code words have a *minimum* Hamming distance of d, then it can be shown that this code has the capability to detect d-1 errors, and to correct $\lfloor (d-1)/2 \rfloor$ errors [1]. Note that both even and odd parity codes (§2.2.1.1) are special cases of such codes, with d=2, therefore 1-error detecting and 0-error correcting capability.

When d=3, the widely used, single-error-correcting / double-error-detecting code, often simply called the (conventional) Hamming code can be defined analytically as follows [1]. If there are q information bits, c check bits are needed, where $2^c \geq q+c+1$. The resulting word consists of $(q+c)$ bits and can be represented as $b_{q+c}\ldots b_2 b_1$. Bits b_2^i, $0 \leq i \leq c-1$ are the check bits. Let n be an integer and $b_j(n)$ the value of the j-th bit of n (represented in binary). Let $p_j=\{$(integer) I / $b_j(I)=1\}$, that is p_j is the set of integers whose binary representation has a 1 in position j. Then consider the following c parity-check equations

$$\sum_{k \in p_i} b_k = 0 \,,\ i=1,\ldots,c \tag{2.6}$$

where the summation is modulo 2 (effectively XOR). From these equations, check bits can be determined. For example, consider 4 information bits. It should be c=3. Then equations (2.6) become

$$b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 0 \tag{2.7a}$$

$$b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 0 \tag{2.7b}$$

$$b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0 \tag{2.7c}$$

enabling the calculation of the check bits b_1, b_2 and b_4 from the information bits b_3, b_5, b_6 and b_7. This example conveniently demonstrates how error correction works. Indeed, consider a single erroneous bit, e.g. b_6. Equations (2.7b) and (2.7c) will now necessarily yield logic 1s. Observe that the outputs of equations (2.7), from (2.7c) to (2.7a), now form the

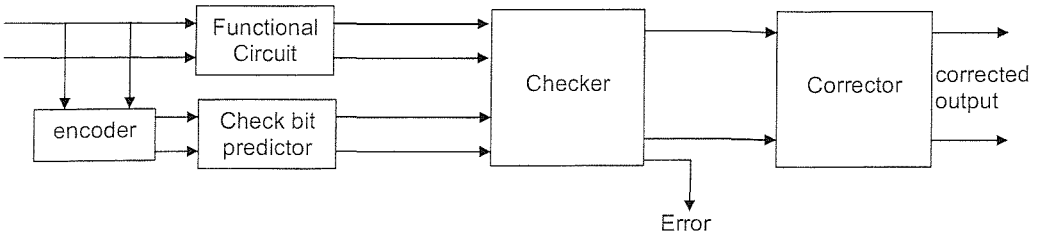


Figure 2.22. Application of an error correcting code.

binary number 110, which corresponds to the decimal number 6, which is the subscript of the erroneous bit b_6. The block diagram of Figure 2.22 shows how error correction is realised. The encoder block effectively realises equations (2.7) and produces the input check bits, while the functional circuit is supplemented by an output check bit predictor block, similar to the parity prediction schemes discussed in §2.2.1.1. The checker effectively again just implements equations (2.7), while the corrector interprets the checker information to determine and invert the faulty bit. The checker also produces an error indication. The only complicated block in the figure is the check bit predictor, which is realisable only when the check bits of the functional result can be calculated from the check bits of the function operands. This is not always achievable; in practice, the code is particularly useful when the "functional circuit" is a system bus or a memory array.

In [36], the Hamming code is used to check a memory (SRAM) block. When a write operation is performed, check bits are also computed and stored together with useful data. When a read operation is performed, the stored word is first checked and then the information part is isolated and used. The overall testing scheme is further armed with BIST resources (§2.1.2) that test memory cells by performing read and write operations to cells when they are not accessed for functional purposes.

Another application of Hamming encoding is found in [37]. The next state logic block of a finite state machine is implemented such that the next state signals are encoded according to the Hamming single ECC, and the scheme of Figure 2.22 is subsequently applied to achieve fault tolerance by means of error correction. Interestingly, the whole process has been coded as a pre-processing step in the synthesis process, therefore producing on-line testable designs by automatically modifying the VHDL descriptions, and is reported to be compatible with commercial synthesis tools.

As opposed to the conventional Hamming code referring to minimum Hamming distance between code words, Bolchini et al [38] propose a *constant* Hamming distance code, and use it once more for the purposes of encoding the states of a finite state machine. The constant Hamming distance property is not required of any two random code words, but rather of two code words that correspond to consecutive states. That is, the encodings of any two consecutive states differ by a constant distance *d*, *but* two non-consecutive states do not differ by *d*, but by a multiple of *d*, depending on the number of states that are in between the two states. It is a scheme that does not strictly conform to the usual self-checking design paradigm, in that the sequence of code words is relevant to the encoding. The authors of [38] propose a graph theory-based algorithm to map states to code words, and also use Berger encoding and checking (§2.2.1.3) for the combinational finite state machine output function. Moreover, in [39] the same authors introduce a suitable TSC checker to verify the constant distance between consecutive states. Further details exceed the scope of this thesis; it has to be noted, however, that in contrast to conventional Hamming code, this encoding does not provide error correction. Nevertheless, it detects faults resulting not only in non-code words, but also in incorrect code words, that is, *incorrect* transitions to *legal* states.

The schemes of [38, 37] are efficient for conventional finite state machines, but do not give a satisfactory solution to the controller self-checking problem where the datapath includes storage elements. This issue is revisited and clarified in §6.1.1.

2.2.1.5 Arithmetic codes

The term "arithmetic codes" loosely corresponds to the family of codes whose words preserve the characteristic code invariant property under arithmetic operations. These codes are typically characterised by their *base* integer A. Let the non-coded word be W. Depending on how W and A are combined to produce the encoded word, three categories of such codes are most often reported in the literature [5, 40] :

- *residue codes*

They are separable codes. The information part is the word W itself, while the check part is calculated as (W mod A).

- *inverse residue codes*

They are separable codes too. Again the non-coded word forms the information part, but this time the check part is [A – (W mod A)].

- *AN codes*

They are non-separable. The code words are the products of non-coded words by the base A (W×A).

Clearly different choices of the base A lead to different incarnations of the above classes of codes. As an example, Figure 2.23 shows a self-checking multiplier configuration based on a base A residue code. Of practical interest are the residue codes with $A=2^k-1$, typically referred to as *low-cost* residue codes. In this case, the modulo generators can be implemented relatively cheaply, as trees of *carry end-around adders* (i.e., adders whose "carry-out" signals are connected back to the "carry-in" ports) [5]. The comparator module is implemented based on a dual-rail checker (see §2.2.2).

It is not within the scope of this presentation to give extensive details on arithmetic codes; an interesting application of such codes can however be found in reference [41]. Its authors show that self-checking schemes similar to Figure 2.23 for large multipliers can be cheaper than the corresponding parity prediction schemes of [16, 15], presented here in §2.2.1.1. They further present techniques to choose the most suitable base for various kinds of multipliers and include these techniques and the resulting multiplier designs in a

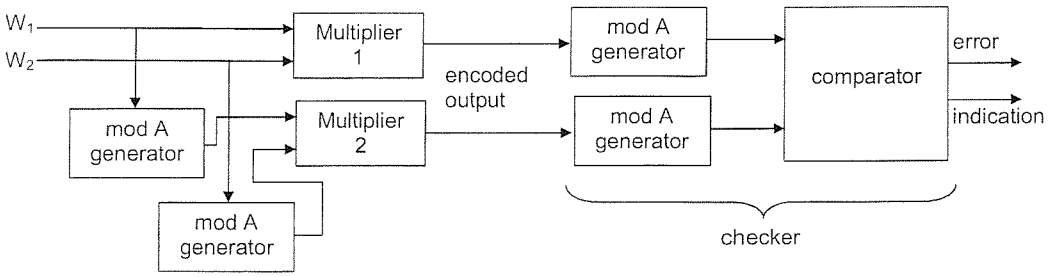


Figure 2.23. A multiplier self-checking scheme based on a base A residue code.

unified CAD tool, which includes the work of [16, 15]. The tool produces HDL descriptions of self-checking data-path modules, which can subsequently be used as building blocks by standard synthesis tools.

Finally, [40] gives the self-exercising checker design solution for low-cost arithmetic codes. Just as in the parity code case [12, 19, 20] presented in §2.2.1.1 (Figure 2.12), a code words generator design (again resembling an LFSR in structure and hardware cost) is

used to provide additional code words to the conventional arithmetic code checker. Alternatively, an accumulator (§2.2.1.2) can be used for code word generation purposes.

2.2.1.6 Sharing on- and off-line testing resources

This subsection focuses on a few approaches that aim at reusing test resources normally employed for off-line BIST (§2.1.2), to provide on-line self-checking functionality as well. The motivation behind such combined approaches is that both of the above families of techniques impose significant hardware overheads to the original designs; having both on a chip can result in prohibitively large cost. Reusing resources in the mentioned manner is an attempt to keep the cost within acceptable limits.

The first successful attempt in this direction has been *Unified Built-In Self-Test* (*UBIST*) [42]. The self-exercising checker design and the overall UBIST scheme proposed therein, are shown in Figures 2.24 and 2.25 respectively.

In Figure 2.24, FIs are functional circuit inputs received by the checker during normal operation (when the control signal T=0). In contrast, when T=1 (test mode), the checker receives inputs from the BILBO register (§2.1.2). The code / non-code indicator specifies if the input word provided by the BILBO is a code or a non-code word. Testing the checker with non-code as well as code words is reported to enhance its self-exercising capabilities. In both the code and the non-code word case, additional logic in Figure 2.24 ensures that the checker outputs f_0 and f_1 will respectively

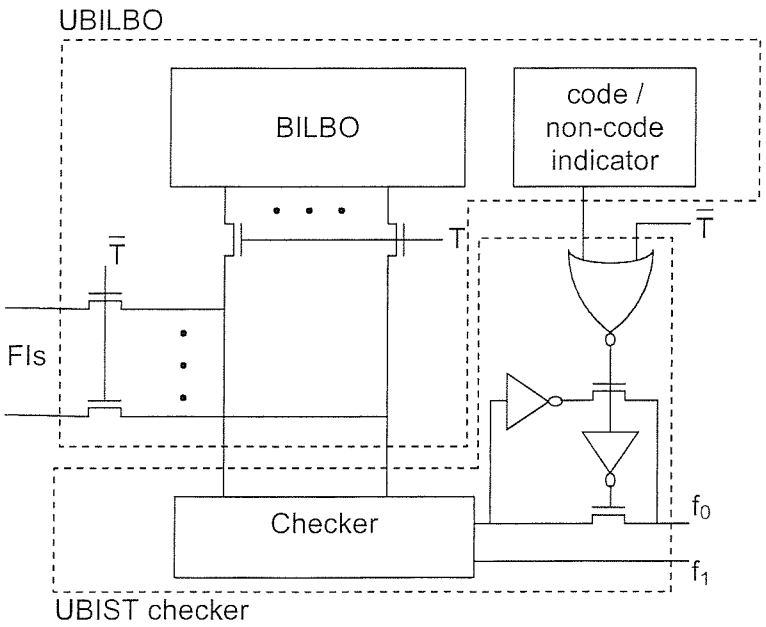


Figure 2.24. A UBILBO and a UBIST checker

provide a fault-free (f_0=~f_1) or faulty (f_0=f_1) indication. Dashed lines in the figure define the *Unified BILBO* (*UBILBO*) block, as the combination of a usual BILBO register with the code / non-code indicator and a few controlling transistors, and the *UBIST checker* as the combination of a normal checker with the additional logic shown.

In Figure 2.25, a part of a circuit configured according to an overall UBIST scheme is shown. Consider the off-line test mode and assume two test sessions, T1 and T2. During T1, odd UBILBOs operate in TPG mode and provide test vectors to odd functional blocks.

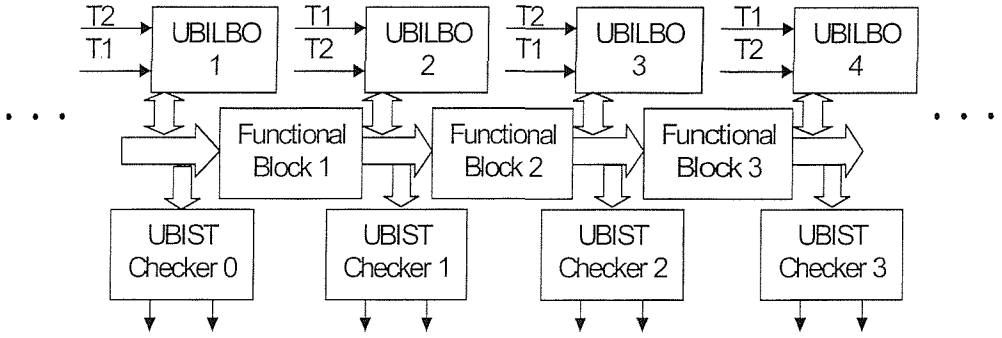


Figure 2.25. The overall UBIST scheme

The responses are compressed by even UBILBOs (in MISR mode), as well as directly verified by odd UBIST checkers. During T2, odd and even blocks mutually exchange roles. During normal operation, BILBOs are isolated from the rest of the circuit (T=0 in Figure 2.24), and functional block outputs are normally checked by the corresponding checker modules, as in conventional self-checking design. It should be noted that the UB-IST technique does not assume a particular error detection code. The designer is free to choose the one that best accommodates his or her needs. BILBO designs that produce code and non-code words for various codes are further included in [42].

A more recent combined off- / on-line testing approach is presented in [43]. The overall

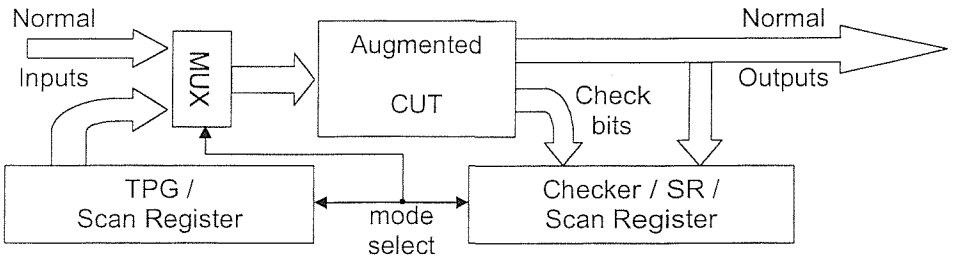


Figure 2.26. A combined on-line / off-line approach

testing scheme is shown in Figure 2.26. The TPG and signature register of the figure can also operate as scan registers, so that the off-line test mode can be realised as either BIST or shift-based testing. In either case, inputs from the TPG / scan register are fed to the augmented CUT and the normal outputs are either compressed in the SR or shifted out. In this off-line mode, the check bits shown in the figure are ignored. During normal operation, clearly normal functional inputs are fed to the circuit. This time, a *check bit generator* residing within the augmented CUT is taken into account. Effectively, the generator is designed such that the check bits it produces, in the fault-free case, equal to the bits residing in the signature register when fed by the given normal output. Their equivalence is then checked using a normal comparator, fed by the check bits and the contents of the SR. The checker is thus composed of the SR, the comparator and some auxiliary logic. Hence, the block labelled "checker / SR / scan register" is a resource shared by off- and on-line testing strategies. The authors of [43] also report a logic synthesis tool that synthesizes the check bit generator, to produce the desired output. As reported in [43], a major disadvantage is high fault latency, i.e. faults are detected on-line a number of clock cycles after they occur. Proposed modifications reduce the latency, but increase hardware overhead, thus cancelling out the benefits of hardware resource reuse.

In another approach [44, 45], the *PMISR* (*Parity-preserving MISR*) is introduced. It receives (n+1)-bit wide even-parity encoded inputs and produces two output signals, r_1 and r_2. In contrast to the usual convention, here it is $r_1=r_2$ if the checker input (CUT output) is fault-free, and $r_1=\sim r_2$ if it is faulty. The structure is a normal MISR with its state bits suitably XORed. With some modifications, it can also be used as a test pattern generator or as a scan register. Thus, BIST or scan-based testing can be configured within an overall design utilising PMISRs, while during normal operation signals r_1 and r_2 from all PMISR structures provide the on-line error indication. This work is extended in [46], to include a generic design methodology for other linear separable codes, thus resulting in a linear *Code-Preserving MISR* (*COPMISR*). This time, the state bits are not XORed. Rather, they are input to a more complex code-specific linear combinational circuit.

2.2.1.7 Other related work

In this subsection, two other interesting pieces of self-checking design related work are presented, that do not fit into any of the above subsections.

In [47], an attempt to automate self-checking design is proposed. Simulatable and synthesizable VHDL [48, 49] descriptions of self-checking design building blocks are presented. These include code word generators and checkers for various error detecting codes. Parity, Hamming and Berger codes are among them. The VHDL feature of parameterised component descriptions, using generic values, is exploited, thus making the descriptions useful for several data-path bit-widths. Two component versions are given for each code, supporting both serial and parallel application of information parts to the code bit generator. The overall system is considered to be supervised by a controlling unit, which receives and handles the error indications. Auxiliary blocks (e.g. special purpose registers) are also presented, to facilitate communication between the controlling unit, the error detection circuits, and the outside world.

Finally, as a supplement to self-checking design, [50] proposes a transient fault tolerance technique, based on *Code Word State Preserving*. This technique augments the functionality of logic blocks receiving encoded inputs, such that the blocks implement their usual operation when fed by a code word, but preserve their previous outputs when fed by non-code words. Clearly the logic blocks have to be augmented to integrate checkers and auxiliary logic within them. They are then said to incorporate *Code Word State Preserving Elements*. Implementations and applications of such elements and resulting logic blocks are discussed in [50]. The technique is effective against transients of short durations, but clearly cannot provide satisfactory fault recovery against permanent faults.

2.2.2 Duplication testing and related schemes

In this subsection, duplication and duplication-related techniques are discussed. Technically, these techniques adhere to the general self-checking scheme of Figure 2.9, and thus fall into the broad category of self-checking design. Therefore the self-checking theory definitions and terminology (§2.2.1) will be used throughout this subsection. However, duplication schemes are addressed separately due to their extensive development and special significance for the purposes of this thesis (chapter 5).

Broadly speaking, duplication techniques adhere to the paradigm of Figure 2.27. The similarities with the general self-checking scheme of Figure 2.9 are evident. Indeed, the func-

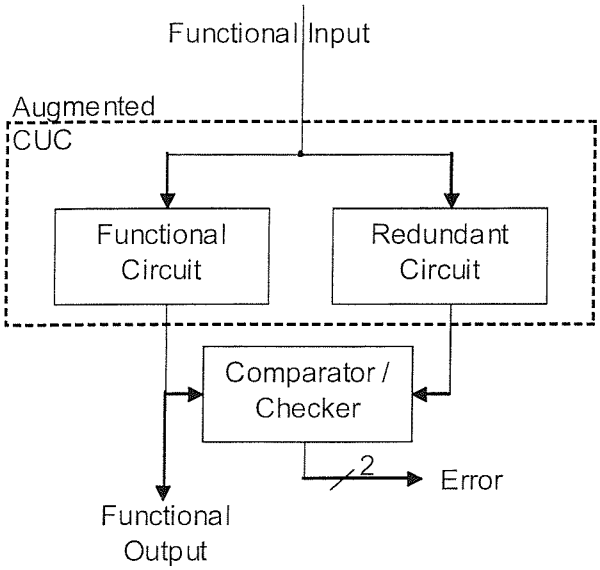


Figure 2.27. Duplication testing

tional circuit is augmented through the addition of a redundant circuit, which can be viewed as a check bit generator. This redundant circuit is fed by the functional input, and may produce either an identical copy of the functional output, or some variation (e.g. the complement of the functional output). In that context, the functional / redundant output pair can be viewed as an "encoded word" in the sense of §2.2.1. A checker module is further used to produce a 2-bit error indication (often simply by comparing the functional and redundant outputs). The totally self-checking properties of Definitions 2.3 and 2.5 are once more desired for the augmented Circuit Under Check (CUC) and the checker respectively. Variations of duplication testing are defined with respect to what exactly the structure and functionality of the redundant circuit are, and whether it is physically introduced or its operation implemented by pre-existing idle functional resources. Other variations do not fully follow the paradigm of Figure 2.27. Indeed, there can be cases where the flow of data through the functional and redundant circuits follows different paths, or where the functional input is first somehow processed (e.g. shifted) before being fed to the redundant circuit. All these techniques share the common property that the size of the redundant circuit is of the same order as the functional circuit (as opposed, e.g. to parity prediction normally using much less hardware than most functional circuits), and the redundant output is typically of the same bit width as the functional output (once more, as opposed for instance to a parity scheme always needing a single additional bit regardless of the functional output bit width). These common characteristics loosely outline the family of duplication-related techniques addressed herein.

2.2.2.1 Physical duplication

In the basic physical duplication checking scheme, the redundant circuit of Figure 2.27 is a replica of the functional circuit, and it is physically introduced together with a comparator.

The scheme is fault secure by nature [9]. Indeed, a fault in the augmented CUC may affect either the functional or the redundant circuit, thus potentially corrupting bits in either the functional or the redundant part of the "encoded" output (*never* both). Any fault that reaches the encoded output will thus create a non-code word (i.e., an output word whose functional and redundant parts are unequal). The comparator module also has to be fault-secure; as a matter of fact, fault-secure comparators are implemented by inverting all bits of one of the inputs, and then introducing a dual-rail checker (to be presented in detail in §2.2.2.2). Further, the hardware overhead associated with physical duplication clearly exceeds 100%.

Physical duplication as explained above is also referred to as *identical* duplication, assuming that the functional and redundant circuit are structurally equivalent. Although very robust against single faults, identical duplication can be problematic in cases where double faults are expected to develop in the system, such that the functional and redundant circuits demonstrate the *same* faulty behaviour (*common-mode* faults). An alternative to identical physical duplication, is to introduce a redundant circuit that is *functionally* equivalent, but *structurally* diverse to the functional circuit, thus implementing *diverse* duplication [51, 52, 53].

In [51], Mitra and McCluskey perform fault simulations on a number of benchmark logic circuits, to compare various self-checking techniques, including diverse and identical duplication, parity prediction (§2.2.1.1) and Berger codes (§2.2.1.3) against multiple faults, and against double common mode faults. Their results are strongly in favour of diverse duplication. The work also includes comparisons in terms of hardware overhead. Interestingly, in many considered examples, Berger code self-checking is more expensive than duplication, due to the complexity of Berger code prediction logic and Berger code checkers.

In [52], Mitra et al once more compare identical and diverse duplication with respect to their vulnerability to double faults and once more establish the increased detection property of diverse duplication through fault simulations. They also provide a theoretical approach to the issue, through the introduction of *self-testable* fault pairs :
Definition 2.13 : A duplication scheme is *self-testing* with respect to a fault pair (f_1, f_2), where f_1 affects the functional and f_2 affects the redundant circuit, if there exists a func-

tional input for which the two circuits produce different outputs under the presence of the faults. (f_1, f_2) is then called a *self-testable* fault pair.
Notice the analogy between the self-testing property considering *single* faults, for *any* self-checking technique (Definition 2.1), and the self-testing property considering *double* faults, as defined above *especially* for duplication schemes. In [52], Mitra et al further present a simulation-based algorithm to identify non self-testable fault pairs in any given duplex system, and propose test point insertion (§2.1.1) to detect such faults, by periodically applying suitable test vectors to the circuit, when it is idle or temporarily taken off-line.

In [53], Mitra and McCluskey further support their work of [51, 52], by presenting a logic synthesis for diversity technique. The technique is fed by a truth table describing the desired functionality, together with a given implementation, and produces the redundant implementation that demonstrates the maximum diversity with respect to the given one, also trying to minimise the area overhead. For this purpose, they quantify diversity as follows :
Definition 2.14 : Given two combinational realisations of the same functionality, the *diversity* $d_{i,j}$ with respect to the fault pair (f_i, f_j) is the probability that the two realisations do *not* produce identical faulty outputs under the presence of the fault pair.
Assuming that all system input vectors are equally probable, Definition 2.14 effectively suggests that the more the inputs that expose a given fault pair, the more diverse the two realisations are, with respect to the particular pair. A unique value for the diversity of the two implementations is computed by calculating the diversity of the implementations with respect to all modelled fault pairs and averaging over the number of pairs. Diversity together with area overhead then define a 2-dimentional design space, which is explored by logic synthesis algorithms also proposed in [53].

An alternative to full hardware duplication is presented in [11]. Only a "sufficiently big" subset of possible faults are targeted, and the redundant circuit this time is a reduced version of the functional circuit, designed such that *only* the targeted faults in the functional circuit can be detected. Input patterns exposing only non-targeted faults are treated as "don't cares" when synthesizing the redundant circuit, thus leading to logic minimisation. Further, the comparator / checker is fed by one or two additional control bits, and equipped with a simple control unit that receives the bits and determines if the checker must check or not, depending on the input word. It thus becomes a *controllable* comparator / checker. Clearly, testability is traded-off for cheaper hardware implementation. The

efficiency of this technique strongly depends on knowing *in advance* the input patterns functional modules are likely to receive, as well as the input patterns functional modules will *not* receive, so that the set of faults that cannot harm the functional output (and therefore do not need to be targeted) can be determined.

2.2.2.2 Dual-rail checking

A variety of diverse duplication is *dual-rail checking* [9]. In a dual-rail design, the redundant circuit of Figure 2.27 does not produce the same output as the functional circuit, but its logic complement (in the fault-free case). A "code word" comprising the functional information part, and a check part of the same size, where every check bit is the complement of the respective information bit, is generically called a *dual-rail* encoded word.

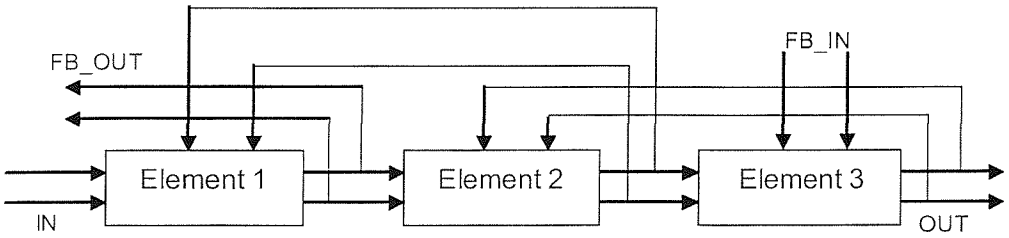


Figure 2.28. The IFIS technique

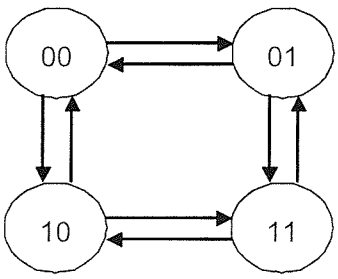


Figure 2.29. Permitted IFIS state transitions

Dual-rail testing following the paradigm of Figure 2.27 is fault-secure by nature, but not widely adopted, since it does not offer any real benefit over physical duplication. Nevertheless, techniques employing dual-rail encoded datapaths have been presented. An example is the *IFIS* (if It Fails It Stops) scheme [54, 55, 56, 57]. Figure 2.28 represents a portion of a system designed using this technique. The system is partitioned into *IFIS elements*. Each element corresponds to a fraction of the overall functionality, implemented using dual-rail encoding, comprising both functional and redundant circuits. Each element is thus an augmented version of a normal functional circuit, whose output is twice as wide as the normal output. The elements further include suitable control logic, so that every pair of functional and respective redundant bits experience *exactly one* change in their logic

values every clock cycle. In particular, if the information bit changes due to the circuit functionality, then the redundant bit remains stable and vice versa. Thus, an IFIS element output pair is only allowed to experience any of the transitions shown in Figure 2.29. As also depicted in Figure 2.28, each element receives feedback from its successor and input from its predecessor. If an element demonstrates an illegal transition due to the presence of a fault, then suitable checkers in its successor and predecessor detect the failure and cause the corresponding elements to stabilize their outputs. Thus, the effect of the fault soon propagates and the system operation stops. Apart from the usual input and output ports, a system implementing the IFIS technique also features input and output feedback ports (FB_IN, FB_OUT), to communicate with a master controller. An important contribution of this work is the implementation of an on-line testable UART – the first on-line testable design of some realistic complexity to be presented in the literature. Note that this technique is proposed at the system level, that is, at a higher level of abstraction than the behavioural level that this thesis is particularly concerned with. This means, for example, that every IFIS element of Figure 2.28 is a full, complex, typically sequential circuit (e.g., the receiver and transmitter are both IFIS elements in the mentioned UART implementation).

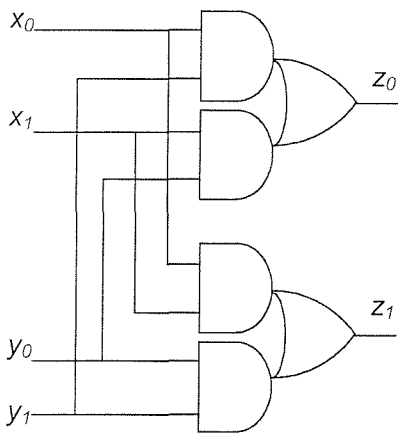


Figure 2.30. The dual-rail checker cell

Although the scheme of Figure 2.27 is not widely used for dual-rail checking, suitable checkers that verify the dual-rail property of their input signals are conveniently applicable in a variety of situations. These checkers are commonly known as *dual-rail checkers* and implemented using the *dual-rail checker cell* of Figure 2.30 [2, 9]. It can easily be confirmed that when the 2-bit input words (x_0x_1, y_0y_1) are complementary $\left(x_0 = \overline{y_0}, x_1 = \overline{y_1}\right)$, then the output pair z_0z_1 is complementary too, thus providing the fault free indication, according to the usual self-checking convention (§2.2.1), ensuring fault-security. The cell thus effectively acts as a 2-pair dual-rail checker. An n-pair dual-rail checker can now easily be constructed as a tree of n-1 such cells, as Figure 2.31 exemplifies for the 5-pair case. The design of the figure checks the dual-rail property of two 5-bit input signals (5 pairs of com-

plementary bits), and the 2-pair constituent blocks are simply dual-rail cells as of Figure 2.30. Observe how the cell outputs are combined together, exploiting their fault-free complementary property, ultimately leading to the typical 2-bit checker output. Clearly, a dual-rail checker is desired to be totally self-checking (§2.2.1). The analysis of the TSC property of dual-rail checkers is analogous to the analysis followed in the case of parity checkers (§2.2.1.1). This is expected, since parity checking functionality is also provided by tree structures (specifically XOR trees). Once more, the code-disjoint and fault-secure properties are ensured by construction (2-bit output). For the remaining self-testing property, every cell has to receive all four possible 2-pair dual-rail code words (0011, 0110, 1001, 1100), and once again a minimum set of words achieving this can be determined by considering all possible code inputs to the final cell producing the ultimate checker output, and tracing back to the overall checker primary inputs. The number of required code words is, again, *only* four, *regardless* of bit width.

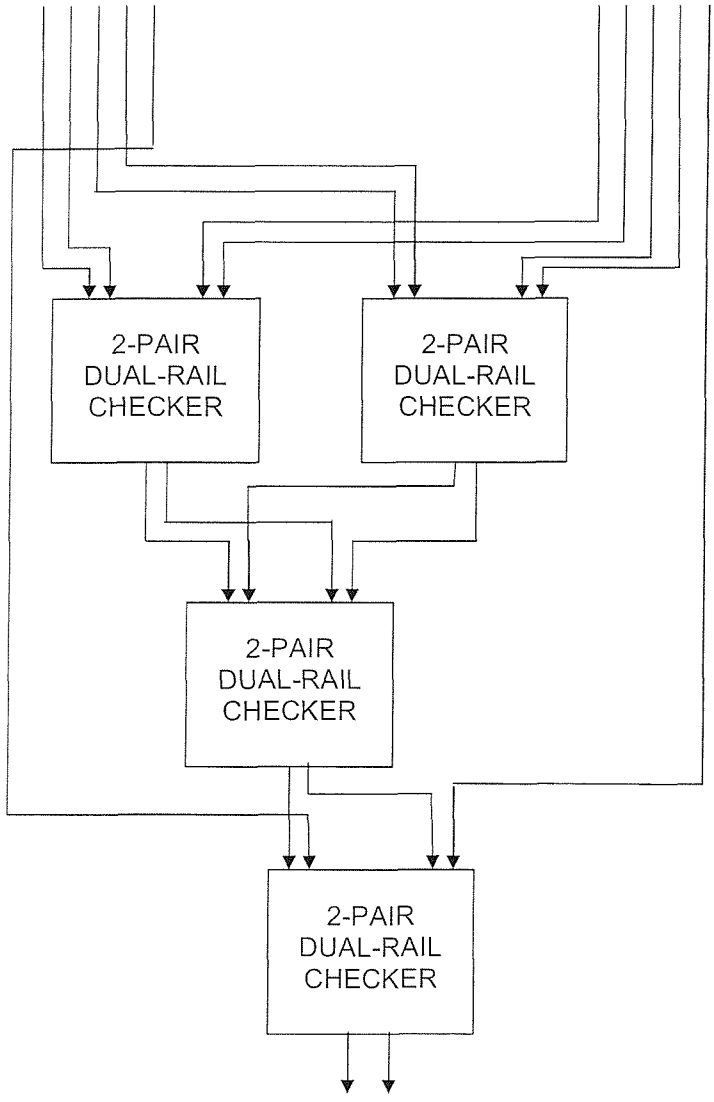


Figure 2.31. A 5-pair dual-rail checker

Further, the following lemma applies [58] :

Lemma 2.3 : Consider a 4×(2×n) Boolean matrix M, whose distinct rows constitute a test set for an n-pair dual-rail checker, composed of 2-pair dual-rail checker cells (Figure 2.30)

only. Then $\forall k \in [1, n]$, the kth and (k+n)th columns of the matrix are bitwise complementary. Moreover, if only the first half of the matrix is considered, by ignoring the (n+1)th, (n+2)th, ..., (2×n)th columns, then two of the four distinct rows of the resulting matrix have even and the other two odd parity, while each column has exactly two 1s and two 0s.

The similarities to the parity-related Lemma 2.2 are evident. Lemma 2.3 implies that if a given configuration requires a dual-rail checker that will receive the rows of a matrix M during normal operation, then there exists *at least one* arrangement of dual-rail checker cells within the overall checker that leads to a TSC realisation. An analytical algorithmic procedure for the extraction of the fastest such realisation (given the matrix M) can be found in [58].

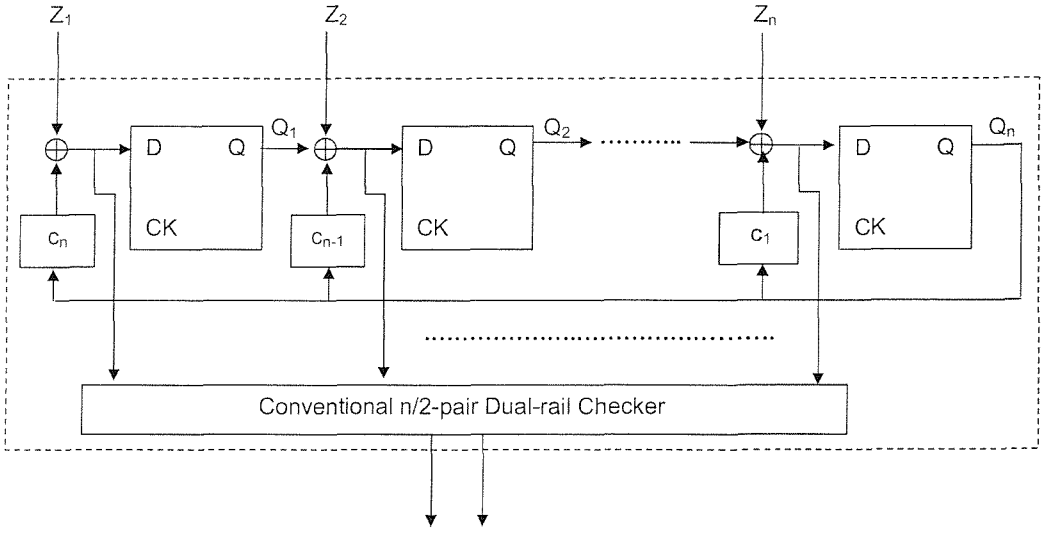


Figure 2.32. n/2-pair embedded TSC dual-rail checker with error memorizing capability

The analogies with parity checkers are further extended in [12, 19, 20], proposing the *embedded dual-rail TSC checker with error-memorizing capability*, to be used whenever the environment is unable to provide the required inputs to the conventional dual-rail checker (Figure 2.32). The design is based on the same theory as its parity-checking counterpart (§2.2.1.1). Consider two n-bit words $W=X_1 \ldots X_{n/2} Y_1 \ldots Y_{n/2}$ and $W'=X'_1 \ldots X'_{n/2} Y'_1 \ldots Y'_{n/2}$. If both words are dual-rail encoded (i.e. $X_i=\sim Y_i$ and $X'_i=\sim Y'_i$ for all i), then elementary Boolean calculus can show that the modulo 2 sum $W \oplus W'$ is *not* dual-rail encoded. Thus, the dual-rail code is not linear. However, if W′ is such that $X'_i=Y'_i$ for all i (duplication encoded), then the result of $W \oplus W'$ can be shown to be a dual-rail word. Therefore, if the

LFSR of Figure 2.32 produces duplication encoded words of bit width n and the n-bit input $Z_1 \ldots Z_n$ is dual-rail encoded, then the conventional dual-rail checker will receive dual-rail words. Further, the n-bit duplication code is cyclic, and $g(x)=x^{n/2}+1$ is a generator polynomial [12]. According to Theorem 2.1, one can construct an LFSR producing all duplication code words of degree n, by choosing a primitive polynomial d(x) of degree n/2 and using g(x)d(x) as the LFSR characteristic polynomial. The resulting checker will be totally self-checking under the sole assumption that the environment provides at least two different dual-rail encoded words [12].

A few applications of dual-rail checkers have been presented in previous subsections, where such checkers were used as building blocks for broader checking schemes. More specifically, a class of m/n checkers (§2.2.1.2), specialised 1/n checkers (§2.2.1.2), Berger code checkers (§2.2.1.3), as well as fault-secure duplication checkers (comparators, §2.2.2.1) all include dual-rail checker blocks. Further, observe that, under the typical convention of §2.2.1, the fault-free response of a checker of any kind constitutes a dual-rail pair (01 or 10). Assuming a complete system with self-checking capabilities attached to several hardware blocks realising the system functionality, the responses from all self-checking blocks should, in the fault-free case, constitute several dual-rail pairs. By combining all these responses and leading the constructed dual-rail word to an appropriate dual-rail checker, a designer can produce a single 2-bit primary output, providing a concise indication of the health of the system [9]. This technique is very popular in self-checking systems, and is often referred to as *self-checking response compaction*. A dual-rail checker employed in that manner is consequently called a *response compactor*.

Another example application of dual-rail checking is presented in [59]. With reference to the paradigm of Figure 2.27, the authors of [59] selectively XOR groups of combinational functional circuit output lines, so that the bit-width of the compacted word reaching the checker is reduced to no more than 5 in all practical cases considered. The redundant circuit is then effectively a *coder*, always producing the complement of the compacted word, and correct operation is verified by a suitable dual-rail checker. Hardware savings are due to the simple structure of the coder, when compared to a redundant circuit that would demonstrate exactly the "complementary" behaviour to the full functional circuit. The reduced bit width of the checker is another source of savings. The authors analyse the functional circuit structure and identify groupings of circuit output lines that minimise the pos-

sibility of fault escapes associated with compaction. On an interesting word of note, this output partition and compaction technique is also shown to be utilisable in an off-line BIST mode, where a MISR structure (§2.1.2) substitutes the dual-rail checker. Overall, this technique demonstrates similarities to the controllable self-checking of [11], in that it trades off testability for hardware savings (by accepting a possibility of fault escapes) and it requires that the functional circuit gate-level structure be known.

2.2.2.3 Algorithmic duplication

Straightforward physical duplication and dual-rail self-checking are primarily defined for isolated, usually combinational circuits. Of more interest is the situation of an overall, complex sequential system, typically described by a conceptual algorithm, synthesized using a CAD tool, and composed of several functional building blocks and storage elements, implementing the algorithm. Clearly, such systems can be fully duplicated and their outputs verified according to Figure 2.27; however, this leads to a significant overhead. Alternative approaches try to analyse the system datapath and identify ways to duplicate and check the system *operations* (functionality), without necessarily duplicating all of the system *operators* (hardware modules). This concept outlines *algorithmic duplication* (also called *algorithmic level re-computing*). The family of algorithmic duplication variants are considered in this subsection. The presentation assumes familiarity with the concept of a *data-flow graph* (*DFG*) and will hereafter use such graphs to describe example system functionality. This assumption is reasonable, since the DFG is a well-established and extensively used scheme in the area of hardware design. In any case, a formal definition of the DFG is provided in this thesis in §3.1.1 (Definition 3.2), as part of the presentation of high-level synthesis. Another idea which is important for the purposes of this subsection, is that of modules' *idle time*. At any given time point, a (typically combinational) hardware module, forming a part of a complex system, is said to be *idle*, if it is not fed by useful functional inputs and does not produce any useful output at this particular point. The concepts of idle time and algorithmic duplication, and considerations, benefits and

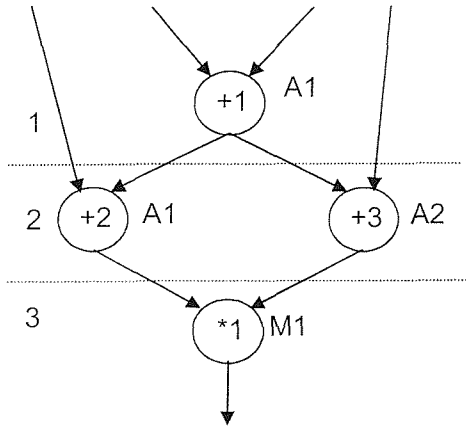


Figure 2.33. Algorithmic duplication motivational example

trade-offs associated with their application are demonstrated in the following, through a motivating example.

Consider the simple but instructive data flow graph of Figure 2.33, defining the functionality of a hypothetical elementary arithmetic chip or part thereof. Operations (additions) +1 and +2 are implemented by module (adder) A1, +3 is implemented by A2 and multiplication *1 is realised by multiplier M1. *Control steps* 1, 2 and 3 define the temporal relationship of these operations. Indeed, in the example, operation +1 is executed a clock cycle before +2 and +3, while the latter are followed by multiplication *1. Arrows in the graph further show data dependencies between operations (e.g., the output of +1 feeds +2). Overall, the realisation of the system functionality requires 2 adders (A1, A2) and 1 multiplier (M1). In line with the introductory comments in the previous paragraph, two copies of the same datapath could be constructed, and the primary outputs of the two copies (i.e. the outputs of multiplication *1 in both cases) could be compared to verify the correct operation. However, in large systems it is often desirable to give a pre-emptive indication of the health of the chip, in this context by duplicating and separately comparing all (or a number of) the constituent elementary operations, rather than the overall design. To this end, a feasible option would be to physically duplicate modules A1, A2 and M1, so that whenever an operation is executed by a module, its duplicate is fed by the same inputs and produces (in the fault free case) the same output; this would clearly result in 4 adders, 2 multipliers and 2 comparators. Observe, though, that adder A2 is idle during control steps 1 and 3, while adder A1 is also idle during control step 3. A2 can therefore be employed during control step 1 to duplicate operation +1. Similarly, operations +2 and +3 can be duplicated during control step 3, mapping the duplicates on modules A2 and A1 respectively. This introduces 1 clock cycle *error latency* (a possible error is detected 1 clock cycle after it occurs), but saves hardware, since the duplication of additions does not require the introduction of any new adder. In order to verify operation *1, the only option is to introduce a new multiplier. So, pre-emptive elementary result verification is achieved with only 2 adders, 2 multipliers and the implied 2 comparators, together with some additional multiplexers, registers and interconnect, while 1 clock cycle error latency is introduced to two of the self-checking operations. Also note that, implementing algorithmic duplication as described above leads to fault-secure schemes (provided that fault-secure comparators are used), since the hardware modules realising duplicate operations of +1, +2, +3 and *1 are all different from the modules realising the corresponding functional operations. An addi-

tional advantage would be to map the duplicates to diverse hardware as well (if possible), thus providing protection against common mode fault pairs (§2.2.2.1). In the rest of this subsection, the presentation overviews works that address the concept of applying algorithmic duplication within complex datapaths.

In [60], Orailoglu and Karri combine fault detection based on algorithmic duplication with self-recovery from transient faults. Their approach slightly differs from the paradigm given in the previous paragraph, in that they do not compare the results of every single pair of a functional and a duplicate elementary operation. Rather, they define control steps during which comparison has to take place (*checkpoints*), and at these checkpoints they compare outputs of *chains* of functional operations, with outputs of chains of duplicate operations. A chain of operations in this context refers to all operations that are executed between two consecutive checkpoints and have data dependencies among them (i.e. directly or indirectly connected by arcs in the DFG). Of course, a chain of duplicate operations cannot use any hardware modules already used by the corresponding chain of functional operations. When an error is detected at a checkpoint, the system *rolls back* to the previous checkpoint control step, so that the faulty chains will be recomputed, hoping that the transient fault will have vanished. Of course, the technique is unsuitable for permanent faults. All chains of duplicate operations effectively constitute a *duplicate DFG*. To this end, tasks addressed in [60] include an algorithmic approach to determine the checkpoints, an analytical and ultimately automated way to construct the duplicate DFG and assign hardware modules to operations, as well as the application of arithmetic properties (distributivity, associativity) on the duplicate DFG, demonstrated to lead to hardware savings in appropriate designs. On the same theme, Narasimhan et al [61] particularly focus on evaluating the placement of checkpoints in a design, taking into account resource constraints (i.e. number of available comparators) and timing specifications (i.e. maximum allowed speed degradation due to rollback and recomputation, given the expected duration of transient faults).

Hamilton and Orailoglu [62] present an algorithmic duplication technique to provide online *fault identification*, together with fault detection and recovery. Fault identification refers to identifying the faulty functional module in a datapath producing erroneous results. In line with [60, 61], they also consider chains of operations. Further, a chain and its duplicate are defined to constitute a *track*. Fault detection is provided by comparing the two

outputs from the two chains of a given track, while fault identification is based on functional unit *differentiation*. Given two functional units A and B, unit A is said to be differentiated from unit B if a track exists that utilizes A but not B. Fault identification is achieved when every unit in the system is differentiated from all other units. For instance, consider three addition operations, +1, +2 and +3, and their duplicates +1′, +2′ and +3′. Let A, B and C be functional units capable of realising them (adders). Assume that +1 is implemented by A, +2 by C, +3 by B, +1′ by B, +2′ by A and +3′ by C. Thus, three tracks are formed, namely track 1 corresponding to +1 and +1′ and utilizing units A and B, track 2 corresponding to +2 and +2′ and utilizing A and C, and track 3 corresponding to +3 and +3′ and utilizing B and C. Clearly all three units are differentiated from one another. If track 1 detects a fault then either A or B is faulty. Additionally, if track 2 also signals a fault, then A is identified as faulty. Alternatively, if track 3 fails, then C is determined faulty. The authors of [62] consequently analyse given design DFGs and assign functional and duplicate operations to hardware modules, such that module differentiation is maximized, while hardware and timing constraints are not violated. Track module utilisation information is stored in appropriately inserted storage elements, while additional control logic exploits all track comparison responses to identify any faulty module. When an error occurs and a faulty module is identified, control rolls back to the previous checkpoint (exactly as in [60, 61]) and recomputation takes place; this time, however, all chains utilising the faulty module are disabled. Thus, the technique provides some limited tolerance to permanent faults as well as transient ones. The same work is carried forward in [63], where redundant logic is added, in order to achieve fault-security (Definition 2.2, §2.2.1) and recovery for a greater set of faults in the overall design (i.e. for faults affecting not only the datapath modules implementing the above mentioned tracks, but also the control logic, and the fault identification and recovery units).

In [64], Karri and Iyer present their *Introspection* technique. Similarly to [60, 62, 61], Introspection fully utilises modules' idle times for algorithmic duplication purposes; however, no additional functional modules are introduced in case the idle time is not enough. Instead, the authors of [64] prefer to produce designs with a number of "unchecked" operations. As an illustrative example, let us refer back to Figure 2.33. As explained above, adder A2 can be used to duplicate operation +1, while A1 and A2 can duplicate +3 and +2 respectively, during control step 3. Under Introspection, no new multiplier is introduced, therefore no duplication testing is applied to operation *1 and the resulting design demon-

strates degraded fault detection capabilities. The Introspection hardware overhead is minimal (simply the result of introduced comparators and registers), but the technique can be very inefficient as far as fault detection is concerned in designs where there is too little idle time. On the other hand, when *too much* idle time is available, the authors of [64] propose exploiting it to implement fault identification, effectively by assigning the same computation to three different modules. Indeed, a pair-wise comparison of module outputs is then enough to identify the faulty one. Clearly the usefulness of this technique for either fault detection or identification very much depends on the considered design.

Lakshminarayana et al [23] revisit the problem of defining and synthesizing a duplicate DFG. In previous approaches [60, 62, 61], pairs of functional and duplicate operations or chains of operations were not allowed to share any hardware modules. The particular novelty of [23] is an analysis of the probability of fault escapes (*aliasing*) if some limited degree of such hardware sharing is allowed, in any given functional and duplicate DFG, for any candidate sharing scenario. Based on this analysis, its authors accept the sharing if the said probability is below a defined threshold. Their starting point is a purely physically duplicated system, where checking takes place at the primary outputs only. However, they perform judicious intermediate result checking, having observed that such checking can minimise the fault escape probability and promote hardware sharing. They further propose parity checking (§2.2.1.1) as a solution to the control path self-checking problem, without, however, paying any attention to the self-testing property (Definition 2.1).

Another alternative is provided in [65], in the form of *semiconcurrent* error detection. In this technique, no intermediate operations (e.g. "+2" in Figure 2.33) are checked. Primary outputs are not *always* checked either; rather, primary outputs are only checked once every P executions of the functional circuit, where P is an integer value (*checking periodicity*). If the functional DFG takes k clock cycles, then the duplicate needs to be constrained within $P \times k$ clock cycles. Typically $P>1$, which leads to a very relaxed time constraint for the duplicate DFG, effectively allowing for area savings through hardware sharing between the original and the duplicate operations (as in [23]). The area / checking periodicity trade-offs are investigated, through the implementation of alternative design solutions, for different values of P. Increased error latency is an obvious disadvantage of this approach.

In [66, 67, 68], Wu and Karri once more address the problem of minimising area overheads and time penalties when employing a duplicate DFG. They partition the functional and duplicate DFGs into several sub-DFGs and compare the sub-DFG outputs / intermediate computation results, together with the primary DFG outputs (an idea already seen in [23]). A novelty in their approach is that they feed selected *original* DFG values to the corresponding *duplicate* DFG operations, having observed that such a rearrangement allows for hardware and clock cycle savings, by breaking data dependencies within the duplicate DFG. In another two publications [69, 70], the same authors reject the idea of a duplicate DFG that is executed in parallel with the functional one; rather, they propose an arrangement in which the original DFG is executed P times, the Pth result is preserved, then recomputation using the duplicate DFG is executed and the resulting outputs are compared against the stored ones to confirm correct operation or produce an error indication. This is reminiscent of the semiconcurrent error detection of [65], in that once again only one every P obtained results is checked; however, in the semiconcurrent case the duplicate DFG is executed *in parallel* to the original, rather than temporarily suspending useful operation. In that sense, [69] and [70] can be classified as *non-concurrent* error detection approaches. Naturally, all previously mentioned works where every primary output was always checked in parallel to the useful operation [62, 60, 61, 63, 64, 23] offer *concurrent* error detection. Returning to [69], one can note that the emphasis is on assigning duplicate operations to different hardware modules from the respective original ones (*allocation diversity*), so as to minimise the possibility of fault escapes (in that sense, it is reminiscent of [23], although the fault analysis is not as thorough). In [70], *data diversity* is also investigated, through *recomputation with shifted operands*. The idea is to keep the same operation-to-operator correspondence between the original and duplicate DFGs, but to do the recomputation having shifted the original input *left* by two bits. The recomputation output is then shifted *right* by two bits, and the result compared against the stored output of the Pth functional computation, as mentioned above. Hardware overheads and fault escape probabilities are also calculated for this technique.

Chapter 5 of this thesis revisits the algorithmic duplication variants presented in the above, evaluates them with respect to their testability characteristics, overheads and synthesis approaches, presents the contribution of this work and outlines comparisons.

2.2.3 On-line BIST and DFT

Having completed the detailed overview of self-checking design, both based on error-detecting codes (§2.2.1) and on duplication-related techniques (§2.2.2), this presentation moves on to *on-line* Built-In Self-Test and Design For Testability. The difference between (externally applied or built-in) *testing* and *self-checking*, is that the former builds up the designer's confidence on the health of a fabricated system through the application of test vectors and collection of test responses, as shown in §2.1, while the latter provides an *on-going* verification of obtained results. It should be made clear that in that sense they are fundamentally different reliability approaches. Testing is typically an off-line operation, as §2.1 showed; the application of test vectors is either done once (*production test*) or by periodically taking the system off-line for testing purposes (*periodic BIST*). The topic of this subsection then, is a presentation of "test vector-based" testing, that, in contrast to what applies typically, *does not* require the system to be taken off-line. Moreover, the following material should not be confused with the works presented in §2.2.1.6, regarding shared resources for self-checking and off-line testing. In the schemes of §2.2.1.6, the system was purely self-checking when on-line – in contrast to the approaches of this subsection that apply test vectors when on-line.

2.2.3.1 Concurrent testing

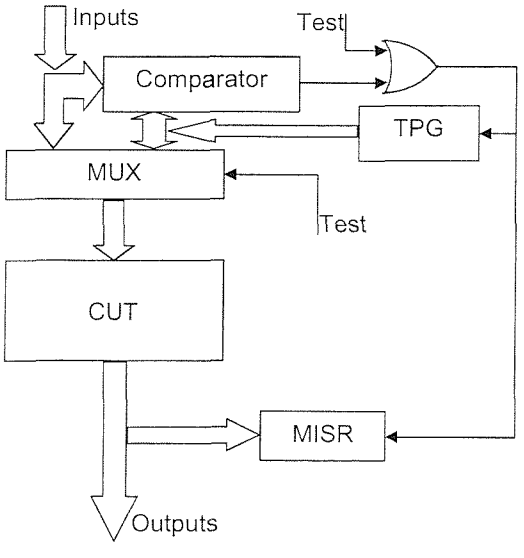


Figure 2.34. CBIST

A historical approach to on-line BIST was proposed by Saluja et al [71]. The *Concurrent Built-In Self-Test* (CBIST) configuration which they presented is shown in Figure 2.34, consisting of the circuit under test CUT, a comparator, a multiplexer MUX, and two typical BIST resources (a TPG and an MISR, see §2.1). In off-line test mode (when signal Test=1), the CUT receives inputs from the TPG and feeds them to the MISR, just as in any normal BIST configuration. In on-line mode, the TPG contents and the functional

inputs are compared. In case they are equal, the MISR compresses the CUT outputs and the TPG advances to the next test pattern. Otherwise, test resources remain idle. Therefore, when the system is on-line, the test resources are active whenever "convenient", depending on the inputs the CUT receives during normal operation. On-line BIST in this context can be conceived as an "extra" feature of the normal operation mode. Obviously, the time required for the TPG to cycle through all states (*test latency*) depends on the functional input data and can be unacceptably high. Still, CBIST is considered a classic approach and it is referenced by several other researchers as probably the very first attempt in this field.

More recently, Santos [72] proposed a similar idea, based on the boundary scan architecture (Figure 2.4). Input test vectors are shifted into input boundary scan cells and functional inputs are compared against them. If they coincide, outputs are collected in boundary scan output cells and shifted out, compacted in a signature register or compared with pre-computed expected outputs. Consequently, the next test vector can be shifted into the boundary scan cells. In order to reduce test latency, not only input test vectors are considered for comparison with functional inputs, but also their complements and vectors resulting by dividing test vectors and complements into two parts, searching for each part individually and considering all possible combinations. For example, if the 4-bit vector 1001 is shifted in, then 0110, 1010 and 0101 are also considered.

2.2.3.2 On-line BIST exploiting idle time

Applying BIST while the system is on-line as presented in §2.2.3.1 has the major disadvantage that the application of a complete test to the CUT can take an unpredictably - and probably unacceptably – long time. Recall the observation of §2.2.2.3, that in realistic systems, combinational modules experience clock cycles during which they do not implement any useful operation (idle time). While §2.2.2.3 showed how such idle cycles can be used for self-checking purposes, the works presented herein investigate the possibility of exploiting these idle cycles to apply test vectors to the idle modules. For example, referring back to Figure 2.33, a test vector can be applied to adder A2 during CS 1. The test response information has to be preserved (using an MISR) or shifted out (assuming a test clock significantly faster than the functional clock), so that the adder can perform its functional operation (addition +3) during CS 2. Afterwards, the test process can resume at the idle CS 3, by the application of the next test vector. When the system primary output is

produced, control normally returns to the first control step; if the idle cycles of a single execution of the DFG are not enough for a whole test set to be applied to the functional modules, then the test is not reset but it is carried on at the next execution of the DFG, effectively spanning multiple repetitions of the normal functionality [73]. When all test vectors have been applied, a fault-free response confirms the results of all the previous functional executions. This idea of applying test vectors to hardware resources when they are not functionally used is exploited in [36] to test memory cells, in combination with a Hamming distance-based ECC (§2.2.1.4). The focus of the present subsection is on datapaths.

From the above description of on-line BIST, it is evident that the more the idle time available in a particular datapath the more efficient the test process. This highlights a difference between exploiting idle cycles for self-checking and exploiting them for BIST. In the former, idle time is only *a benefit*, since it can reduce hardware overheads (§2.2.2.3). In the latter, idle time availability actually *determines the feasibility* of applying the technique in a given design, since too little idle time may result in unacceptably high test latency. Therefore, a major task in on-line BIST is to fit a full test set within as few functional executions as possible. This can be done either by favouring idle time when designing the system of interest, or by reducing the number of required test vectors (*test length*), by using suitable functional modules. Finally, one needs to define a *test schedule* for his or her design, i.e. define the flow of test data through the design, together with the flow of functional data. These crucial issues (idle time availability, test length minimization, test scheduling) are discussed in the following three subsections.

2.2.3.2.1 Idle time availability

The most notable systematic approaches to the analysis of datapaths in search of idle cycles have been presented by Baker et al [74], Brown et al [73] and Williams et al [75]. In [74], all combinational functional blocks in a given design are considered separately, and a *latent profile* is generated for each one of them. A latent profile is a data structure that contains detailed information about the utilisation of modules in clock cycles, i.e. denoting if a functional module is "busy" or idle during a given clock cycle. [73] provides an extension, wherein example designs of substantial size are considered, module latent profiles are extracted, and it is illustrated that idle periods are enough for the application of full

sets of test vectors in many practical cases, of realistically long data-flow graphs. Further, the availability of idle time is identified as a possible *design goal* (as opposed to a design natural property). Loop structures and conditional execution of operations are also briefly discussed, by considering "best" and "worst" case scenarios, corresponding to "as much as possible" and "as little as possible" available idle time. In [75], data-dependent conditional execution is further investigated. Operations that are executed conditionally are assigned execution probabilities; these probabilities are subsequently combined with latent profile information to calculate *test completion probabilities* for functional blocks. Effectively, for every given functional block in an overall design, the work in [75] calculates a probability that a full test can be applied to it, in the potentially available idle time.

2.2.3.2.2 Test length

The term "test length" refers to the number of test vectors that need to be applied to a CUT, so that all modelled CUT internal faults can be detected. Minimising the test length is clearly of particular importance in the context of idle cycles-based on-line BIST. In fact, it is the combination of idle time availability and a test sequence short enough to fit in that idle time, that determine the feasibility of on-line BIST.

Once again, consider the generic testing scheme of Figure 2.1. Assume that the CUT is fed by n inputs. An exhaustive test set for this circuit consists of 2^{n-1} non-zero test patterns (§2.1.2). For large values of n, the exhaustive test length can be prohibitively long. However, the test length can be significantly reduced if *pseudoexhaustive* testing techniques are applied. This involves some form of segmentation of the CUT, through the insertion of redundant logic. A typical approach is to partition the CUT into k segments, where the output of each segment i depends on n_i primary inputs only. Each segment can then be exhaustively tested separately from the rest of the CUT, by 2^{n_i} test vectors. It is often possible to define such a partitioning that $\sum_{i=1}^{k} 2^{n_i} << 2^n$, thus greatly reducing the overall test length. Moreover, it is also possible that different segments can be tested in parallel, leading to further test time reductions. Several segmentation techniques and associated TPG designs for pseudoexhaustive testing can be found in [1].

Furthermore, it is sometimes worth examining circuit functional blocks to check if pseudoexhaustive testing can be applied to them by nature. The following definitions are relevant [1].

Definition 2.15: An *iterative logic array (ILA)* is a circuit composed of identical cells interconnected in a regular pattern (Figure 2.35).

Definition 2.16: An ILA is *C-testable* if it can be pseudoexhaustively tested with a test set whose length does not depend on the number of cells.

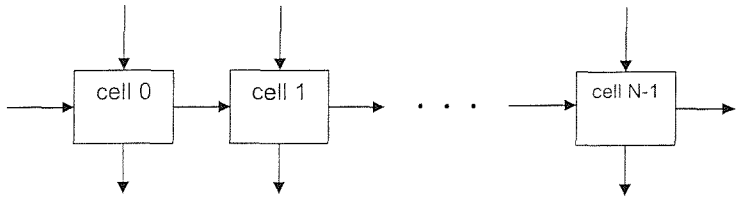


Figure 2.35. An Iterative Logic Array

Clearly C-testability is a very useful property for data path modules, since it limits the test set regardless of the bit-width. A good example of a C-testable ILA of special practicality is the ripple carry adder. This adder consists of a number of full adder cells, connected through their carry-in and carry-out ports, in a fashion that closely resembles the generic ILA structure of Figure 2.35. C-testability then implies that each full adder can separately be tested by its own test set, and also that adder cells can be tested concurrently, thus resulting in a test set whose length is truly independent of the bit-width (i.e. independent of the number of full adder cells).

An alternative concept is presented in [76]. Let us consider the circuit model of Figure 2.36.

Definition 2.17: A circuit C, as of Figure 2.36, is defined as *scalable* if its output function Z(n) is independent of the number n of its input data buses.

Most data path modules normally implement a function of the form Z(A(n),B(n)), where $A(n)=A_{n-1}\ldots A_1A_0$ and $B(n)=B_{n-1}\ldots B_1B_0$. In the formulation of Figure 2.36, $D_i=(A_i,B_i)$, i=0,…,n-1 and w=2, u=n and a control bus K of bit-width v may or may not be present. In the sense of Definition 2.17, such modules can be considered scalable, since their function

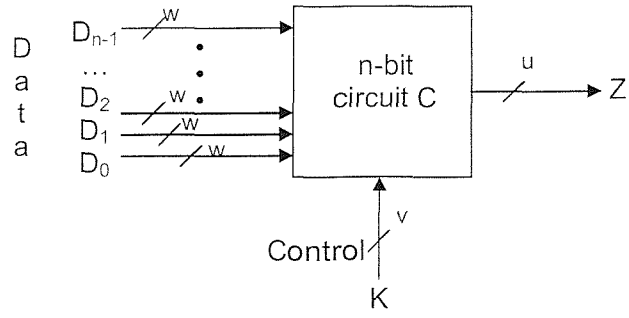


Figure 2.36. General Scalable Circuit

Z(n) (e.g. addition, multiplication, shift) is independent of n. It is interesting that scalability is a broader concept than C-testability. Indeed, the authors of [76] prove that a ripple carry adder is both C-testable and scalable, while a carry look-ahead adder is scalable but not C-testable. They further demonstrate that scalable circuits can be tested by very compact test sets, and they derive analytical test sets and test generator structures for a number of example scalable circuits.

Reductions in the length of test sets are also reported when *functional*, as opposed to structural, fault models are used. An example is the *mutation testing* technique of [77]. Mutation testing originates from the software testing domain; the authors of [77] apply it to derive functional tests for hardware, having observed the obvious similarities between a piece of software and an HDL-described hardware design. In mutation testing, HDL descriptions are repeated several times, and in every repetition a single functional error is injected (for instance, a "+" operator is substituted by a "-"). These corrupted descriptions are called *mutants* and their erroneous behaviour represents functional faults. Consequently, test vectors are applied (by a simulator) to the correct description and to mutants. When a mutant output differs from the correct output, that mutant is considered to be "killed", in the sense that a vector that detects the modelled fault has been identified. Results presented in [77] show that fault coverage is sufficient, while the time required to determine the test set is much less than that required by exhaustive fault simulation applied to synthesized low-level hardware descriptions.

A technique similar to mutation testing can be found in [78]. This time, HDL specifications are translated into binary decision diagrams (BDDs). Faults are injected in the BDD constructs, rather than in the specification itself. Again, inconsistency between the fault-free and the faulty case determines test vectors. Some additional post-synthesis gate-level simulation is employed here, to uncover faults not detected by the faults injected in the BDD representations.

2.2.3.2.3 Test scheduling for on-line BIST

Having discussed the issues of idle time analysis (§2.2.3.2.1) and techniques for the reduction of the test set length (§2.2.3.2.2), this presentation now focuses on test scheduling for on-line BIST. In other words, assuming that the maximum possible availability of idle

time has been achieved, and that test length reduction techniques have been applied, it is now desirable to identify *when* and *how* test vectors can be applied to the functional modules constituting the overall design, as well as how this can be done concurrently with the functional operation.

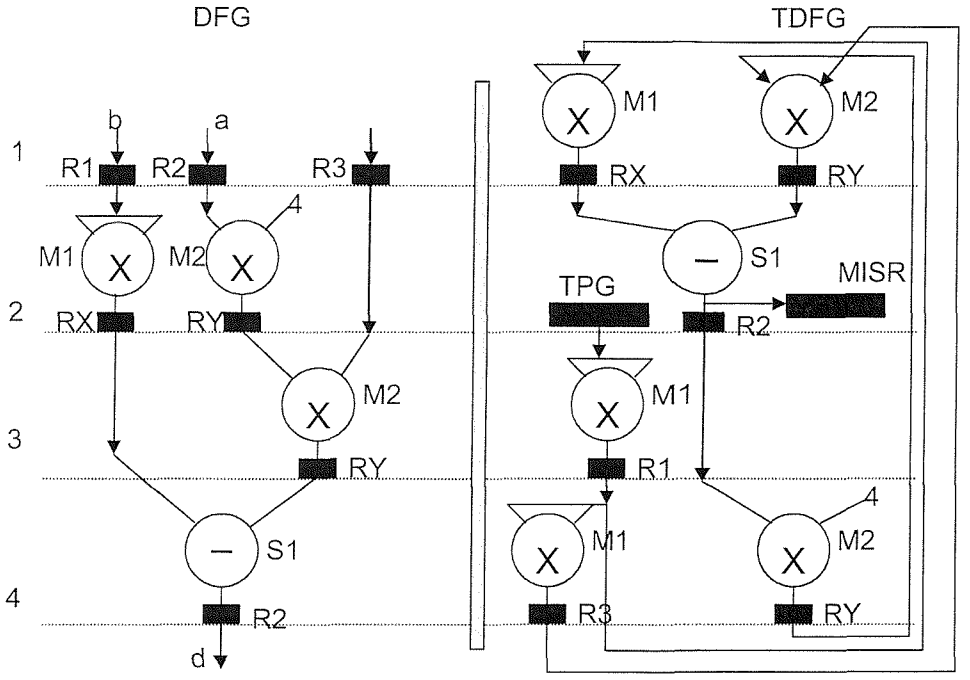


Figure 2.37. Example DFG and TDFG

For this purpose, Singh and Knight [79] propose the *test data-flow graph* (*TDFG*). In Figure 2.37, a DFG and a corresponding TDFG are shown. In both graphs, circles represent operations (exactly as in Figure 2.33), while solid rectangles correspond to registers. The graphs are also annotated with the symbolic names of the hardware units that implement corresponding operations or register loads (e.g. multiplier M1, register RX etc.). It is assumed that control returns to control step 1, after step 4 is finished. From the figure, it is obvious that hardware resources are used in the TDFG during a CS only if they are idle during that CS in the DFG. A dedicated TPG and a dedicated MISR are further introduced. The TPG provides test patterns at CS 3, while the MISR compacts the response of a chain of operations during CS 2 of the subsequent execution. Observe that test data produced by the TPG goes through all system functional blocks and registers before reaching the MISR, thus providing a degree of testing for all system hardware resources. According to the test schedule of Figure 2.37, one test vector is applied to the system for every func-

tional execution. Longer DFGs would allow for more tests per execution. Flottes et al [80] extend this work, by considering data dependent conditional branches in the system. Effectively, each conditional branch is considered separately and small TDFGs are derived for each one of them. In [81], a practical case study of the ideas presented in [79] is given, through the construction of a TDFG for a discrete PID regulator, while in [82] the mutation testing idea (§2.2.3.2.2) is proposed to determine the test set the TPG will provide. A problem with all these TDFG-based techniques is that the quality of test vectors applied to modules can be rather poor. For example, in Figure 2.37 most multiplications in the TDFG are either squarings of the input operand or multiplications by a constant; it can easily be shown that both of these operations cancel out the pseudorandom properties of the vectors the TPG is providing, thus leading to reduced detection capabilities (lower *fault coverage* [1]). The insertion of more than one TPG in the TDFG is mentioned in [80] to partially remedy this weakness.

2.2.3.3 On-line shift-based DFT

As well as BIST, shift-based design for testability has also been proposed in the on-line context. Most of the work done in this field has been carried out by Ismaeel et al [83, 84, 85]. Naal and Simeu [86] presented their own contribution. The underlying principle in all these works is that selected DFG operations are targeted; both their input and output signals are shifted out at "convenient" moments, and the partial result produced by the chip under test is compared against the expected result, produced by external test equipment using the above mentioned shifted-out input signal values. The goal is to test each hardware module using this shift-based technique at least once in a time frame called "pass". The first obvious restriction of this approach is that the chip needs to be constantly monitored by off-chip testing devices on the field.

In [84], idle-time operations are inserted in the DFG and targeted instead of the functional ones implemented by the same hardware modules. This is shown to promote register sharing, thus minimizing the number of signals to be shifted out. In [86], factorisation of complex arithmetic calculations is proposed, in an attempt to minimize the number of hardware modules required and increase idle time, which can in turn be used for redundant operations, again providing opportunities for more efficient signal shifting. In [85], multi-type units (ALUs) and multi-cycled operations are included in the discussion. ALUs are

tested at least once in a pass, for every operation they implement. Finally, [83] is the most comprehensive presentation of this family of techniques. Signal shifting is explained in detail and the use of concurrent testing registers (CTRs) is shown. CTRs are shift registers, where copies of the values to be shifted out are loaded at suitable moments, so that the functional signal registers can be fully devoted to the functional operation, which is thus not disturbed. Additional dedicated control logic provides the interface between the normal operation and the CTRs. The "pass" is formally defined as $\max(NC,N_{test})$, where NC is the total number of steps in the DFG, and N_{test} is the number of steps required to test each module in the design exactly once.

2.2.3.4 Other approaches

The material presented herein concludes the background presentation of digital design for on-line testability. The following two subsections cover generic techniques based on *arithmetic BIST* (§2.2.3.4.1), as well as schemes that are based on the analytical algebraic description of the system functionality (§2.2.3.4.2).

2.2.3.4.1 Arithmetic on-line BIST

Arithmetic BIST is based on the observation that the combination of an arithmetic unit (e.g. an adder) and a register can be used either as a TPG (by adding a constant value to the contents of the register) or as a response compactor (by adding the test response to the contents of the register). The arithmetic unit – register combination is defined as an *arithmetic accumulator*. The accumulator concept has already been encountered earlier in this thesis (§2.2.1.2), where such a structure was used as a building block for a programmable m/n checker (Figure 2.20). Originally exploited in off-line mode, arithmetic accumulators are alternatives to the traditional LFSR-based BIST resources, since their outputs exhibit similar properties to the LFSR outputs. Given enough functional resources that can be combined into accumulators, the hardware overhead introduced by LFSRs can be avoided [87]. There have been several off-line arithmetic BIST techniques in the literature. For the sake of completeness, a few recent ones are briefed here. In [87], the CUT is partitioned into test blocks (consisting of one or more hardware modules) and accumulators are configured around the boundaries of the blocks. LFSRs are introduced only when not enough accumulators can be configured by the hardware resources available. Partitioning is driven

by a cost function integrating hardware overhead and performance penalty introduced by test resources and the problem is formulated into an integer linear programming task. In [88, 89], once more accumulator-based test configurations are presented and the module assignment and module sharing problems are dealt with by a cost function driven heuristic. The cost function incorporates area savings and testability gain, where the testability gain is expressed by the accumulator-forming potential of any hardware assignment or sharing decision.

On another note, Mukherjee et al [90] consider fixed-width multipliers fed by test patterns, and observe that their outputs experience reduced pseudorandom properties, due to the truncation of the least significant part. Reduced randomness results in inadequate testing of modules driven by multipliers (a concept already encountered in this thesis in §2.2.3.2.3). The authors of [90] propose adding the (normally truncated / wasted) least significant part of the multiplier output to the most significant part, when in test mode. Simulations establish the improvement in pseudorandomness of the patterns produced at the multiplier output. Motivated by this work, Gizopoulos et al [91] subsequently propose partitioning a substantially sized circuit into chains comprising one multiplier and one or two adders or subtractors. LFSRs are later inserted to provide BIST functionality to each chain separetely, thus providing acceptable test coverage for the arithmetic modules of the overall circuit.

The previously presented works form the foundation of an arithmetic BIST-based technique particularly named *Versatile BIST (VBIST)*, introduced by Karri and Mukherjee [92]. In VBIST, adders are used for test pattern generation (as in arithmetic BIST) instead of the LFSRs used in [91]. The multiplier - adder chains of [91] are formed, and multiplier outputs have their two halves added together for increased randomness as in [90]. Response compaction finally takes place, again in the arithmetic BIST fashion, using adder-based accumulators. In addition to that, the whole problem is addressed at the HDL level, by modifying the functional descriptions of synthesizable circuits to include VBIST operations. Moreover, testing can be performed either off-line (as in [91, 87, 90, 88, 89]) or *on-line*, during the widely mentioned and in many ways exploited module idle time (hence the versatile property). The technique is suitable for a rather restricted number of designs, namely only those that can be partitioned into the multiplier – adder(s) chains of [91].

2.2.3.4.2 Analytical approaches

This subsection briefs a couple of techniques that especially focus on linear digital systems, manipulate their analytical equations, and derive suitable invariants that are monitored to ensure correct operation.

Bayraktaroglu and Orailoglu [93, 94] deal with digital filters. They start from the digital filter equation

$$y[n] = \sum_{k=0}^{M} h_k x[n-k] \tag{2.8}$$

n in equation (2.8) is a point of (discrete) time, while vectors $y[k]$ and $x[k]$ are the output and input vectors of the filter respectively, h_k is the coefficient matrix denoting the filter functionality, and M is the order of the filter.

After a few steps of algebraic manipulation [94], equation (2.9) is reached

$$\left| I \cdot \sum_{n=0}^{N} x[n] - \sum_{n=0}^{N} y[n] \right| \leq T_{\max} \tag{2.9}$$

where I is an invariant property depending on h_k only, and T_{max} depends on filter coefficients and maximum (expected) input magnitude. Equation (2.9) is the invariant relation that should always hold in the fault-free case, and it is this relation that the filter is constantly monitored against. Two adders and two registers are introduced in the filter realisation to calculate the sums of input and output signals and a checker determines if their difference is within the specified tolerance T_{max}. A fault or the accumulation of minor fault effects is detected when it is not. With the addition of two multiplexers, a designer may reuse the adder – register pairs in the input and output of the filter as arithmetic accumulator-based TPG and MISR for off-line testing purposes.

Another analytical approach can be found in [95]. The authors address linear digital systems in general. Such a system can be described in matrix form as in the following.

$$\begin{aligned} x(t+1) &= A \cdot x(t) + B \cdot u(t) \\ y(t) &= C \cdot x(t) + D \cdot u(t) \end{aligned} \tag{2.10}$$

where $x(t)$, $y(t)$ and $u(t)$ are state, output and input vectors respectively, while A, B, C and D are system parameter matrices.

Manipulating the above *state-space equations*, the authors come up with the equation

$$r(t) = v^T \cdot [Y^{[k]}(t) - H^{[k]} \cdot U^{[k]}(t)] \quad (2.11)$$

$r(t)$ is defined as the system *parity check*. $Y^{[k]}(t)$ and $U^{[k]}(t)$ are vectors comprising the present and k delayed values of the output and input signals respectively. Matrix $H^{[k]}$ is a time-invariant function of A, B, C and D, while vector v^T depends on A and C only.

In the fault free case, the invariant property is $r(t)$=0. On-line testing is performed by synthesizing the system defined by equation (2.11), including it in the overall implementation and monitoring its output $r(t)$.

2.2.4 Analogue electronics related techniques

A few representative analogue electronics-related techniques are briefly mentioned here. The goal of such techniques is to detect a fault by means of its impact on analogue characteristics rather than on logic values. Sometimes faults are detected because analogue characteristics are corrupted at the same time as logic values (e.g. current monitoring, crosstalk effects), while sometimes the effect of a fault on some analogue characteristic enables detection *before* the logic value is corrupted (e.g. delay testing). As chapter 5 will argue, these approaches are not particularly useful for the purposes of the present thesis; the presentation herein is, therefore, very brief, and truly representative rather than exhaustive.

Current monitoring is the most developed of all the techniques in this family. It is based on the concept that most physical defects in VLSI systems result in abnormal current consumption. It can be performed either externally or by embedded *built-in current sensors* (*BICS*) [9]. An application of current monitoring is presented by Bogliolo et al [96]. Fault-tolerant circuits based on *triple modular redundancy* (*TMR*) are considered. Such circuits consist of three copies of the same hardware, followed by *majority voters*, and they offer tolerance against single faults in any of the three replicas, simply by fault masking within the voting hardware. It is, however, desirable not only to mask a single fault, but also to acknowledge its presence, since any faulty situation that leads to permanent damage in any of the three copies will necessarily result in a circuit that is defenceless against any subsequent additional fault. The authors of [96] therefore design a novel majority voter, utilising an embedded current sensor for this purpose. When the sensor detects abnormal current flow, the environment is informed that the circuit has lost its fault tolerant property.

Paschalis et al [97] address *concurrent delay testing*. A delay fault refers to a circuit wire that does eventually take the value of the port driving it, albeit with an unacceptably long delay. Delay faults are regarded as forerunners to logic faults; it is therefore desirable that they be detected as soon as they appear. [97] defines a maximum delay that a wire can experience in order for the situation to be regarded as fault-free (*discrimination time*). A TSC (Definition 2.3) *error indicator* is then shown, receiving a two-bit input and producing a two-bit output. This indicator consists of two 2-pair dual-rail checkers (§2.2.2.2), suitable delay elements and some elementary control logic. It stabilises its output at the fault-free indication whenever a fault-free value ("01" or "10") appears at its inputs. It also does so when the input is fed by a faulty indication ("00" or "11") of duration less than the discrimination time. In contrast, if the faulty input persists, then the error indicator locks its output at the "00" or "11" state, and ignores any subsequent transition of its inputs, until it receives a special "reset" control signal. The structure can be appended to the output of a TSC checker (§2.2, §2.3) of any arbitrary code, enhancing it with concurrent delay testing capabilities.

Favalli and Metra [98] consider *crosstalk faults*, i.e. logic faults that are due to the capacitive coupling between two parallel lines in a system bus. Such faults are sometimes multiple; therefore they are likely not to fall within the detection capabilities of a particular EDC (§2.2). In [98], electrical simulations are conducted for buses encoded according to a number of typical EDCs. The results of these simulations are used to establish analytical expressions for the probability that a crosstalk fault will be detected by the EDC at hand. Motivated by the significantly high fault escape probability of this study, the same authors [99] present a novel, transistor-level detector design, especially tailored to target crosstalk, delay as well as short-lived transient faults. This detector signals an error indication if a transition occurs during the *stability time interval*, i.e. during a specified interval following the rising edge of the system clock.

A few more reliability indicators are briefly mentioned in [9]. These include temperature, voltage, output activity and radiation. None of them is reported to have been widely exploited though.

2.3 Summary

This chapter has given a full and comprehensive presentation of on-line testing theoretical concepts and practical approaches. In the area of self-checking design, various error-detecting codes were reviewed. Such codes provide excellent solutions for the reliability of small-scale, mostly combinational circuits. Regarding large-scale systems, reliability improvements have been attempted through the setting up of on-line BIST, scan-based DFT, and, most usually, algorithmic duplication schemes. Some of the works proposing such schemes included elements of synthesis-related considerations. However, none of them comprehensively addressed all the aspects of automatic large-scale system synthesis within the context of existing synthesis tools. Detailed critical evaluations of these techniques in the synthesis context are needed; these are carried out in chapters 5 and 6 of this thesis, followed by the specific contributions of this research work.

While the overall discussion in this chapter was very broad, particular emphasis was given to elements of on-line testing that are actively exploited in the rest of this research work and are therefore of particular importance for the purposes of the presentation herein. These include :

- self-checking design theory
- parity, m/n and dual-rail checker designs
- algorithmic duplication-based self-checking, for substantially-sized sequential circuits

Chapter 3

High-Level Synthesis

High-level synthesis is addressed in this chapter. The fundamental definitions and concepts are given in section 3.1, while section 3.2 focuses particularly on the specifics of the *Multiple Objective Optimisation in Data and control path Synthesis* (*MOODS*) High-Level Synthesis Suite, which is used in chapters 5 and 6 for all the implementation and experimental results of this thesis. Section 3.3 summarizes the chapter. Only synchronous systems with a single clock are considered in this work, and this will be implied throughout this thesis.

Emphasis throughout this chapter is given to these high-level synthesis elements that are most significant for the purposes of the present thesis. More detailed presentations can be found in two recent dedicated PhD theses, by Williams [8] and Kollig [100].

3.1 Fundamentals

Definition 3.1 : *High-level* (or *behavioural*) synthesis (also referred to as *HLS*) of digital systems is the process of automatically extracting a *structural* realisation of the system from the description of its *behaviour* [8, 100].

Typically, a high-level synthesis system is fed by a behavioural description written in a *hardware description language* (HDL), most commonly the *Very high speed integrated circuits Hardware Description Language* (*VHDL*) [48], although attempts at using other languages such as *SystemC* have also been reported [101]. Note that this behavioural description is limited to an abstract, purely algorithmic representation of the relationship between system inputs and outputs, with no explicit timing or structure information. In fact,

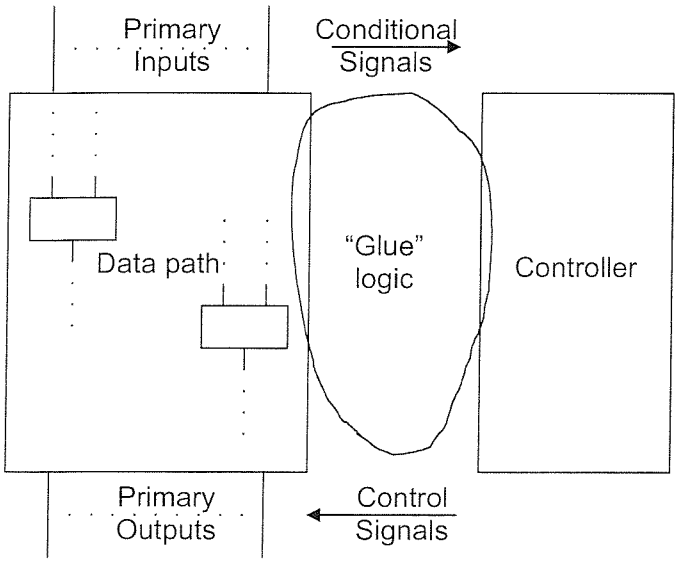


Figure 3.1. Target architecture

it is within the synthesis process itself that such information is derived, and it is included in the high-level synthesis output, which is again in the form of an HDL, albeit at a lower level of abstraction, corresponding to a netlist of components, storage units and interconnect, typically referred to as a *register-transfer level* (RTL) description, suitable for subsequent synthesis by commercially available low-level synthesis tools. This output is graphically depicted in the *controller / datapath* target architecture of Figure 3.1. From the figure, it is evident that the structure of the resulting system consists of *data path* units, implementing the primary input / output behaviour of the system, and a *controller* (or *control path*) part, determining timing issues. These two constituent parts communicate by means of the elementary, gate-level *"glue"* logic. In essence, the controller realises a finite-state machine (FSM), thus providing timing information in the form of *control signals* to the data path. In addition, when the initial behavioural description of the system includes conditional or loop statements, then some of the signals produced by the data path need to be fed to the controller in the form of *conditional signals*, in order for the FSM to correctly produce suitable next-state information.

Figure 3.2 (taken from [8]) captures the typical HLS-based digital system design flow, where the dashed rectangle defines the areas that a generic high-level synthesis tool operates on. The initial behavioural description is compiled, and an intermediate internal representation of the system functionality and structure is formulated. It is on the data structures corresponding to this representation that the system *optimisation algorithms* are applied, taking into account the designer parameter specifications (typically area and delay goals), together with a *technology library*. This library contains parameter information regarding the data and control path units, storage elements and interconnect modules that are used as

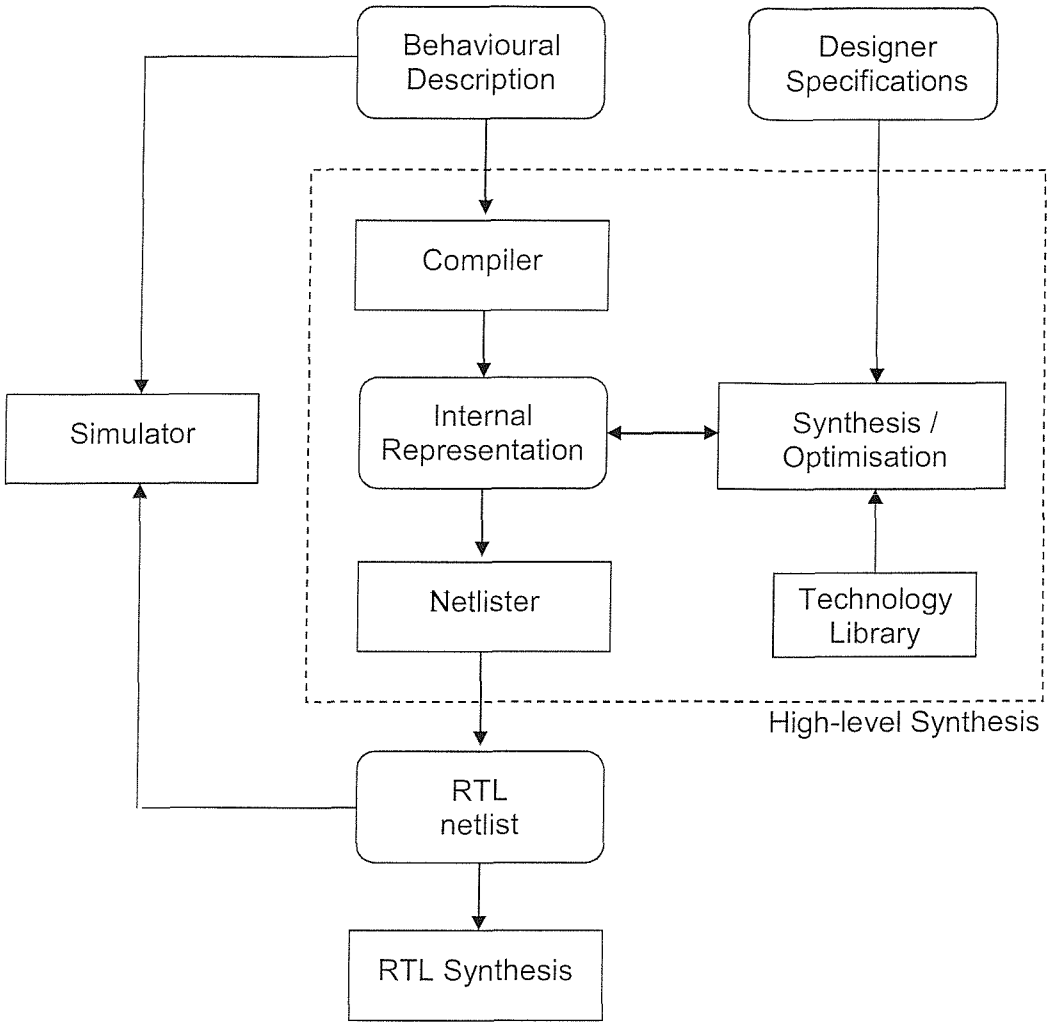


Figure 3.2. HLS-based design flow

building blocks for the realisation of the system. That way, the tool is able to determine the quality of a given realisation of the system at any time, thus providing feedback to the optimisation algorithm. Typically, this feedback greatly influences automatic optimisation decisions. After all optimisation, a back-end netlister produces the HLS RTL output. A *cell library* file (not appearing in the figure) is associated with this output. This file comprises synthesisable RTL HDL descriptions for all the above-mentioned system building blocks. Note the relationship between this cell library and the technology library of Figure 3.2 : the latter comprises *technology-specific* characteristics and properties (in a non-standard, non-HDL format) for the *technology-independent* HDL cells of the former. Finally, it is good design practice to simulate both the behavioural and the RTL descriptions,

in order to verify their equivalence, before feeding the latter to commercial, technology-specific RTL synthesis tools for the actual implementation.

From the above brief presentation, it is obvious that in HLS-based design, the designer's role is limited to providing the algorithmic description of the system functionality, along with his or her specifications; the tool is responsible for the hardware realisation. Clearly, this speeds up the design process tremendously, and minimises the possibilities of a designer error, since describing the functionality of a system is much easier, less time-consuming and less error-prone than designing the actual structure. This highlights fast *time-to-market* as the big advantage of adopting an HLS-based design flow. It is also interesting to observe that the only place in the design flow where target technology is considered, is the technology library. Given that a technology library file is normally a simple add-on to the synthesis system, it can be deduced that the behavioural synthesis process is, in essence, independent of target technology, and it can easily be modified to target alternative technologies, thus offering enhanced opportunities for experimentation.

Clearly, the heart of a high-level synthesis tool is the internal representation of the circuit, and the synthesis algorithms that operate on it. The rest of this section is therefore dedicated to these two elements.

3.1.1 Internal Representation

The internal intermediate form of a given digital system is the product of the behavioural description compilation, sometimes including some source-code level trivial optimisations, and it should be chosen such that it can consistently represent the behaviour and structure of the design. A widely adopted choice for this representation is the *Data Flow Graph (DFG)*. According to De Micheli [102], this graph is formally defined as follows.

Consider a digital system whose overall functionality can be broken down to n_{ops} elementary tasks. These tasks can be logical (e.g. AND, OR), arithmetic (e.g. addition, multiplication), comparisons, or data transfers. These operations are assumed to be fed by one or more inputs, and to produce one or more elementary results.
Definition 3.2 : A data-flow graph $G_d(V,E)$ of a given digital system is a directed graph whose vertex set $V=\{v_i;\ i=1,2,\ldots,\ n_{ops}\}$ corresponds to the set of elementary tasks of the

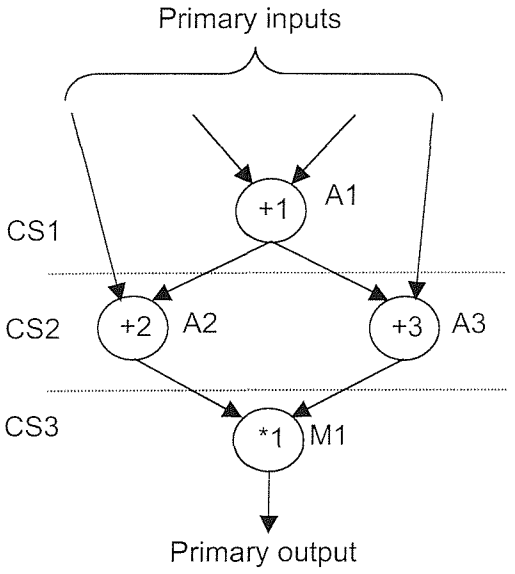


Figure 3.3. An example data-flow graph

system, while the directed edge set $E=\{(v_i,v_j);\ i,j=1,2,\ldots,\ n_{ops}\}$ corresponds to the transfer of data from one operation to another.

An example DFG with only four operations is shown in Figure 3.3. The vertices and edges of definition 3.2 can be observed in the figure. In addition, some common conventional notations and terms can also be defined. In particular, observe that every vertex / operation is assigned a unique symbolic name (conveniently indicating the type of operation the vertex is representing), and annotated with the symbolic name of the data path unit that implements the task. For example, in Figure 3.3, operation (addition) +1 is shown to be implemented by data path unit (adder) A1, multiplication *1 is implemented by multiplier M1 etc. In the prevailing terminology, +1 is *allocated* to A1, *1 to M1 etc. Related to allocation is the concept of *binding* functional modules to particular hardware instantiations, taken from the cell library (§3.2). In the example at hand, +1 is allocated to A1, and then a suitable (adder or ALU) cell is chosen from the cell library and bound to A1, taking its parameters (area, delay etc.) into account. Further, in Figure 3.3 data dependencies between operations can also be observed. For example, +2 has to be executed after +1, since it is fed by its output. Recalling that only synchronous systems are considered, this practically means that +1 needs to be executed one clock cycle before +2, and its result stored in an appropriate storage unit (register). +1 is then said to be *scheduled* one *control step* (*CS*) before +2. Figure 3.3 clearly exemplifies the concept of control steps, by representing their boundaries with dashed lines, and assigning a unique name to each one of them (CS1, CS2, CS3). The total number of control steps in the DFG determines the overall delay of the circuit, and is defined as the *critical path*. A DFG annotated with such scheduling information is sometimes referred to as a *scheduled data-flow graph* (*SDFG*) [103].

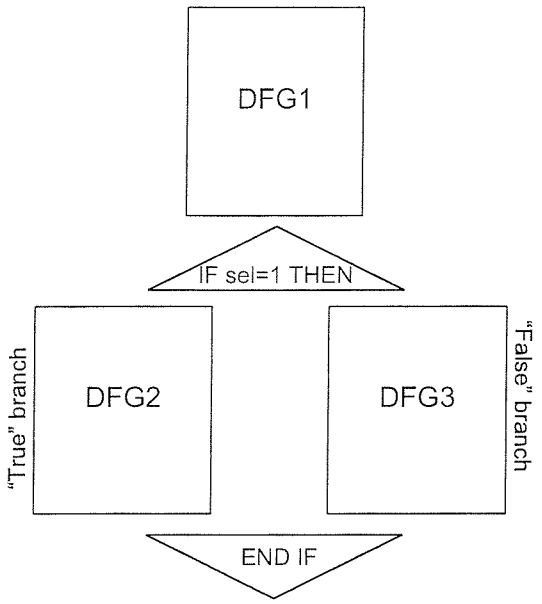


Figure 3.4. An example control and data-flow graph

While a DFG representation is widely accepted as a convenient notation to represent sequential circuits, it is not suitable for representing designs that include conditional branches or iterative loops. It has, however, been extended to include such constructs, thus giving rise to the *control and data-flow graph* (*CDFG*), loosely defined in the following [8, 102].

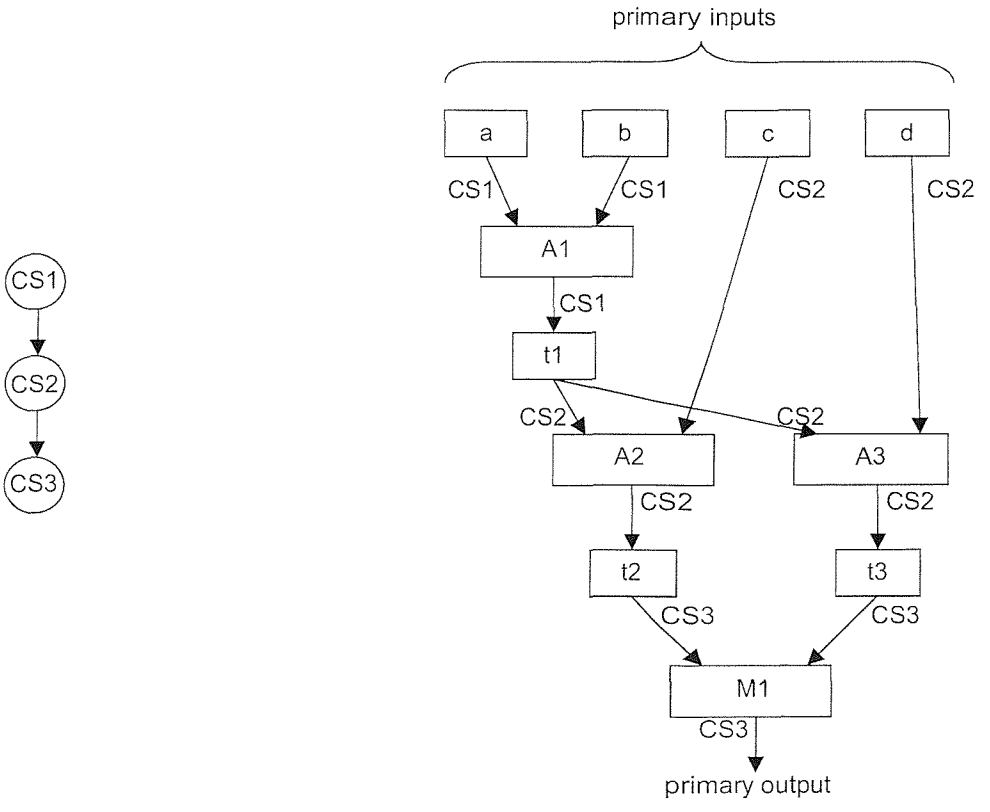
Definition 3.3 : A control and data-flow graph is a hierarchical structure, which at the top level describes the flow of control through the system as a directed graph, where each vertex either corresponds to a separate DFG segment, or is a special "branching" vertex.

Figure 3.4 exemplifies the concept of a "branching" vertex, to represent a conditional execution situation. The rectangles annotated DFG1, DFG2, DFG3 correspond to normal DFGs, like the one of Figure 3.3, while triangles signify branching nodes. Note that the delay through branch DFG2 is not necessarily the same as that through DFG3; in such cases, the critical path is defined as the longest among all paths that lead from the initial control state to the final one. Once again, the critical path determines the overall delay value of the system.

Alternatively to the DFG / CDFG representations, *extended timed Petri-nets* (*ETPNs*) [8] can be formed. In contrast to the former, the latter require two different graph structures for the control and data path parts of each design. In ETPN representation, the control path is represented by a directed graph whose vertices correspond to the control states of the design, and whose edges signify the flow of control. The graph representing the data path is composed of nodes naturally corresponding to functional units *and* storage elements, with edges connecting nodes when there is flow of data between them. Edges in the data path graph are annotated with the symbolic name of the control state during which flow of

data occurs; conditional edges in the control path graph are annotated with the symbolic name of the data path signal that determines which branch will be followed. As an illustrative example, Figure 3.5 shows a Petri-net equivalent of Figure 3.3. In contrast to the DFG case, storage units are explicitly shown in the data path graph, and in Figure 3.5b they are signified by the symbols a, b, c, d, t1, t2 and t3. Simple comparison of Figures 3.3 and 3.5 is enough to show the increased memory storage requirements that a Petri-net based internal implementation requires. It is also obvious, however, that such a representation makes more information readily available; it is therefore more beneficial in terms of performance if frequent access to the data structures is needed.

In the rest of this thesis, both DFG and ETPN-based representations will be used for illustration purposes, as applicable per situation.



(a) control path *(b) data path*

Figure 3.5. Extended Timed Petri-net based representation of an example digital system

3.1.2 Optimisation and Design Space Exploration

Bearing in mind the concepts of scheduling, allocation, and binding explained through Figure 3.3, it is now possible to provide a definition for the optimisation task.
Definition 3.4 : The design *optimisation* is the process of determining the optimal scheduling, allocation, and binding for a design, such that the user specifications are satisfied.

The design optimisation problem has been shown not to have an analytical solution in finite time. Several heuristic algorithms have therefore been proposed, that aim at providing as good approximate solutions as possible, in as little time as possible. Algorithms exist that address the scheduling, allocation, and binding problems separately, or simultaneously [8, 100]. In brief, scheduling algorithms can be :

- *constructive* : operations are scheduled in turns, one at a time, based on algorithm-specific criteria
- *transformational* : a default schedule is initially formulated, and suitable transformations are subsequently applied to it. They can further be distinguished into *deterministic* (e.g. integer linear programming-based), and *stochastic* (e.g. simulated annealing)

Similarly, allocation is typically done using either :

- *iterative / constructive* techniques : similarly to their scheduling counterparts, operations are allocated one at a time in turns, or
- *global* techniques : these techniques rely on analysing the data path as a whole, and then trying to simultaneously allocate all (or a significant number of) operations. They are normally based either on *graph theory*, or on *mathematical programming* (e.g. once again, integer linear programming).

Any further presentation of generic optimisation algorithms exceeds the scope of this work. The algorithms employed by the MOODS system are, however, explained in detail in §3.2. For the time being, the concept of *design space exploration* is introduced [8, 102].
Definition 3.5 : Let n be the number of design parameters / user specifications. The *design space* is an n-dimensional space spanned by these parameters, whose points include all possible alternative realisations of a single given design behaviour.

The two parameters always considered first in HLS are the design area (related to the actual production cost of the circuit), and the design delay (corresponding to the system performance). This gives rise to a typical 2-dimensional design space, depicted in Figure 3.6. Clearly, not all points in the design space are achievable, since there are physical limits as to how fast and / or small a circuit implementing a given behaviour can be. The *achievable region* of the design space is thus shown in the figure. However, not all achievable designs are acceptable. The *acceptable region* is the part of the achievable region that comprises designs that satisfy the designer constraints. The process of considering alternative designs within the design space achievable region until a design in the acceptable region is reached, is commonly referred to as *design space exploration*. Since the designer requirements cannot be known a priori, it is important that a high-level synthesis tool be able to explore as much of the design space as possible, as fast as possible, so as to be more likely to satisfy strict constraints, in as many design scenarios as possible.

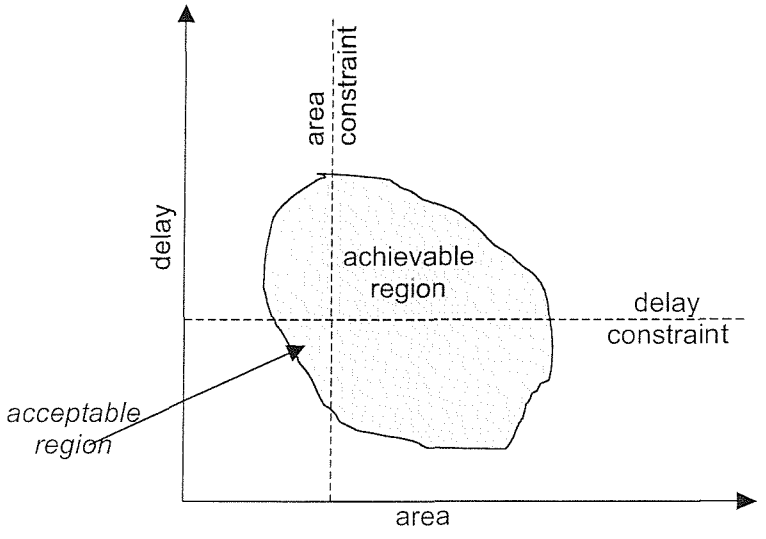


Figure 3.6. Typical 2-dimensional design space

Three-dimensional design spaces have been proposed recently, the third dimension most commonly being power consumption [104], or testability [103]. As will be made evident in chapter 5 (§5.3.3.1), this present work also considers a three-dimensional design space, where *on-line* testability is the third dimension. Of course, in theory the design space can have more than the physically representable three dimensions.

In principle, transformational optimisation approaches are more abstract, take more computational time and are capable of escaping local minima in the design space. Suitable constructive approaches have sometimes been quoted to give better solutions [100], but in theory they may not always reach the global minimum.

3.2 The MOODS High-level Synthesis System

In this section, elements of the way MOODS performs the design space exploration process and comes up with design implementations are provided. In brief, when MOODS is first invoked, the behavioural VHDL description is parsed and analysed, and an initial, naïve, maximally serial implementation of the design is formulated and stored in its internal data structures (internal representation). In this implementation, every operation is scheduled on a separate control step and allocated to a separate data path unit. Clearly, it is the biggest and slowest possible realisation of the design and it serves only as a starting point. This initial design is consequently optimised by applying local, semantic-preserving scheduling, allocation and binding *transformations* to it, in an iterative manner, through multiple repetitions of an *optimisation loop*. The selection and targeting of transformations to be applied is supervised by a suitable *algorithm*, and guided by a *cost function*. After optimisation, and in line with the paradigm of Figure 3.2, an RTL VHDL netlist is output. This netlist is effectively an interconnection of instances of cells from a suitably provided MOODS cell library.

Topics covered in the following subsections are : the design internal representation within MOODS (§3.2.1), the optimisation loop (§3.2.2), the set of available transformations (§3.2.3), the cost function (§3.2.4), the algorithms currently available (§3.2.5), details about the hardware model assumed for the control path (§3.2.6), and finally a list of the MOODS cell library components (§3.2.7). Emphasis is naturally given to these elements that are essential for this thesis, while further details can be found in the literature [74, 73, 8, 75, 104].

3.2.1 The MOODS Internal Representation

From the brief description in §3.2, it is clear that optimisation within MOODS is an iterative process. This applies to both the scheduling and allocation tasks, since they are actually considered simultaneously, within the same optimisation process (§3.2.2). Therefore, the data structures that form the internal representation are expected to be accessed very frequently. As explained in §3.1.1, this makes Petri nets a tempting option for the internal representation. Indeed, the representation formed within MOODS closely resembles

ETPNs in that separate structures are stored for the control and the data path, in principle formulated following the ETPN rules. However, it also features some non-ETPN elements. To name just a few, these include :

- the control path node data structures include information about operations scheduled for execution in them
- there exist software pointers called *implementation links*, connecting operation nodes in the control path with datapath functional units
- a comprehensive set of control path node types is used, enabling the efficient representation of a substantial subset of behavioural VHDL constructs
- an additional *condition list* data structure encompasses information about instructions executed only on a certain condition

All these additions (plus others not mentioned here) significantly enhance the semantic power of the representation, and are presented in detail in [8].

3.2.2 The Optimisation Loop

The optimisation loop of Figure 3.7 is the heart of the optimisation process. It defines the stages through which the system routinely cycles whenever an optimisation transformation is considered, regardless of the actual nature of the transformation. The whole iterative optimisation process is thus nothing but several repetitions of this loop. The different phases of the optimisation loop are explained in the following.

During the *selection* phase, a transformation is picked from the set of available transformations (§3.2.3) and the data which it will target are also selected. The optimisation algorithm (§3.2.5) determines which transformation and data will be selected. Alternatively, MOODS can run in an interactive mode, during which the designer goes through the optimisation loop "manually". It is to be noted, however, that irrespective of the applied algorithm or interactive mode option, optimisation always proceeds according to the scheme of Figure 3.7.

As is further clarified in §3.2.3, any given transformation can only target specific kinds of data. For example, if a unit sharing transformation is selected, appropriate data are two distinct datapath units of the same type (or compatible types). In addition to that, the design characteristics at a given time may (and often do) prevent a particular transformation

from being applied to a given set of data. In the example at hand, unit sharing is prevented if, for instance, the two datapath units are both active during the same given control step. Such design characteristics are checked during the *validity test* stage.

If the given transformation on the given data is determined to be valid, the system proceeds to the *cost estimation* stage. It is during this stage that transformations are actually evaluated and it is determined if they are beneficial or degrading. This is done through the *cost function* (§3.2.4). Note that the same transformation on the same data in a given design may be either beneficial or degrading, depending on the designer specifications, reflected on the cost function.

Whether or not the transformation will actually be applied is finally determined, once more by the algorithm currently in use (or by the designer, if in interactive mode). Indeed, there are optimisation algorithms that occasionally accept degrading transformations.

The execution stage of the loop is self-explanatory : the transformation is finally applied, that is, the internal system data structures are modified so as to reflect the change in the conceptual design realisation.

After execution, or if either the

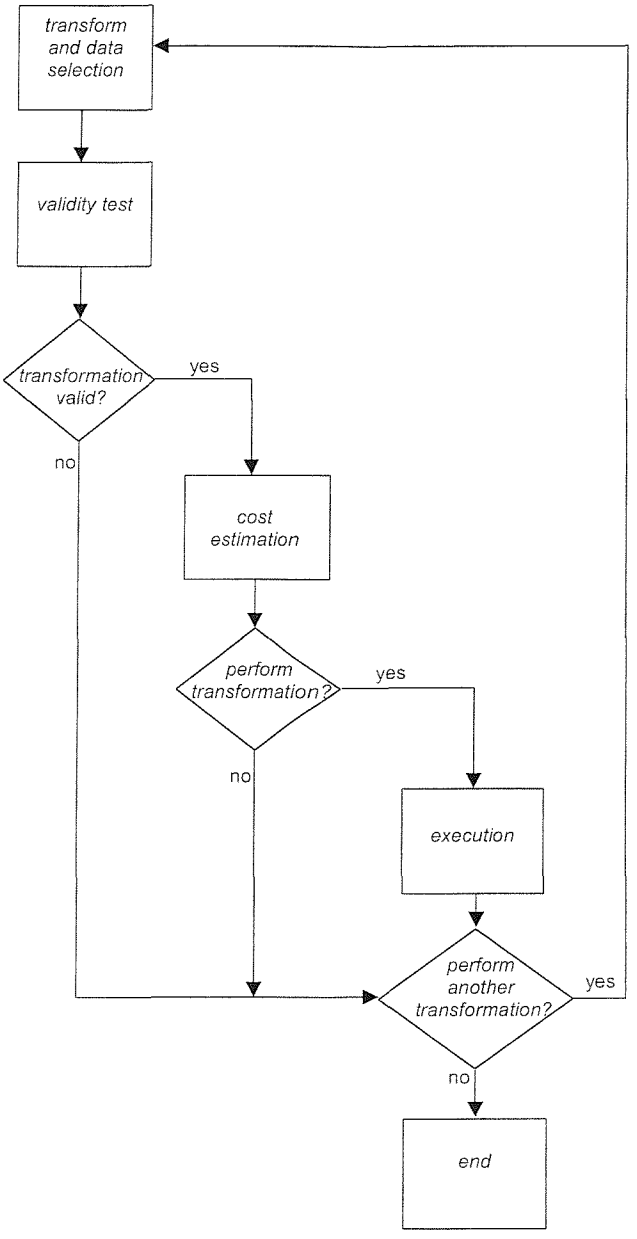


Figure 3.7 : The MOODS optimisation loop

validation or the execution stage fails, there is the option to either proceed to another transformation or finish optimisation. The point at which optimisation actually finishes is once again determined either by the algorithm in use or by the designer interacting with the system.

3.2.3 Transformations

symbolic name	description	type of transform
TF2	sequential merge	scheduling
TF3	group instructions on register	scheduling
TF6	ungroup to time	scheduling
TF7	ungroup on group	scheduling
TF8	merge fork and successor	scheduling
TF9	parallel merge	scheduling
TF10	share functional unit	allocation
TF12	unshare functional unit fully	allocation
TF13	unshare single instruction from functional unit	allocation
TF21	unshare single instruction from control state	scheduling

Table 3.1. The set of available transforms

Table 3.1 presents the set of transformations available within MOODS. Each transformation is uniquely identified by a symbolic name appearing in the first row; a brief description is also provided in the second row. Finally, the third row gives the type of the respective transformation, that is, classifies it as either a scheduling or an allocation transformation. These transformations are explained in more detail in the following [8].

The *sequential merge* transformation (TF2) targets two sequential control nodes, as are, for example, nodes CS1 and CS3 of Figure 3.5. It results in a single control step, encompassing all operations of the targeted steps. Practically, all operations of the temporally preceding step (CS1 in the example) are moved to the temporally succeeding (CS3), and the former is optimised out, thus saving one control step in the overall critical path. If the merged control nodes include any two operations that feed one another, then the register that originally stored the intermediate result across the CS boundary is also optimised out, and the two operations are *chained* within the resulting control step. One single operation, or two or more operations scheduled for serial (chained) operation within the same control step will hereby be referred to as an operation (or instruction) *group*. In order for transformation TF2 to be applied, the test phase of the optimisation loop checks that : a) no in-

structions in any of the merged control steps share any hardware, unless they are mutually exclusive, b) any possible chaining does not violate the designer clock period specification, and c) there are no data dependencies between the top state instructions and any instructions in the states between the targeted ones. For example, referring back to Figure 3.5, this last check ensures that no output of any operation in CS1 is needed in CS2; therefore, the operations of CS1 can be moved to CS3.

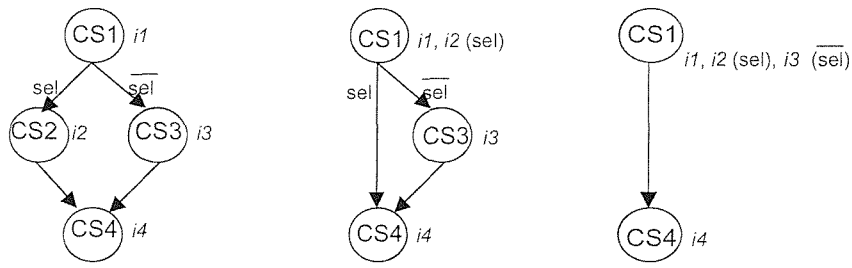
Transformation TF3 (*group instructions on register*) targets a given register, and aims exclusively at optimising it out, by chaining the two instructions writing to and reading from it. Once more, the corresponding control states are merged, and the instructions form a group. Clearly, TF2 and TF3 can often have exactly the same effect; however, their starting points (targets) are different, and are therefore considered separately. The tests required to ensure validity of this transformation are the same as for TF2, plus an additional check that no other instruction writes to the given register, or reads from it, so it can safely be removed.

Transformation TF6 (*ungroup to time*) is the first "undo" transformation presented here. It targets a single control node, and it is only meaningful if the targeted node is the result of any of the merging transformations (TF2, TF3, TF8, or TF9, the last two presented later in this section). It also takes a maximum execution time value as input, and checks whether the given node requires more than this time for all its operations to be fully executed. If it does, then the transformation tries to locate any groups of chained instructions, and ungroup them, by introducing new control steps and new registers to carry intermediate values across their boundaries. Although new control steps are introduced, possibly lengthening the critical path, the transformation can result in actual improvements in system performance, since breaking long chains of operations is often expected to enable higher clock frequency values to be achieved. Further, "undo" transformations are useful in algorithms that accept temporary degradation in system quality within the optimisation process, such as the simulated annealing algorithm (§3.2.5). In practice, TF6 can rarely be invalid, mostly in situations resulting from the sharing of registers among several system variables. Such sharing is, however, not permitted in this work.

TF7 (*ungroup on group*) is another "undo" transformation. Once more, it targets a single CS, and it is meaningful only when the given CS encompasses more than one group of in-

structions. This can be two or more single (or chains of) instructions executed in parallel within the same step. It simply creates a new dedicated control step and schedules a chosen group in this step.

TF8 (*merge fork and successor*) is a control step merging transformation, allied to TF2 and TF3. It involves *fork* nodes in the control path, that is, nodes with multiple output edges, resulting from conditional or loop behavioural VHDL statements. It merges a given fork node, with its immediate successor, practically by moving the operations executed in the successor node up to the top one, in the form of operations executed *conditionally*. A simple example is considered in Figure 3.8. 3.8a depicts the original situation. CS1 is the fork node, while CS2 and CS3 are the two successors. The CSs are also annotated with the instructions that are scheduled in them, *i1*, *i2* and *i3* respectively. When condition "sel" is true, then CS2 is visited and *i2* executed; otherwise CS3 is visited and *i3* executed instead. Both cases are followed by CS4 and the execution of its respective instruction *i4*. In Figure 3.8b, TF8 targets steps CS1 and CS2. As the figure shows, CS2 is dropped and *i2* is moved to CS1, together with its execution condition "sel". A second immediate execution of TF8, this time targeting CS1 and CS3, results in the simple situation of 3.8c, where the fork node and both of its successors have been substituted by a single node, featuring the original *i1* and two mutually exclusive instructions. Having abolished the fork construct, the system now has enhanced potential for scheduling optimisation, by further applying sequential merge transformations. Interestingly, TF8 can be considered as a generalised version of TF2, since any normal control state within a sequential branch (Figure 3.5), can be considered as a "fork" with a single successor. This is why it is often used within MOODS instead of TF2. Naturally, the validity check for TF8 consists of the usual hardware sharing and clock period tests.



(a) original state (b) applying TF8 on CS1, CS2 (c) applying TF8 on CS1, CS3

Figure 3.8. TF8 example

The last control step merging transformation available in MOODS, is the *parallel merging* one (TF9). This transformation targets two parallel control nodes, that is two nodes that are unconditionally visited concurrently. Clearly, there is no reason why the control path cannot be simplified by merging the two into one, encompassing all concurrently executed instructions of both. When TF9 is considered, the check phase of the optimisation loop simply needs to verify that the given states are truly parallel.

Three allocation transformations are presented next. TF10 (*functional unit sharing*) naturally targets two functional units and attempts to create a combined one, and allocate to it all the instructions originally allocated to the targeted units, by introducing suitable multiplexers to implement time-sharing. Clearly, the validity check phase should ensure that the units are not concurrently active (no concurrently executed operations have been allocated to them, except mutually exclusive ones). Of course, the two units must be either of the same type (e.g. two multipliers), or of such types that can be combined into a single arithmetic and logic unit (ALU). The latter case will not be further considered here.

TF12 (*unshare functional unit fully*) targets a single functional unit that has been the result of one or more executions of the previously presented TF10. The result of TF12 is a number of new, suitable, non time-shared units, each one of them implementing only one of the operations previously allocated to the targeted unit. This transformation is always valid, although it is meaningless if a unit implementing a single instruction is targeted.

TF13 (*unshare single instruction from functional unit*) is a low-level version of unit unsharing. Just like TF12, a previously combined functional unit is targeted; this time, though, one of the instructions it implements is also given. It results in a single unit implementing the given instruction, and an additional unit implementing all the instructions previously assigned to the targeted unit, except the extracted one. TF13 is naturally also always valid, provided that it is meaningful.

The last transformation presented here is TF21 (*unshare single instruction from control state*). It targets a particular instruction, and creates a dedicated control state for it, either before or after its original control state. Any other instructions originally scheduled for the original control state are either unaffected, or have a new control state created for them, if data dependencies suggest so. This transformation can always be applied, and it does not

greatly contribute to the tool optimisation potential. It is, however, a powerful tool in certain situations (e.g. in the expanded module experiments described by Williams [8], as well as in §6.4.3 of this thesis).

Note that no binding transformation has been described in this subsection; indeed, in the version of MOODS used for the purposes of this work, there only exists one hardware cell for every functional module type (§3.2.7). Binding is therefore restricted to a one-on-one mapping of modules to cells, and does not provide scope for iterative optimisation decisions. However, an "alternative binding" transformation exists within MOODS [8]; this transformation could accommodate a more evolved version of the cell library, thus considering the binding problem within the optimisation loop. Finally, a few additional transformations are mentioned in [8], such as register sharing, and clock period scaling, but are mostly implemented for experimental purposes, they are not shown to be critical for the optimisation process, and are not explicitly considered here.

3.2.4 Designer specifications and the cost function

As a first step towards setting up a synthesis session, the designer specifies his or her constraints in terms of the design characteristics, namely :

- area (in a technology-specific unit, e.g. logic gates or FPGA slices for ASIC or FPGA technology respectively)
- delay (typically in nanoseconds)
- clock period (also in nanoseconds)

Other characteristics also mentioned in [8] (total number of nets, static power consumption) are not considered here.

The clock period value is used during the test phase of several state merging transformations (as mentioned in §3.2.2), to determine the feasibility of the given transformation. Typically, a low period value prevents excessive state merging and operation chaining, but of course leads to a high frequency final implementation. If the designer specifies no clock period value, the system calculates a default one based on the current implementation details [8]. This is not further considered in this thesis, and a designer-specified period is implied hereafter.

Regarding the other two design parameters, the designer also gives corresponding priority (first or second) preferences. Using these priorities, MOODS implements a *cost function*, quantifying the quality of any given instance of the system under optimisation. This function is invoked in the optimisation loop during the cost estimation phase (§3.2.2, Figure 3.7), and is used to forecast the effectiveness of the considered transformation, with close respect to user requirements, as these are reflected in both the desired values and the specified priorities.

More specifically, MOODS constantly keeps track of the circuit area calculated using the following formula :

$$area = \sum area_{dp} + \sum area_{cp} + \sum area_{i} \tag{3.1}$$

where the three factors represent the sum of the area of all data path units, the sum of the area of all hardware modules constituting the controller, and the sum of all interconnect modules (multiplexers) respectively. It should be recalled that the module area values used to calculate (3.1) are known to the system through the technology library.

The total delay is simply calculated as the product of the critical path by the clock period :

$$delay = (critical\_path) \times (clock\_period) \tag{3.2}$$

The cost function characterising the system can now be expressed as :

$$Cost = c_{area} \times area + c_{delay} \times delay \tag{3.3}$$

c_{area} and c_{delay} are priority-related (therefore *designer-specified*) weighting constants. In essence, the goal of the whole optimisation process is the minimization of equation (3.3).

Practically, during the estimation phase, the synthesis system calculates the change in "energy" of the circuit that is expected to occur if the transformation under consideration is applied. The change in energy for a given parameter P is given by :

$$\Delta E = \frac{P_{estimate} - P_{current}}{P_{initial} - P_{target}} \tag{3.4}$$

where

$P_{current}$ is the current value for parameter P (calculated by either equation (3.1) or (3.2), depending on P)

$P_{estimate}$ is a rough estimation of the effect the transform will have on the value of P if it is applied

$P_{initial}$ is the value of P in the initial, totally unoptimised design (§3.2)

P_{target} is the user specification itself

Very often P_{target} is assumed to be assigned the zero value, both for area and for delay. While neither of them is feasible, such a set of specifications can be used by a designer to demand a circuit that would be "as hardware-efficient as possible, and as fast as possible". For the rest of this thesis, this assumption will be implied, unless otherwise stated. Under this assumption, P_{target} can be omitted from equation (3.4), and the equation then expresses the estimated change in the value of the parameter, normalised over its initial value. Regarding the $P_{estimate}$ value, this is calculated separately for every transformation, by specially written software functions within the synthesis system, giving emphasis on speed of calculations, rather than on accuracy.

The overall energy change of the design is nominally calculated by averaging the energy changes of all *first* priority requirements *only*. Given that only two parameters have been mentioned up to now, this practically means that if area optimisation is the first priority and delay the second, then only the change in area energy is considered, and vice versa. Averaging occurs when the designer specifies equal priorities.

Ultimately, if the energy change of the design is negative, then the transformation is considered to be beneficial; otherwise, it is regarded as degrading.

3.2.5 Available algorithms

As regards the algorithm that supervises the optimisation process, in the current version of MOODS there is a choice of either the general-purpose simulated annealing algorithm, or goal-oriented tailored heuristics. These choices are described in the following subsections.

3.2.5.1 Simulated annealing

Simulated annealing [8] is a generic optimisation algorithm for minimising functions of many variables (in our case, the cost function). Its name is derived from the statistical me-

chanics method of annealing in solids. In the synthesis context, the designer specifies an "initial temperature", a "terminating temperature" and parameters to determine how slowly "temperature" will decrease. Random transformations are chosen and evaluated. At any given point of time, the current temperature value T and the estimated energy change associated to the transformation at hand (§3.2.4), are used to calculate a *threshold* value *th*, as in the following :

$$th = e^{(-\Delta E/T)} \tag{3.5}$$

If a transformation is improving, it is applied; otherwise, a random number is generated, and if it is greater than the threshold, then the transformation is rejected. If the random number is lower than the threshold, then the transformation is applied *although* it is degrading. The temperature is decreased in every optimisation loop step, and at the same time the threshold is reduced, as can easily be confirmed from equation (3.5). Therefore, the more time passes (and the lower the temperature gets), the more the probability that degrading transformations will be accepted decreases. Accepting degrading transformations in early stages of the design process can be useful to avoid cost function local minima, therefore exploring the design space better, in the search for the global minimum. As the design "cools down", only upgrading transformations are accepted, so that the global minimum is reached. Despite its randomness, this algorithm asymptotically converges to the global minimum of the function under minimisation.

The main advantage of simulated annealing is its abstractness and its ignorance of any physical significance of the variables that the cost function under minimisation depends on. Effectively, using simulated annealing, whatever can be quantified and included in the cost function, can also be optimised. The main problem is its very slow speed, especially for large designs. In essence, while an optimum solution is theoretically guaranteed, the algorithm is so slow that it can be impractical for the designer to wait for it.

3.2.5.2 Tailored heuristics

In order to speed up the design process, goal-oriented tailored heuristics are also available. There are three versions : oriented towards minimising area, delay or both. The basic idea behind these heuristics is reflected in Figure 3.9. In the DFG of 3.9a, the original state of two control steps, featuring a single addition each, is shown. In 3.9b, transformation TF10

(unit sharing, §3.2.3) is applied, and the adders implementing the two operations are combined. The design is clearly optimised for area, *but* control steps CS1 and CS2 can no longer be merged unless the algorithm allows a degrading "undo" transformation. In the alternative 3.9c, the design is optimised for speed, by merging control steps CS1 and CS2 through TF2 (sequential merge, §3.2.3); *however*, the adders cannot be combined any-

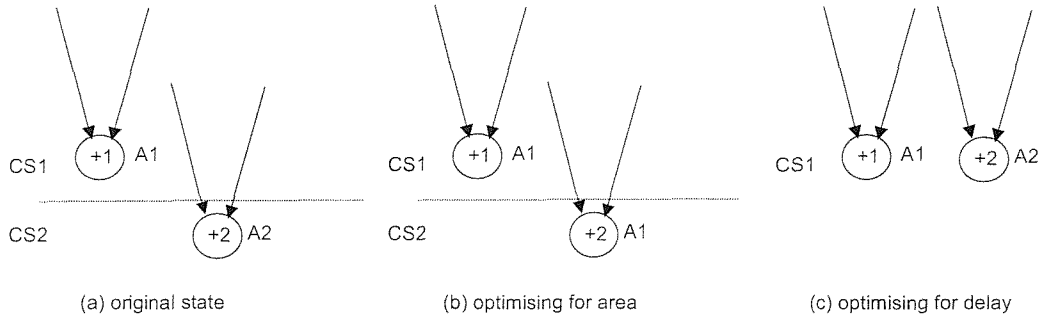


Figure 3.9. A simple data-flow graph : optimising for contradicting goals

more, as they are active concurrently, therefore the design will be fast and comparatively expensive. This small example illustrates the well-known concept that area efficiency and speed are contradicting goals; further, it shows that if either of them is first priority over the other, then as much optimising of the first priority as possible needs to be carried out, *before* considering the second. Otherwise, optimising the second priority goal is likely to block the first, and that would be a most undesirable effect. Moreover, if the topology and the operation of the circuit permit it, it would be beneficial to optimise the first goal in such a way, that situations like these of Figures 3.9b, 3.9c are avoided, in order that the optimisation potential of the second goal is not hindered unnecessarily.

In order to serve these purposes, the tailored heuristics framework further associates a number of metrics and indicators with the MOODS internal data structures corresponding to a given design. These metrics and indicators are briefed in the following :

- a *shareability factor* is associated with each data path unit. In effect, this factor expresses the area that will be saved if the unit at hand is combined with all other units of the same type, thus quantifying its hardware sharing potential. Clearly, when optimising for area, datapath units with high shareability factors should be preferred.
- a *slack* value is also associated with each control node, suggesting how "far away" from the critical path the node is. A zero slack value signifies a node on the critical path, while positive values indicate non-critical path nodes; further, the shortest the path on

which the node is, the highest the slack value [105]. When optimising for delay, control nodes on the critical path should be targeted primarily.

- a *critical path factor* is calculated for each datapath node [8], corresponding to the percentage of instructions implemented by the unit at hand, that are scheduled for execution at control nodes on the critical path. If units with high critical path factors are shared, then situations like that of Figure 3.9b are likely to arise and block subsequent critical path node merging / delay optimisation. It is therefore desirable that when optimising for area, preference be given to units with low critical path factors.
- a *share factor* is calculated for each control node; this corresponds to the percentage of operations scheduled for the particular node, that have been allocated to a unit with a positive shareability factor. Merging control nodes with high share factors is likely to produce situations like the one of Figure 3.9c, where no subsequent area optimisation is possible. It would therefore be preferable to choose control nodes with low share factors, if such nodes can be identified in the system.

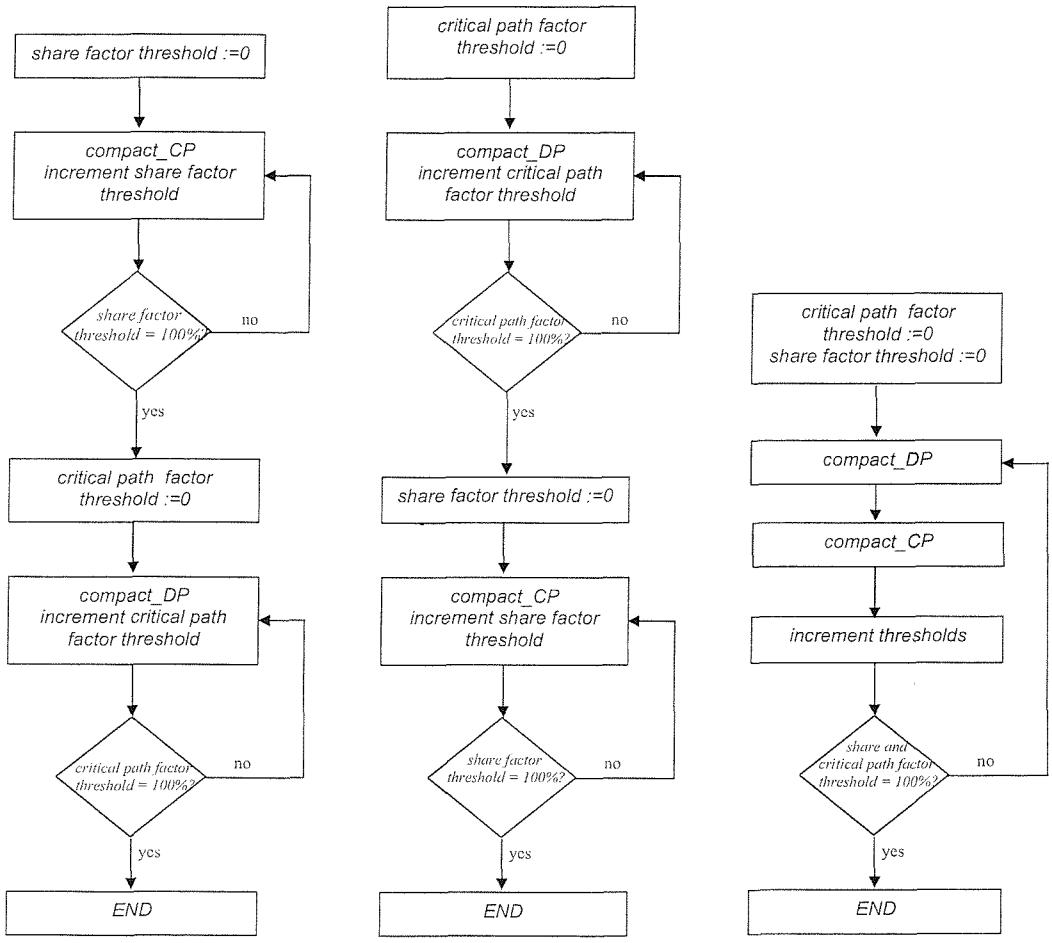
Based on the above indicators and metrics, two software routines have been defined, that are later suitably combined to construct the heuristic optimisation algorithms. These routines aim at optimising the first priority objective, while minimising the negative effects on the secondary one. They are :

- *compact_CP* : it is used to minimise the critical path length, by successively applying transformations TF8 and TF9 (§3.2.3). It is fed by a threshold share factor value, and targets all nodes whose share factors are calculated below that threshold.
- *compact_DP* : performs hardware sharing between functional units, by repeating transformation TF10 (§3.2.3). A threshold critical path factor value is given to it, and all data path units with critical path factors below that threshold are considered.

The flow charts for the tailored heuristic optimisation algorithms are now shown in Figure 3.10, taken from [8]. From the flow charts, it is obvious that an initial zero value is first assumed by the threshold values, to be incremented in subsequent iterations. This way, the optimisation moves that are most effective as far as the first priority is concerned, and less impairing, as regards the second criterion, are given preference. A more complete version of the tailored algorithms is given in [8], taking into account the possibility to meet user constraints before the threshold value takes the 100% value; in this presentation, the flow

charts have been simplified under the "as cheap as possible and as fast as possible" assumption mentioned in §3.2.4.

As is evident from the above description, only a small subset (TF8, TF9, TF10) of all the available transformations are considered in the tailored heuristics. The reason for this is that these three transformations have been shown to contribute the most towards the optimum design solution. The heuristic approach has been proved to be much faster than simulated annealing. However, it is absolutely parameter-oriented and therefore not easily expandable to include additional criteria. Further, there is always a risk to end up in a local minimum instead of the global that is searched for, because only improving transformations are considered.



(a) Tailored to delay (b) Tailored to area (c) Equal priorities

Figure 3.10 : Flow charts for the heuristic optimisation algorithms

3.2.6 Hardware model

This chapter continues with a presentation of the actual hardware model of the MOODS output designs. As already mentioned, the data path is simply an interconnection of functional units, registers and multiplexers (a list of these building blocks is given in §3.2.7). Therefore, it conforms to the typical structural / RTL data path modelling. The MOODS controller architecture, though, has some interesting properties, and it is for that reason that it is presented here in detail, given also that it is greatly referred to in chapter 6, when controller self-checking design is considered.

Figure 3.11 shows a conceptual model depicting the communication between the controller and datapath in a system like the one of Figure 3.1. The datapath is shown on the right-hand side of 3.11 in a form that resembles a data-flow graph, where storage elements are also shown (although this is not consistent with the formal definition 3.2). In the particular data path example, four operations (O1 – O4) are scheduled over three control steps (N – N+2), and the registers shown are used to store and preserve intermediate results across control state boundaries. The internal structure of the controller is not yet revealed; nevertheless, the figure shows how the controller outputs (hereafter *control signals*) connect to the data path. Specifically, the control signals feed the storage registers' "load enable" ports, and this connection determines when the operation is actually executed. For in-

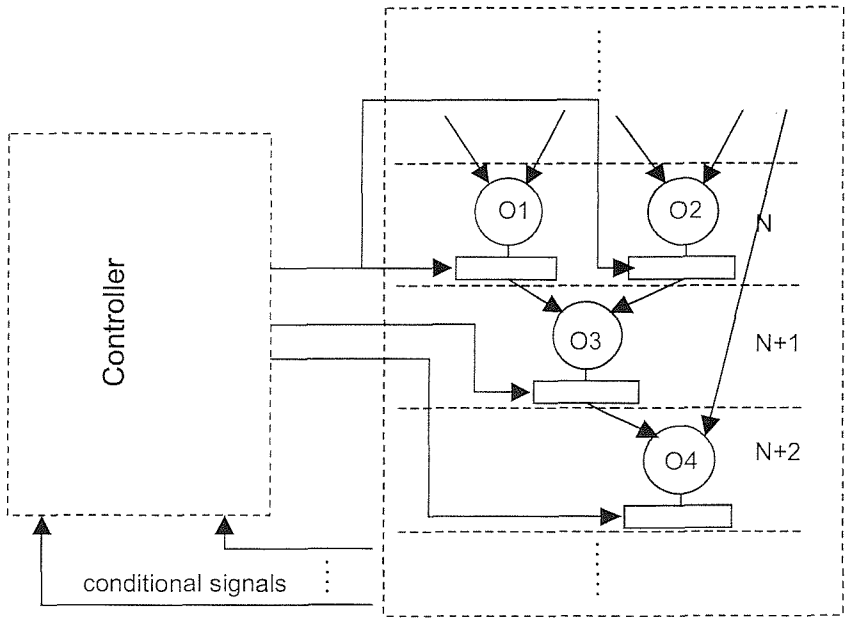


Figure 3.11 : Communication between the data path and the controller

stance, operation O4 is assigned to a functional module; assuming that the module is combinational, some logic value always appears at its output port. However, the value is stored in the appropriate register *only* at CS N+2; in that sense, O4 is executed *only* at N+2, and the corresponding functional unit is said to be *active* only then. Clearly, in order for this to happen, the control signal corresponding to N+2 should assume the "true" value during CS N+2, while all other control signals should assume "false". If (without loss of generality) "active-high" encoding is assumed, then this example shows that the controller output should by definition be "one-hot" encoded (§2.2.1.2). While this is a general observation that applies to all controller / datapath architectures, the actual controller implementation can be quite different from system to system.

For the sake of completeness, it should be noted that, together with register "load-enables", the controller outputs also feed multiplexer "select" ports in the data path. This has no implication whatsoever as regards the purposes of the present thesis, and will therefore not be considered any further.

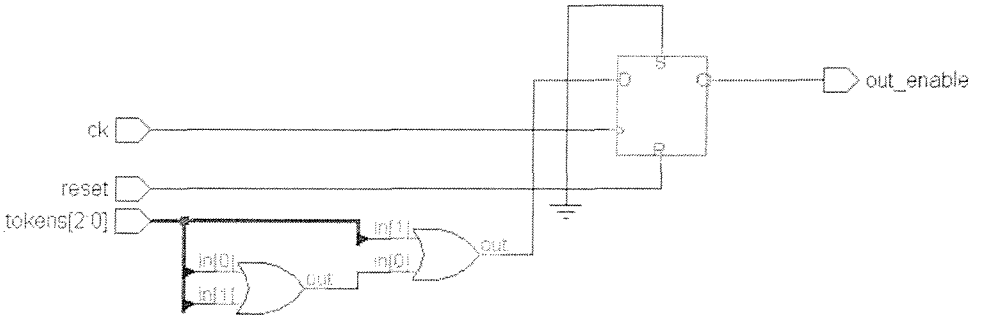


Figure 3.12 : The general control cell

Within MOODS, the controller is implemented using a special hardware cell defined within the VHDL cell library (§3.2.7), namely the *general control cell.* Being part of the library, this cell is described in RTL VHDL, and its actual structure is derived by RTL synthesis tools. Figure 3.12 shows the typical implementation for this cell, as synthesised by Mentor Graphics LeonardoSpectrum, version 2002e.16. A cell of this type corresponds to a unique state in the control path. A D-type flip-flop is the basic building block for the cell. The D-input of the flip-flop is the OR function of a number of tokens, corresponding to the predecessor states in the control path. In the example of Figure 3.12, a 3-bit token input is shown, meaning that the given state is the successor of any of three different

states. If the control state implemented by the particular cell is visited conditionally, then the input tokens are the result of the AND function of the corresponding predecessor state signal(s) with suitable conditional signals, produced by the data path. Finally, the flip-flop Q-output (labelled out_enable in 3.12) is essentially the control signal of the state at hand, which is fed to appropriate data path storage units, as well as to the successor state(s) general control cell input token(s).

The whole controller is thus implemented as Figure 3.13 shows. The control signals are directly led to the datapath; they are also fed back to the general control cells, properly rearranged so that each control signal is input only to the cell(s) corresponding to its successor state(s). In addition, conditional control transitions are implemented where necessary by a block of AND gates, also fed by the appropriate conditional signals, as shown. In effect, the operations described above (ANDing, followed by rearrangement, followed by ORing within the control cells, as shown in Figure 3.12) correspond to the next state sig-

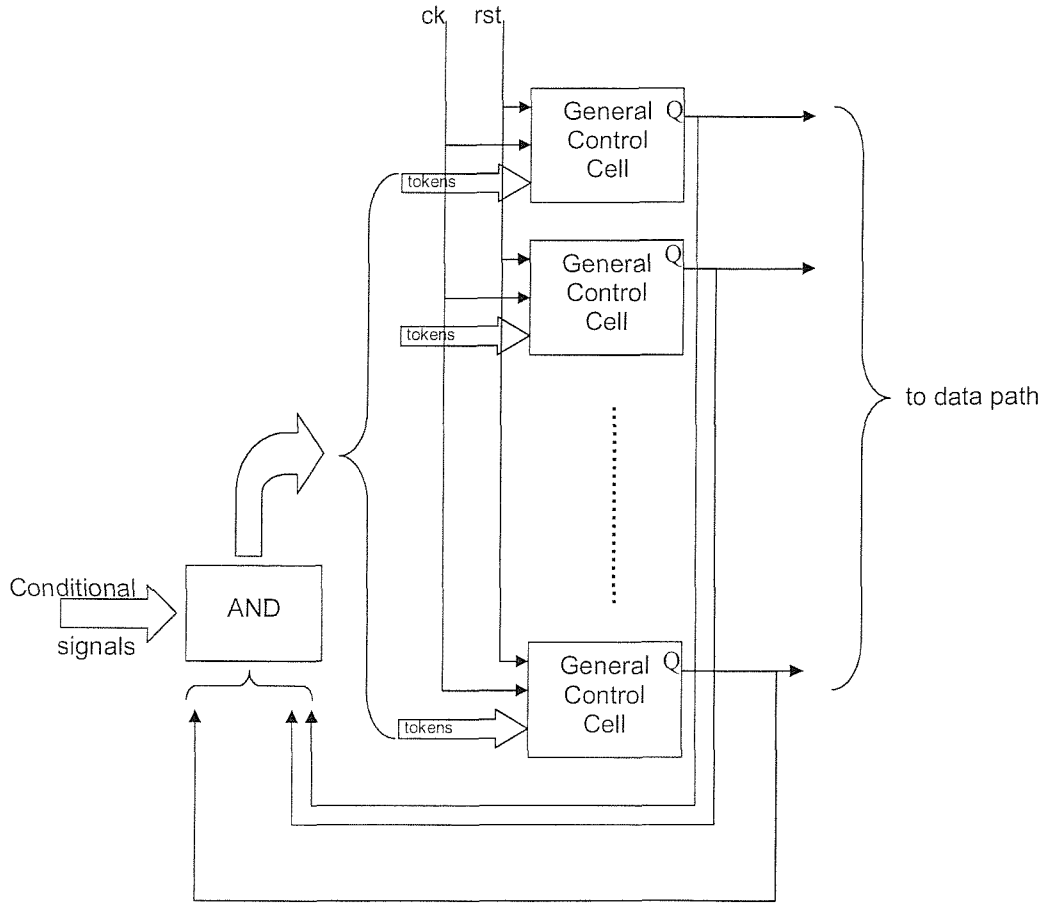


Figure 3.13 : The controller generated by MOODS

nal generation, thus making the MOODS controller model a proper FSM. Clearly, the number of flip-flops is equal to the total number of control states. This is considered to be an expensive but fast implementation in VLSI (ASIC) technology, while it appears much cheaper in FPGA technology, due to the existence of dedicated storage elements within FPGA slices [106].

3.2.7 The cell library

This subsection concludes the presentation of MOODS by providing a list of the hardware cells made available to the system through the cell library. These cells include [105] :

- logic gates : "NOT", "AND", "OR", "NAND", "NOR", "XOR", "XNOR"
- conventional, single-bit output equality comparators : "=", "≠"
- unsigned and signed integer arithmetic comparators : "<", "≤", ">", "≥", ">/≤" (ALU)
- left and right "shift" and "rotate" modules and ALUs
- both unsigned and signed integer arithmetic functional blocks : negator ("unary minus"), ripple-carry adder – subtractor – add/subtract ALU, incrementer, decrementer, multiplier, absolute value calculator
- typical digital logic RTL blocks : register, up-counter, down-counter, multiplexers, decoder
- control cells : general control (§3.2.6), call control
- auxiliary cells : concatenation, unsigned and signed bit extension

The functionality of most of these cells is obvious from their names, as they correspond to the usual elementary operations found in VHDL or any other programming language. ALUs in this context are essentially combined cells capable of implementing alternative types of operations, depending on the value of suitable controlling signals. The "call control" cell is a special cell used to implement VHDL `procedure` and `function` constructs. With such a rich collection of hardware modules, a very good subset of the VHDL language can be synthesised. This subset includes all common logic and integer arithmetic statements, loop and conditional statements, subprograms, as well as multiple concurrent communicating `process` blocks.

3.3 Summary

The fundamental concepts of high-level synthesis have been covered in this chapter. Particular emphasis has been given to the incarnation of these concepts within the MOODS high-level synthesis system. As a concluding remark, it is important to once again stress that the whole performance of MOODS highly depends on the following three elements :

- the set of transformations
- the available algorithms
- the cost function

Clearly, this means that any attempt to alter, refine or enhance the MOODS functionality should focus on expressing the alterations, refinements or enhancements through the above elements.

Chapter 4

Fault Simulation Techniques

When a fault testing or fault tolerance strategy is applied to a digital circuit, it is desirable to determine or demonstrate its effectiveness against the most commonly occurring faults, before putting the circuit into action. For this purpose, a number of controlled experiments are typically conducted, wherein the behaviour of the system is intentionally altered to imitate its predicted behaviour in the presence of the targeted faults. This is the topic of *fault injection* and *fault simulation.* The relevant material in the literature is extensive; practically, every research group concerned with testing, has to a certain extent also worked with fault simulation, in order to validate their work. Fault simulation experiments have been carried out in this work as well (§7.1.2.1). The present chapter briefly describes a small number of representative fault simulation techniques, thus providing the foundation for the experiments of chapter 7.

4.1 General

In order to validate the reliability of a design, four alternative approaches have been applied [107, 108, 109] :

- *hardware implemented fault injection* : this is done after fabrication, and it consists of injecting faults in a sensitive fabricated chip, by disturbing critical factors of the environment. Most commonly either heavy ion radiation or electromagnetic interference is used for this purpose.
- *software implemented fault injection* : the software of a microprocessor-based system is changed such that the processor behaves as if under the presence of a physical fault.
- *programmable logic implemented fault injection* : the hardware system is initially prototyped on a programmable logic part (FPGA). Faults are injected on the part either

through suitably added control lines or, in certain state-of-the-art FPGAs, through dynamic partial reconfiguration.

- *simulation-based fault injection* : this is done at the pre-manufacturing design stage. Typically, at this stage the system is described in the form of some hardware description language code, and fault injection is done by suitably perturbing this description, so that the resulting system would emulate faulty behaviour.

A survey of hardware- and software-implemented fault injection methods can be found in [110], while [111] includes comprehensive information on radiation-based fault injection in an industrial setting, followed by standardised certification of chip performance in hostile environments. Injecting faults on programmable logic parts is an interesting and relatively new idea, constantly gaining ground as FPGAs themselves gain ground. It is proposed as an alternative to HDL-simulated fault injection, and the main motivation behind an FPGA approach is that programmable logic emulations of hardware parts are much faster than HDL simulations running on general-purpose computers. Therefore, an FPGA-based fault injection experiment is due to finish faster than an equivalent experiment on a software simulator. In [112], Civera et al set up a fault injection configuration based on programmable logic, wherein bit-flips (i.e. bits that have their fault-free values complemented) are injected on the storage elements of an FPGA prototype by a host computer. The injection is implemented by dedicated hardware added to the storage elements, while the information regarding which faults will be injected during a given experiment is communicated to the FPGA from the host PC through a suitable additional system primary input and a chain of "mask" flip-flops connected together in a scan register fashion similar to that of Figure 2.3. Alternatively, Antoni et al [109, 113] exploit the runtime partial reconfiguration capabilities of modern Xilinx Virtex FPGAs [106] to inject faults once more in memory elements, this time by partially substituting the original fault free FPGA configuration with one that demonstrates selective faulty behaviour. The advantage is that no permanent additional hardware infrastructure or primary input needs to be inserted for fault injection purposes; the price is that frequent reconfiguration, even partial, slows down the experimentation, cancelling out the speed benefit of FPGA emulation.

This chapter is hereafter concerned solely with simulation-based fault injection on HDL descriptions.

4.2 Representative simulation techniques

Motivated by the extensive use of the VHDL language [48] in present-day CAD, several researchers have proposed approaches to facilitate fault injection and simulation in VHDL models of digital systems. A representative number of such approaches are covered in this section.

A typical example of injecting faults at the logic gate level can be found in [114]. The first three tasks addressed therein are to analyse the fault behaviour of the basic logic gates, identify fault dominances and equivalences [1], and define corresponding *mutant* gate VHDL descriptions. Mutant descriptions in this context are VHDL models that behave identically to the original gates in the fault free case, but imitate well-defined faulty behaviours when suitable values of added control signals dictate so. Armed with these mutant gate models, and given any complex system gate-level netlist, the authors of [114] substitute the original gates with the mutants, thus providing fault injection capabilities to the overall netlist. They subsequently specify an explicit list of targeted faults. A suitable test bench is further written, that uses the information of the fault list to suitably inject the desired faults (typically one by one) into the modified netlist and observe the responses, with respect to the responses of a fault free simulation run, thus evaluating the effectiveness of the fault detection or tolerance mechanism incorporated within the simulated circuit. Notably, gate substitution and test bench production are fully automated in a fault simulation tool presented in [114]. The designer only needs to provide the original circuit netlist and the fault list, while any commercial HDL simulator can be used for the fault experiments (e.g. ModelSim [115]).

The work of [116] concentrates on the technology-specific lowest level of the design flow and provides a "bottom-up" perspective of fault injection. Its authors conduct analogue electrical simulations of the cells within a standard gate-level cell library. They simulate both the ideal fault-free situation with the cells operating properly, and all combinations of possible manufacturing defects in the semiconductor devices that constitute the standard cells. Comparison of electrical simulation results enables the "mapping" of fault effects from the analogue to the digital domain. Accurate "mutant" standard cells in VHDL are thus made possible. These cells can subsequently be used in any standard cell level fault simulation environment (typically as in [114]).

DeLong et al [117] conduct fault injection and simulation experiments at a high level in the design flow, namely at the architectural level of a microprocessor system. Faults are injected in the internal processor buses, through VHDL *resolution functions*. Effectively, each bus is driven from two sources. The first source is the functional logic driving the bus under fault free operation, and the second is a constant logic signal, denoting a stuck-at-0/1 type fault (if it assumes the logic 0/1 value), or no fault injection (if it holds the "unknown" value x). Clearly, driving a signal from two sources results in conflicts over which value will ultimately be assumed; typically in VHDL the conflict is resolved by a suitable function (*resolution function* [48]). In this case, the resolution function consults the constant fault injection signal to determine whether the target bus is to be driven to logic 0/1 regardless of the functional driving source, or whether the fault-free scenario is in effect, wherein the bus is driven to the value dictated by the functional driving logic.

An interesting study of different HDL fault simulation approaches has recently been published in [107, 108]. Its authors identify and implement three alternative simulation strategies. In the first, they simply use *simulator commands* offered by a commercial VHDL simulator [115] to force targeted signals to desired faulty values. In the second, they add suitable *saboteur* modules at desired locations in the original system description. These modules suitably corrupt signal values, in a manner similar to the resolution functions used in [117]. Finally, the third approach considered uses *mutant* descriptions. This concept has already been encountered in [114, 116]; the authors of [107, 108] configure mutant descriptions using the (generally unpopular) *guarded blocks* VHDL construct. In brief, a VHDL guarded block is a block of statements that are only executed when a defined Boolean condition (the *guard*) is true; more details can be found in [48]. [107, 108] propose a different mutated architecture for every modelled fault in every component in the system netlist. One obvious disadvantage of this is the need of an enormous number of alternative VHDL architectures when a realistic number of faults is to be modelled. Arguably there are ways to implement fault injection based on mutants that do not suffer from this problem (using control signals as in [114], or conceptual linked lists of faults as §4.2.1 will present).

The presentation of this section has revealed that even when only HDL simulation-based fault experiments are considered, the designer of a fault testable or fault tolerant system is

presented with a number of options regarding exactly how to conduct such experiments. Firstly, a choice regarding the level at which the experiments will be carried out is required; secondly, one of three different perturbation philosophies needs to be favoured.

4.2.1 Transparent fault injection and simulation

This subsection gives a detailed presentation of a particular VHDL approach, namely the *transparent fault injection and simulation* technique developed by Zwolinski in [49, 118]. In the terminology of §4.2, the technique at hand should be classified as a member of the mutant modules based family of fault simulation approaches, and its current form is applied at the gate level. The following presentation both exemplifies the generic fundamental concepts of fault simulation described in §4.2, and stresses the specific advantages of the technique at hand. Further, §7.1.2 of this thesis will constructively utilise and extend the following material, to implement an RTL variation of the particular technique. The presentation of this subsection at times uses VHDL and "pseudo"-VHDL code segments to better illustrate the approach.

```
use std.textio.all; -- contains definition of line
package fault_inject is
  type fault_model;
  type fault_ptr is access fault_model;
  type fault_ptr_array is array (integer range <>) of fault_ptr;

  type fault_model is
    record
      fault_name : line; -- line is access string
      simulating : boolean;
      detected   : boolean;
      next_fault : fault_ptr;
    end record fault_model;

  shared variable first_fault : fault_ptr := null;
end package fault_inject;
```

Figure 4.1 : The `fault_inject` *package*

The technique firstly involves defining the `fault_inject` package of Figure 4.1. As can be seen in the figure, the `fault_model` data type defined in the package is a composite type (a *record*, similar to the record data structures found in programming languages). It contains the following four fields :

- `fault_name`, effectively a pointer (*access* in VHDL terminology) to a string holding a symbolic name for the fault
- `simulating`, a Boolean flag denoting if the fault represented by the record is injected to the circuit at a given time point
- `detected`, a second Boolean flag which should be set as soon as the fault of interest has been detected

- `next_fault`, a pointer to the next `fault_model` type record

It is through this last pointer that a linked list of `fault_model` type variables can be formed, exactly as in procedural programming languages such as C++ [119]. To enable this, a *shared variable* (equivalent to the global variable concept) named `first_fault` is also declared in the package. This variable is simply a pointer to a record of type `fault_model` and it is initialised to the `null` value.

As soon as simulation starts, the shared variable `first_fault` becomes the head of the linked list of faults. The pseudo-code of Figure 4.2 shows how this is achieved, and how faulty behaviour is imitated for the example of a two-input NAND gate. Applying fault equivalence and fault dominance principles [1] on the NAND gate shows that only three discrete faults need to be considered. These correspond to any of the inputs a, b or the output z of the gate to be stuck-at-1. Three local pointers to `fault_model` records are thus declared and initialised to `null`, one for each of the faults. At the first execution of process `nn`, new record objects are created to represent the faults, and appended at the head of the fault list, using the shared variable `first_fault`. The first part of the code of Figure 4.2 shows how this is done for variable `z_sa1` (representing the fault according to which the gate output z is stuck-at-1). Variables `a_sa1` and `b_sa1` are handled similarly. The code clearly shows that this first part becomes ineffective as soon as non-null values have been assigned to `z_sa1`, `a_sa1` and `b_sa1`, i.e. it is effective only in the first execution of the process, and its purpose is

```
library ieee;
use ieee.std_logic_1164.all;
use work.fault_inject.all;
entity nand2 is
  port (z : out std_logic; a, b : in std_logic);
end entity nand2;

architecture inject_fault of nand2 is
begin
nn : process (a, b) is
       variable z_sa1, a_sa1, b_sa1 : fault_ptr := null;
     begin
       -- first part (variable initialisation)
       if z_sa1 = null then
         z_sa1 := new fault_model'(
                  new string'(inject_fault'instance_name & "z_sa1"),
                              false, false, first_fault);
         first_fault := z_sa1;

        -- similarly for other faults

       end if;
       -- second part (functionality)
       if z_sa1.simulating then     -- z/1
         z <= '1';
       elsif a_sa1.simulating then -- a/1
         z <= not b;
       elsif b_sa1.simulating then -- b/1
         z <= not a;
       else                        -- fault-free
         z <= a nand b;
       end if;
     end process nn;
end architecture inject_fault;
```

Figure 4.2 : 2-input NAND gate with fault injection capabilities

purely the automatic formation and initialisation of a linked list of fault variables. The second part of the code corresponds to the NAND gate model functionality. A chain of `if` statements describes the alternative behaviours, depending on the experiment scenario (any of three possible fault injections or fault free operation).

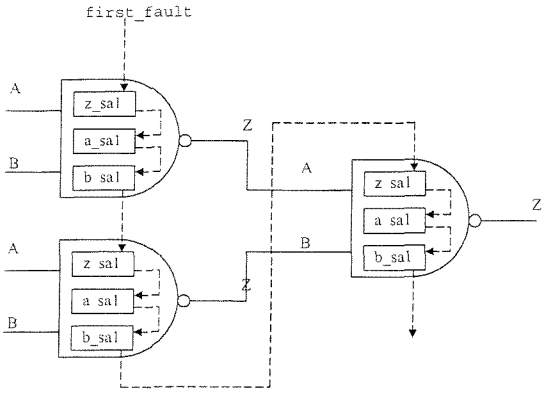


Figure 4.3 : Example netlist

Note that the VHDL model of Figure 4.2 can be considered a "mutant" NAND gate, since it demonstrates normal functionality in the fault free scenario and the appropriate faulty behaviour under the presence of a fault. Similar mutants can be written for any other elementary logic gate functionality along the lines of Figure 4.2, using the framework of package `fault_inject`. Thus a library of "fault injectable" logic gates can be developed. More complex netlists can subsequently be configured using this library. Figure 4.3 shows a simple example netlist comprising three NAND gates. Solid lines in the figure depict physical connections; in contrast, dashed lines correspond to conceptual software links, thus illustrating the fault list. All three `fault_model` objects within each gate are linked together as explained through the code of Figure 4.2; moreover, conceptual links between objects

```
library ieee;
use ieee.std_logic_1164.all, std.textio.all,
    work.fault_inject.all;
entity tb is end entity tb;

architecture fileio of tb is

  signal declarations
begin
a1: entity work.Top_level_netlist port map (association list);
p1: process is
      variable head_ptr : fault_ptr := null;
      variable fault_count, faults_detected : natural := 0;
        other auxiliary variables
    begin
        execute fault free simulation for every input in vectors.txt
        and write results with corresponding inputs in results.txt
        wait for 100 ns;
        head_ptr:=first_fault;
        while head_ptr /= null loop
        fault_count := fault_count + 1;
        head_ptr.simulating := true;
        while not endfile(results) loop
            read results.txt and apply input vector
          wait for 100 ns;
          if (output differs from that written in results.txt) then
            head_ptr.detected := true; -- fault detected
               write detection information in faults.txt
          end if;
        end loop;
        head_ptr.simulating := false;
        wait for 100 ns;
        head_ptr := head_ptr.next_fault;
      end loop;
        summarize results
       output fault coverage information in faults.txt
       i.e. faults detected / faults injected
      wait; -- halt
    end process p1;
end architecture fileio;
```

Figure 4.4 : Example testbench

in different gates are also formed as Figure 4.3 graphically shows. This is achieved automatically, since there is only one `first_fault` pointer variable, shared by all processes in all entities in the overall netlist. Automatically including all faults in the list is an advantage over the techniques described in §4.2, since no explicit list of faults needs to be provided by the designer. Another interesting observation on Figures 4.2 and 4.3 is that the physical gate interface and the connections between gate inputs and outputs are not affected by the inclusion of fault injection capabilities in the VHDL model. Indeed, the mutant model of Figure 4.2 has only two input ports a and b, and an output port z, exactly as if it was a normal NAND gate. Further, in Figure 4.3 the outputs of the first logic level are fed to the inputs of the second, exactly as if the NAND gates did not have fault injection capabilities. In essence, the structural properties of the original netlist are fully preserved when mutants replace the usual logic gates. This means that a normal netlist can readily be used for fault experiments as soon as the mutant gate library has been formed, simply by instructing the VHDL simulator to use the mutant descriptions in place of the normal ones. This is typically done in VHDL in a single line of code (*configuration specification* [48]). It follows that the technique is particularly easy to instrument and leaves large parts of the original structural VHDL netlist descriptions unaffected; this justifies the *transparent* property attributed to it.

Figure 4.4 shows a possible testbench template (in pseudo-VHDL) required to orchestrate the overall fault simulation experiment. In this particular testbench, a set of input test vectors is provided by the designer in the `vectors.txt` file. A round of fault-free simulations is initially conducted for the top-level design (`Top_level_netlist` in the figure), and the results together with the corresponding inputs from `vectors.txt` are stored in `results.txt`. Subsequently, elements in the fault list are accessed one by one, using pointer variable `head_ptr`). Each fault is simulated by having its corresponding `simulated` flag set. All test vectors are applied to the design and the responses are compared against those written in `results.txt` during fault free simulation. Whenever a mismatch is found, the fault is marked as detected and relevant detection information is output to `faults.txt`. Such information may include the symbolic name of the fault, simulation time at which it was detected, the input vector that detected it, or indeed anything else the design requires. After all faults have been simulated, some kind of summarizing information can conclude file `faults.txt`. For example, the total number of detected faults can be calculated and reported.

4.3 Summary

The fundamental concepts of fault simulation and related reliability evaluation techniques were given in this chapter. Most importantly, Zwolinski's transparent fault simulation technique was detailed. This technique will be constructively used in chapter 7 for the reliability evaluation experiments of this thesis.

Chapter 5

Datapath Self-checking Design

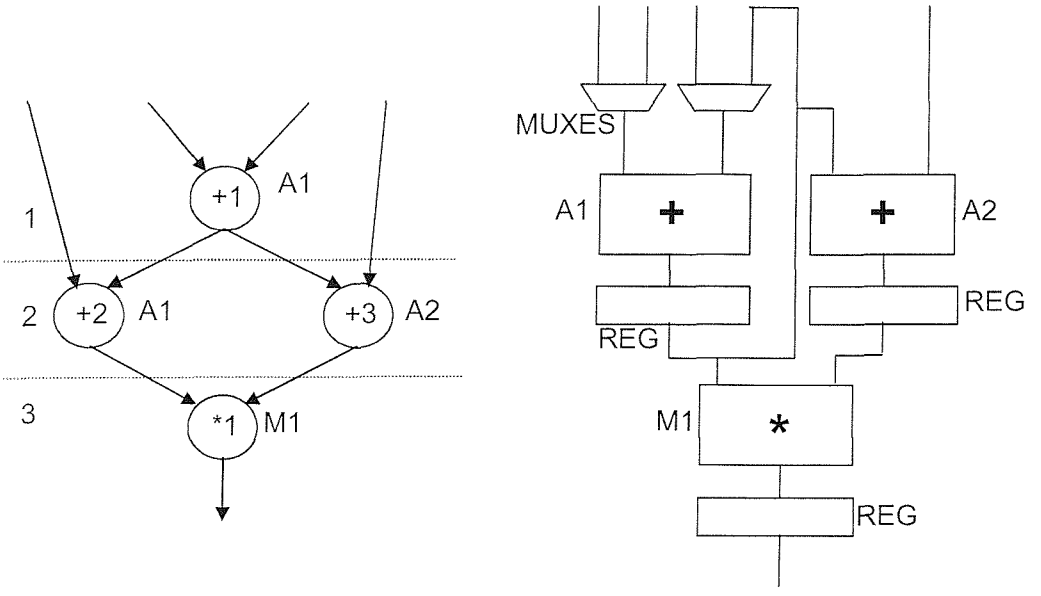
This chapter focuses on the on-line testing of the datapath part of controller / datapath designs. In the context of this thesis, such designs are considered in the form of RTL netlists, automatically generated by high-level synthesis. When such a netlist is ultimately implemented on silicon or downloaded onto an FPGA, it can normally be observed that most of the silicon area / FPGA resources are occupied by data rather than by control operations. It is therefore sensible that datapath on-line testability is the first issue to be addressed towards implementing high-level synthesis for on-line testability.

This chapter is organised as follows. Section 5.1 specifies the requirements of datapath on-line testability, revisits the families of on-line testing techniques presented in §2.2 and evaluates them in the light of the specifications of the problem at hand. Ultimately, the family of algorithmic duplication and related self-checking design techniques are chosen as the most appropriate solution. Section 5.2 elaborates more on the chosen technique in relation to background material (§2.2.2.3) and presents the *inversion* testing idea. Section 5.3 details the implementation of datapath self-checking design within the MOODS (§3.2) high-level synthesis system and presents experimental results and comparative comments. Finally, section 5.4 draws the concluding remarks of this chapter.

5.1 Problem statement and discussion of potential solutions

This section presents a discussion of requirements and potential solutions to the datapath on-line testing problem. Throughout the whole chapter, the datapath is shown either using the DFG representation (Definition 3.2) or through the actual hardware used to implement the DFG functionality, as appropriate per situation. Figure 5.1a shows a familiar simple

DFG example (also used in Figures 2.33 and 3.3), while the datapath netlist realising the DFG is depicted in Figure 5.1b. A comparative inspection of 5.1a and 5.1b establishes the correspondence between the DFG and the hardware implementing it. Indeed, adder A1 is used twice in the DFG; therefore, two multiplexers (MUXES) are used in the implementation to choose between the two possible inputs. Registers (REGs) are also employed to preserve values across DFG boundaries. Both multiplexers and registers receive control



(a) Data-flow graph *(b) Hardware netlist*

Figure 5.1. Alternative views of the datapath

signals from the controller part of the design; these signals act as "select" and "load enable" inputs respectively (§3.2.6); thus the correct timing is in effect ensured. The controller and the mentioned signals are omitted in the figure, for clarity. The problem addressed in this chapter is to augment datapaths of the form of Figure 5.1, in such a way as to enable the user to have an on-line indication of the health of the system and a timely report of any hardware failure. Further, the insertion of the resources necessary for this additional functionality has to be done within the high-level synthesis process, concurrently with the rest of the synthesis tasks (§3.1.2), and as transparently to the designer as possible. This last proposition primarily means that the synthesis tool should be able to generate on-line testable systems at the designer's request, without requiring *any* modification of the original HDL description of the system behaviour (along the lines of Figure 3.2).

5.1.1 Problem requirements

In order to choose an *on-line* testing solution for the datapath of a complex design, to be realised by a *behavioural synthesis tool*, one naturally has to take into account precisely the particular characteristics of these two concepts, in addition to the usual performance and cost specifications.

More specifically, when a system is on-line, it is desirable that any fault corrupting its operation be detected as soon as possible, so that any existing recovery mechanism can be triggered (low error latency). At times, short-lived faults develop into the system but do not manifest themselves at the outputs of the system, because they happen not to be sensitised by the functional input vector applied to the system throughout their lifetime. These fault are termed *latent faults*. An *on-line* testing solution need not target latent faults. In fact, detecting a latent fault and taking corrective action typically involves performance degradation; since the system is on-line and producing useful output, it is preferable to avoid such degradation unless absolutely necessary (i.e. unless a fault manifests itself by corrupting logic values). Therefore, the approach taken in this thesis is that latent faults *should not* be detected in the on-line context, so that undesirable "false alarms" will be avoided.

Addressing the whole problem at the behavioural synthesis level has its own implications. Firstly, just as a behavioural synthesis tool should understand and synthesize as broad a range of HDL descriptions and design styles as possible, so should a "behavioural synthesis for on-line testability" tool be able to generate acceptable testing solutions for as wide a class of designs as possible. This suggests that the adopted testing technique should be generically applicable rather than application-specific. Further, recall that the high-level synthesis process as such is largely independent of the target technology, while its output, being an RTL netlist, is still relatively high in the design flow and does not necessarily restrict the lower-level tools to a particular gate level structure of the RTL building blocks (§3.1). A testing strategy maintaining these benefits should therefore not depend on the target technology or gate level structure. Another important benefit of high-level synthesis is the interaction between the designer and the tool, through the cost requirements of a given project. Recall, for example, that the MOODS cost function of §3.2.4 affects the choice of optimisation algorithm, by determining the particular incarnation of the tailored

heuristic to be used (§3.2.5.2). Any testing technique considered in this context should be able to take advantage of this versatility.

In summary, before choosing any of the techniques presented in chapter 2 for implementation within synthesis, the said techniques need to be evaluated based on the following criteria.

(a) error latency

(b) avoidance of "false alarms"

(c) general applicability, including independence of low-level structure and target technology

(d) ability to take advantage of high-level synthesis versatile design space exploration

Efficiency in terms of area overhead and time penalty is, of course, an important issue not included in the above points. The approach of this thesis is to pursue efficiency by exploiting any area and performance optimisation techniques already existing in the high-level synthesis tool of interest (as will be seen in §5.3.3.2), rather than addressing efficiency through an appropriate choice of technique. It should be borne in mind that this work addresses *tool development* rather than design *case studies*. It is therefore important for a tool to be generic (requirement (c) above), even if some of the solutions it provides may be less efficient than manually derived, application-specific ones.

5.1.2 Evaluation

The general families of on-line testing approaches of chapter 2 are considered here, as potential datapath on-line testing solutions. Self-checking design, based both on general EDCs (§2.2.1) and on duplication-related techniques (§2.2.2), on-line BIST (§2.2.3.2), shift-based on-line DFT (§2.2.3.3), and analogue characteristic monitoring (§2.2.4) are all included in the discussion. Special attention is paid to requirements (a) – (d) of §5.1.1.

Error-detecting codes (§2.2.1) could be utilised in a high-level synthesis design flow, by analysing all RTL cells that consist the tool cell library (§3.1, §3.2.7), and defining self-checking versions of them, that can be further included in the cell library, together with appropriate checkers. Referring back to Figure 5.1, the self-checking design of a datapath would then involve the utilisation of the self-checking versions of all datapath modules,

e.g. adder A1 in the figure would be realised by a self-checking adder incorporating an appropriate checker, multiplier M1 would also need to be a self-checking multiplier etc. Such an approach could easily use data from the literature. For instance, recall that [16, 15, 41] presented self-checking ripple-carry addition based on parity checking, as well as self-checking multiplication based both on parity and on arithmetic codes (§2.2.1.1, §2.2.1.5). Registers could also employ parity checking or even support error correction (§2.2.1.4, [36]). It can be observed that this solution has no "active" interaction with the high-level synthesis process, in the sense that it only deals with the cell library and final operation binding, but does not interfere with the scheduling and allocation phases. In other words, it cannot take full advantage of the versatile high-level synthesis optimisation. Further, it necessarily requires some degradation in the maximum achievable clock speed; indeed, all operations in Figure 5.1 would include a certain invariant property checking, thus made slower. On the other hand, no false alarms could normally be produced by a self-checking system, while error latency would be minimal, since any logic error would be detected in the clock cycle it manifested itself. However, a property of self-checking design that is actually a disadvantage in the context of high-level synthesis is its total dependence on and

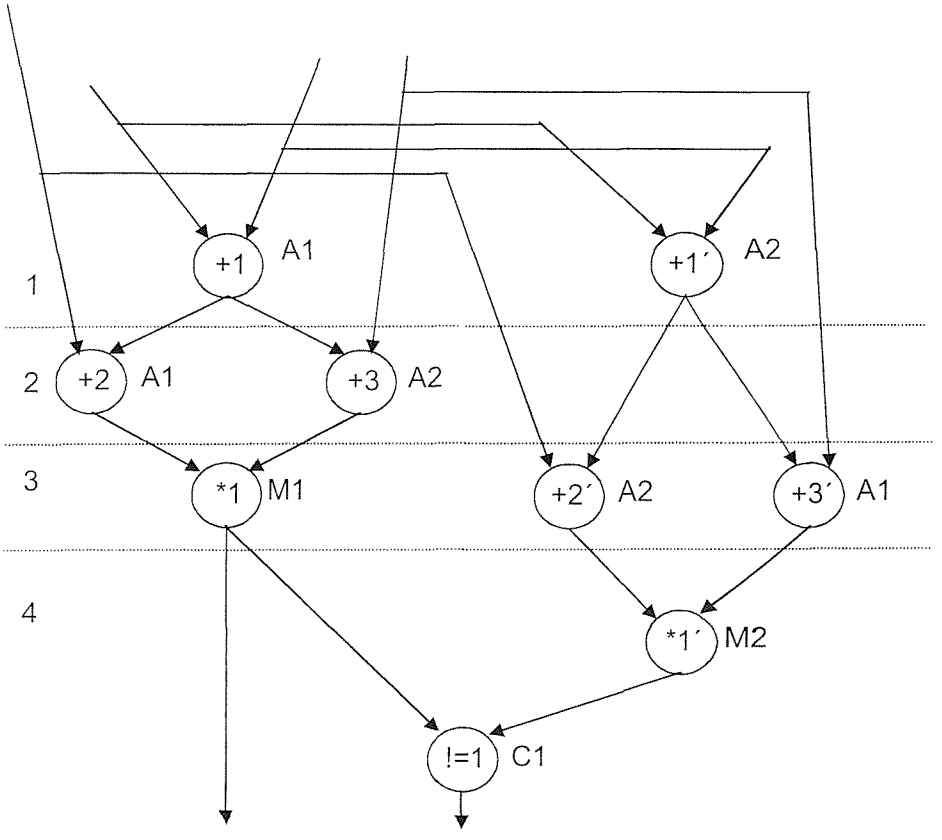


Figure 5.2. Self-checking design based on algorithmic duplication

intervention with the gate-level structure of the circuit under check (see for example Figure 2.10). In short, with respect to points (a) – (d), EDC-based self-checking satisfies (a) and (b), but lacks (c) and (d).

Self-checking design based on algorithmic duplication and related techniques can provide an interesting alternative. Figure 5.2 shows how it can be applied to the simple DFG example of Figure 5.1a. Operations +1′, +2′, +3′, and *1′ in the figure denote the duplicates of the respective functional operations, while !=1 is a comparison operation, implemented by the newly introduced fault secure comparator module C1. In line with §2.2.2.3, a functional and a duplicate operation are never implemented by the same hardware module. The scheme as presented in Figure 5.2 experiences a delay degradation of a clock cycle, and it may also experience an error latency of a few clock cycles. For instance, if adder A1 is faulty and produces a failure in addition +1 during CS 1, the failure will not be detected before CS 4. Further, the chaining (§3.2.3) of comparison !=1 after multiplication *1′ within CS 4 will probably lead to clock speed degradation. A remedy to the fault latency problem could be the introduction, scheduling and allocation of multiple comparison operations at intermediate points in the DFG (§2.2.2.3), while better clock speed could be achieved by further accepting an additional fifth control step and scheduling the final comparison there. From the above it is evident that the considerations and trade-offs associated with algorithmic duplication have direct relevance to the high-level synthesis design space exploration tasks (allocation, scheduling). The whole problem can therefore ideally be formulated within the core of the synthesis process. An additional strong point is that the scheme of Figure 5.2 is purely generic and behavioural, in that it makes no assumption about either the gate-level structure of the modules realising the system functionality, or about the overall functionality as such, or even about the target technology. It can therefore be stated that algorithmic duplication and related schemes *both* retain the benefits of EDC-based self-checking design *and* fit well into the behavioural synthesis context.

An additional benefit of a "behavioural" self-checking scheme such as algorithmic duplication, in the context of CAD tool development, is its natural *support for future expansions*. Consider a given synthesis tool, and an associated cell library. Assume that only one cell of a particular functionality is available in the library, for instance only one type of adder such as a ripple-carry adder. If EDC-based self-checking is desired, then the library will also include a self-checking version of the adder, as explained above. If a structurally

alternative cell implementing the *same* behaviour is added to the library during a subsequent development phase, then clearly the new cell will need to be analysed, for instance along the lines of [15], and its self-checking version developed from the beginning. In the adder example mentioned above, such a new cell could be a carry look-ahead adder. As [16, 15, 41] have shown, the development of a self-checking version will require a considerable amount of analysis and logic level design work. In contrast, algorithmic duplication, being a naturally behavioural technique, will readily lend itself to future tool expansions. In the running adder scenario and referring to Figure 5.2, either of adders A1 and A2 could be of any structure. The structure itself is chosen during the binding phase (§3.1.1) of high-level synthesis, and the self-checking scheme is valid in any case.

On-line BIST during idle cycles, as explained in §2.2.3.2, is another potential solution. The concept of a TDFG (Figure 2.37) associated with a given DFG initially gives the impression that the approach is very relevant to synthesis. However, as §2.2.3.2.3 already pointed out, low test quality can be a real problem with TDFGs. Further, test quality as well as test length highly depend on the gate-level structure of the circuit constituent blocks (§2.2.3.2.2); therefore, the approach is not generic enough. Finally, the error indication itself that BIST provides is of doubtful usefulness in the on-line testing context. To understand this, refer back to the example TDFG configuration of Figure 2.37. Putting aside the test quality considerations, assume that the TPG provides all of its test vectors in k executions of the functional circuit. An erroneous signature in the MISR after the k executions provides the error indication. However, this indication does not specify *which* of the k functional results produced by the circuit was corrupted. In fact, it is likely that by the time the MISR detects the fault, the fault will have propagated to other parts of the overall system, probably with catastrophic effects. In other words, on-line BIST experiences *unpredictable* error latency. It is also possible that the MISR has detected a latent fault, thus leading to a false alarm. In summary, none of requirements (a) – (c) is satisfied.

At this point, it can be mentioned that the on-line BIST evaluation of the previous paragraph is equally applicable to on-line arithmetic BIST (§2.2.3.4.1), the latter in essence being a form of BIST with a certain non-standard implementation (i.e. using accumulators instead of LFSRs). In fact, it can be expected that arithmetic BIST will have even more restricted applicability, since it cannot accommodate designs in which too few adder – register pairs can be configured.

On-line shift-based DFT (§2.2.3.3) is discussed in the following. In this family of techniques, the inputs and outputs of selected operations are shifted out of the chip and tested by an external testing unit, which effectively repeats the operation. A mismatch between a shifted output and an output produced by the testing unit signifies the presence of a fault. On one hand, the scheduling and choice of operations that will have their inputs and outputs shifted out can be formulated as a synthesis task. Further, there is no obvious danger of a false alarm. On the other hand, however, error latency is unpredictable and uncontrollable. Even further, there are serious concerns regarding the practicality of implementing concurrent shift-based testing. In particular, shifting out a number of variables while the system is operating would involve an additional shift clock. If realistic bit-width values are considered, this clock would need to be tens of times faster than the functional clock, so that a number of variables can be shifted out during a single cycle of the functional clock [83]. This will limit the scope of the technique to very low speed applications. Moreover, the idea itself of utilising an external testing unit for concurrent testing is of doubtful practicality, since such a unit will need to be compact enough to accompany the chip on the field. Furthermore, if reliable testing is desired, then the testing unit as such will need to be designed using some on-line testability strategy, further complicating the problem. The above critical remarks are backed by the absence of convincing experimental results in the relevant publications [83, 84, 85, 86]. In summary, while the idea of shift-based on-line DFT is likely to satisfy requirements (b) and (d), it is also likely to experience high error latency (requirement (a)). Most importantly, general applicability (requirement (c)) is not guaranteed; as a matter of fact, there is not enough evidence that even partial applicability is feasible.

Let us now focus on the family of techniques labelled as monitoring analogue characteristics (§2.2.4). Such solutions detect faults through abnormalities in their electrical properties, sometimes even before the corruption of logic values. This is an interesting advantage as regards error latency, although it can be stated that alarms will rise even if logic values are not corrupted. The strongest argument against them, however, is that they are only relevant to the target technology (e.g. abnormal flow of current can *only* be defined with respect to the technology), and by nature address the on-line testing problem at a very low level in the design flow. Therefore they are neither generally applicable nor related to the behavioural HDL level of abstraction, thus not fitting the perspective of the present thesis.

The background presentation of chapter 2 also includes analytical techniques (§2.2.3.4.2). These are purely application-specific, thus not satisfying the critical general applicability requirement (c) of §5.1.1.

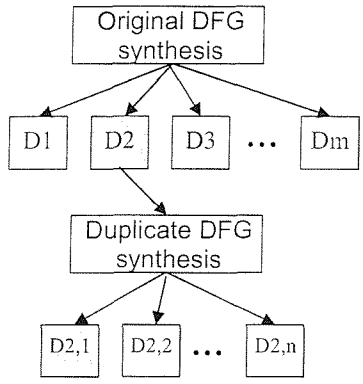
The detailed evaluation of this section establishes that algorithmic duplication related, "behavioural self-checking" techniques are the most suitable for implementation within a high-level synthesis environment.

5.2 Detailed presentation of proposed technique

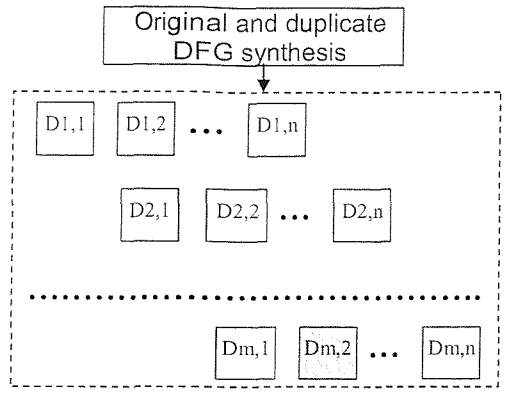
Section 5.1 justified why algorithmic duplication related techniques should form the basis of a datapath self-checking solution in the context of this thesis. However, subsection 2.2.2.3 has presented a significant number of algorithmic duplication choices. These choices vary both as regards their self-checking related properties (e.g. error latency, potential fault escapes etc.) and as regards their implementation details (e.g. at which level of abstraction testing resources are inserted and exactly how this is done). The following subsection 5.2.1 critically evaluates the techniques of §2.2.2.3, identifies strengths and weaknesses, and defines concepts not adequately covered by them. Subsection 5.2.2 proposes a variant of duplication testing (namely inversion testing) and shows its potential usefulness within DFGs. Subsection 5.2.3 summarises the conclusions of §5.2.1 and §5.2.2, and defines the goals of the algorithmic duplication-based datapath self-checking implementation, to be presented later in §5.3.

5.2.1 Algorithmic duplication revisited

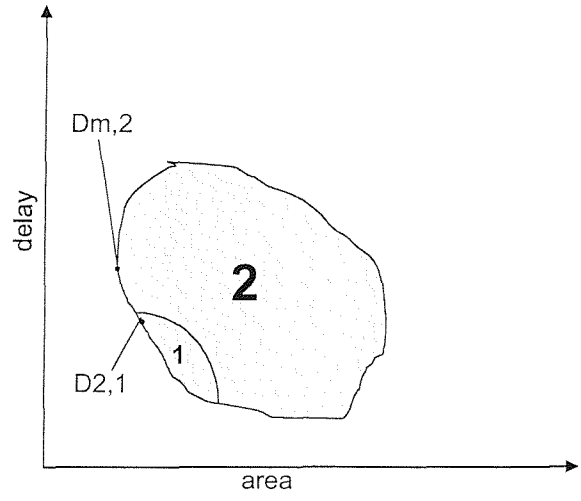
The first pieces of published research work with reference to a variant of algorithmic duplication were the ones advocating checkpointing, rollback and recomputation as means of recovery from transient faults [60, 61]. Regarding its fault handling characteristics, the idea of rollback and recomputation can lead to deadlocks if a permanent fault appears in the system. Further, error latency is not considered and not identified as a design goal. Regarding the duplicate DFG synthesis approach proposed in [60], it can be observed that the presented algorithm receives the fully scheduled original DFG as an input. From the optimisation point of view (§3.1.2), this is a disadvantage, since a significant area of the over-



(a) Synthesis of the original DFG followed by synthesis of the duplicate DFG



(b) The original and the duplicate DFGs synthesized in the same optimisation process



(c) Design space point of view

Figure 5.3. Design Space Exploration for the original and the duplicate DFG

all system design space remains out of reach. Figure 5.3 illustrates and clarifies this idea. In Figure 5.3a, a synthesis process is applied to the original DFG, a total number of m different design space points D1-Dm are visited, and the example point D2 is highlighted as the most favourable. The duplicate DFG is independently synthesised next, n candidate designs D2,1-D2,n are identified for the overall system and the example point D2,1 is chosen. Clearly, only n candidate choices are considered for the overall synthesis solution. Now focus on Figure 5.3b, where the two DFGs are optimised simultaneously. The dashed rectangle in the figure includes all possible overall design choices, corresponding to the combination of all choices for the original and duplicate DFGs (D1,1-Dm,n). As the figure depicts, all n×m possible design space points are now considered for the overall design, by

synthesizing the two DFGs in the same optimisation process, effectively treating them as *one* DFG. This is a much preferable synthesis path, as illustrated by the fact that, although D2 in Figure 5.3a is the best choice for the original DFG *in isolation*, there is no reason why an initially suboptimal solution Di with i≠2 cannot yield an overall better solution for the final system. Indeed, Figure 5.3b exemplifies such a scenario, by highlighting design Dm,2 as the most favourable out of all n×m choices. Observe that point Dm,2 cannot even be reached by the process of Figure 5.3a. An alternative view of this concept is the design space graph of Figure 5.3c. The coloured area in the graph corresponds to the overall self-checking system achievable region (§3.1.2). The region explored when the original DFG is fixed at the D2 design choice is marked as region 1, while the rest of the achievable design space area is called region 2. If the original DFG is synthesized first and fixed at D2 in the manner of 5.3a, then only region 1 will be visited by the overall synthesis process. In contrast, if both the original and the duplicate DFG are optimised simultaneously as in 5.3b, then all of the coloured area (regions 1 *and* 2) will be explored.

An additional weakness of [60, 61] is that they do not address loop and conditional constructs in the designs they synthesise, thus restricting the usefulness of the technique.

The next family of techniques covered in §2.2.2.3 are [62, 63], proposing fault identification through functional unit differentiation. An initial comment that can be given regarding the differentiation idea is that it is expected to work under the assumption that faulty units *never* or *rarely* mask faults. Indeed, consider once more the simple differentiation example given in §2.2.2.3, summarized in Table 5.1. Units A, B and C are pair-wise differentiated. For example, consider A and C. Track (1,1′) utilises A but not C; while track (3,3′) utilises C but not A. Tracks of functional and redundant computations with differentiation properties can also be noticed if one considers either of the remaining pairs of units (A,B and B,C). Theoretically, if A experiences a fault, track (3,3′) will be fault-free, while both (1,1′) and (2,2′) will signal faults, thus identifying A as faulty. However, depending on the inputs that unit A is fed with, it is entirely possible that either (1,1′) or (2,2′) will experience a fault masking event. This will result in a single fault indication, making fault identification impossi-

Computation	Unit
1	A
1′	B
2	C
2′	A
3	B
3′	C

Table 5.1. Example of unit differentiation

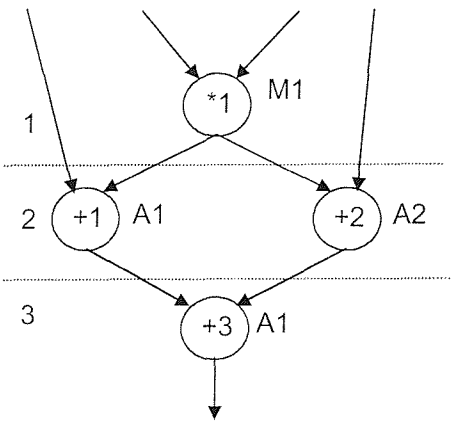
ble. Fault simulation experiments would be needed in [62, 63] to estimate how serious this problem could be; such experiments are however missing, and the differentiation technique remains of unproven, questionable practicality. As regards the synthesis approach of [62], design space exploration is more complete than in [60], since both functional and redundant computations are considered as constituting one DFG. However, loops and conditional branches are still not accommodated. Further, the synthesis cost parameters are only given in terms of number of functional units used and clock cycles needed; thus, important information such as the relative area cost of particular units in certain technologies, and the clock speed are missing. In principle, omitting this information can lead the synthesis process towards wrong decisions; in [63], this concern is confirmed by the fact that the experimental results report a hardware overhead of 100% (equivalent to *physical* duplication).

The Introspection technique of [64] fully utilises any existing module idle time, but is by nature unable to cope with cases where there is too little idle time, since it totally rejects redundant module insertion. In that sense, it is case-specific rather than generic. As covered in §5.1.1, this is not consistent with the philosophy of the present thesis. From the synthesis point of view, an interesting binding algorithm is outlined in [64]; however, the algorithm input is a fully scheduled DFG (as in [60]). As a result, the design space is not explored efficiently (as illustrated previously in Figure 5.3). Finally, loops and conditionals are not explicitly addressed here either.

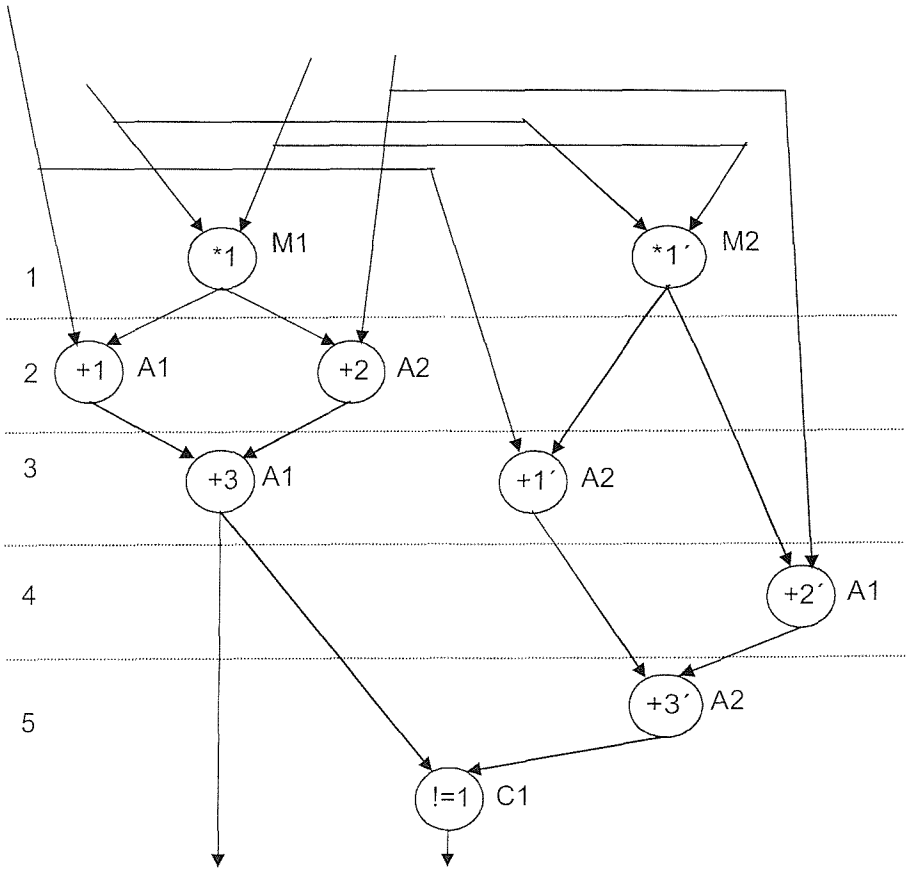
The next scheme presented in §2.2.2.3 is the behavioural synthesis of fault secure systems of [23]. It is probably the most complete of the algorithmic duplication approaches; however, a number of weaknesses can be spotted in it as well. The synthesis process starts with a scheduled DFG, followed by a full physical duplication and comparison of the primary output results of the circuit of interest. Comparisons of selected intermediate results (e.g. the results of additions +1 and +1′ in Figure 5.2) are introduced under certain conditions, particularly when the fault study in [23] suggests that such a comparison promotes hardware sharing between the original and the duplicate DFG, while keeping the probability of fault escapes below a defined threshold. A weakness here is that all comparisons, including those of intermediate results, take place at a dedicated control step, after the execution of the functional circuit has finished (e.g. after CS 3 of Figure 5.2). One can then understand that designs with realistically long critical paths will experience high error

latency (possibly of the order of the critical path length). In that sense, this self-checking approach is efficient and suitable for an *a posteriori* validation of the obtained result, *but* unsuitable for the pre-emptive indication needed in safety critical applications, so as to trigger any existing recovery or self-repairing mechanism. Further, the authors of [23] are the first to mention that fault secure comparators (§2.2.2.1) are needed in algorithmic duplication applications, and therefore assume that their comparators are such. However, they do not elaborate on the actual comparator structure to ensure this property. As regards the synthesis approach they use, one can observe that they feed their algorithm with a scheduled and bound DFG. Their subsequent self-checking synthesis steps are in fact allowed to make slight changes to the original DFG scheduling; this is an improvement in terms of design space exploration with respect to [60, 62, 64, 61, 63], but the allowed changes are indeed very limited, only applicable under the strict condition that they lead to an immediate improvement. An additional improvement of [23] over [60, 62, 64, 61, 63] is the ability to handle loop constructs in designs. Conditional branches are, however, still, not accommodated; in fact, this author thinks that the approach of [23] is particularly unsuitable for conditionals, since it very much relies on analytical calculations of fault escape probabilities. Conditional branches would make the calculations very complicated because the probability of visiting or not visiting a particular DFG node would need to be taken into account when calculating the probability of fault escapes.

Semiconcurrent error detection [65] is considered next. The evaluation is illustrated by the example of Figures 5.4 and 5.5. Figure 5.4a shows a simple DFG, comprising 1 multiplication and 3 additions and having a critical path length of 3 control steps. In 5.4b, a possible algorithmic duplication solution is shown. Only the final primary output results are compared in the presented scenario. Further, the example solution has been configured such that no new hardware modules are added; a delay degradation of 2 clock cycles (66.6%) is accepted instead. Figure 5.5 shows a semiconcurrent error detection solution for the same example, with checking periodicity P=2 (P has been defined in §2.2.2.3). The primary inputs and outputs in Figure 5.5 are exactly as in Figure 5.4 (e.g. addition +1 is fed by a primary input), but are not explicitly shown in order not to overload the figure. The configuration graphically depicts that semiconcurrent error detection sacrifices some testability for area and / or delay savings. Indeed, in Figure 5.5 two executions of the functionality of 5.4a are conducted with a nominal latency of 3 control steps each; only one of them needs to be checked, because P=2. This means that the duplicate DFG has a very



(a) Original DFG



(b) An algorithmic duplication solution, with primary output comparison only

Figure 5.4. Example DFG

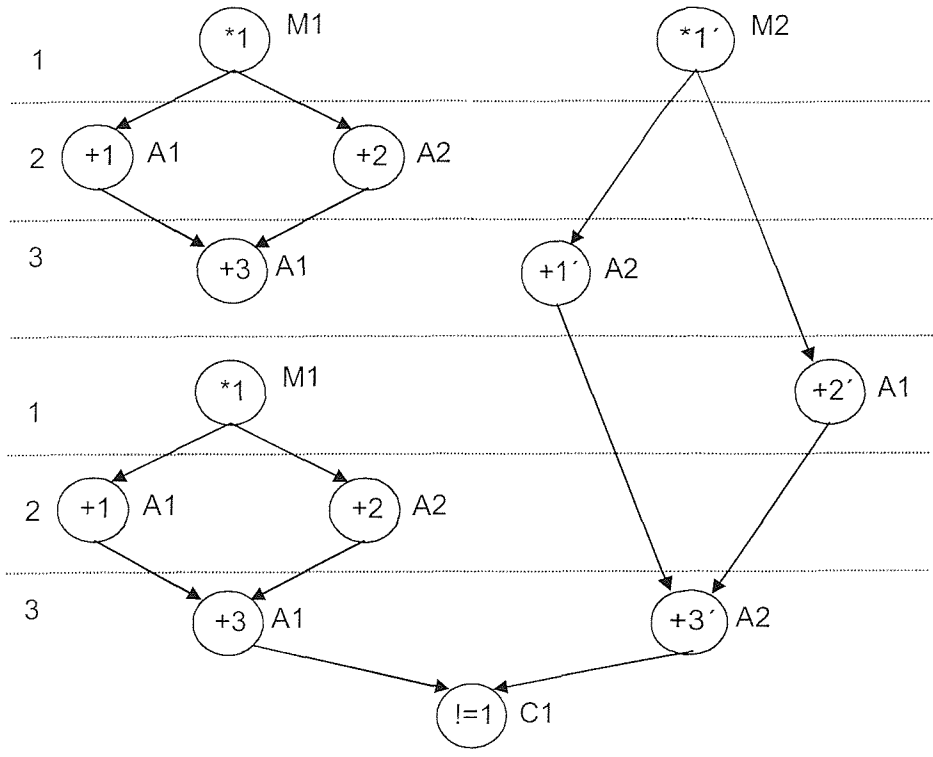


Figure 5.5. Semiconcurrent error detection solution for the example of Figure 5.4 (checking periodicity P=2)

relaxed delay specification of 6 control steps. It is easily scheduled within these 6 steps, and it does not require any additional hardware modules. This way, a low-cost self-checking solution is instrumented; the quality of test, however, is highly degraded. Indeed, consider a DFG with a realistically long critical path, and / or $P>>2$. An error indication at the output will simply signify that there is *a certain* malfunction in the chip; it will not determine *when* the fault first appeared, or *how many* of the P executions have been affected by the fault. Clearly, there is both unpredictable error latency and uncertainty as to the magnitude of the effect of a given detected fault. In line with [23], semiconcurrent error detection is suitable for a theoretically inexpensive but limited periodic checking of the health of the system, possibly to detect non-fatal malfunctions; it is unsuitable for pre-emptive error checking in safety-critical applications. Regarding the synthesis characteristics of [65], a set of constructive (§3.1.2) synthesis algorithms is given for the scheduling of the duplicate, relaxed-latency DFG, given the original, scheduled and bound, functional DFG. The approach suffers from the poor design space exploration problem explained in Figure 5.3. On a positive note, extended versions of the algorithms are also given, accom-

modating both conditional branches and loop structures in the DFG, for the first time. The algorithms are said to be under inclusion in an experimental integral synthesis tool; however, no results from this tool are given. The experimental results of [65] have been obtained by commercial synthesis tools. From the information provided in [65], one concludes that this was done by modifying the original HDL descriptions of the considered designs, and implementing script-based scheduling and binding of the duplicate DFGs on commercial CAD tools, in effect *manually* applying the presented constructive algorithms. From the point of view of this thesis, this requirement for substantial designer intervention is a serious disadvantage.

The research of [69, 70] proposes two diverse realisations of the same DFG. The two versions are differentiated from each other either because of different allocation of operations to operators or because of the recomputation with shifted operands applied in [70] (§2.2.2.3). The first realisation is executed P times; then the second is executed once, and thus the last functional result is verified by comparison. It is evident that once again only 1 out of P obtained results is verified; therefore the technique suffers from potentially high error latency and uncertainty exactly as explained above for the semiconcurrent solution. It is therefore again unsuitable for safety critical systems. From the synthesis point of view, the approach of [69, 70] is fully manual and there is no mention of any design automation attempt. In that sense, these works are not relevant to the goal of the present thesis, since they address the whole problem at a lower level of abstraction.

Finally, [66, 67, 68] propose constructive algorithms for the configuration of duplication and comparison schemes (as of Figure 5.2). They do not offer anything theoretically novel with respect e.g. to [23]; they only compare final results, thus being unsuitable for preemptive self-checking; furthermore, they are also manual RT-level approaches, therefore concepts such as behavioural design space exploration are not applicable in them.

5.2.2 Inversion testing

Figure 5.6 depicts the *inversion testing* paradigm. The figure accurately follows the nomenclature of Figure 2.27 (duplication testing). Indeed, a redundant circuit is again added to the functional one, and a checker / fault secure comparator (§2.2.2.1) is employed to signify the potential presence of a fault. The difference with duplication is that the redun-

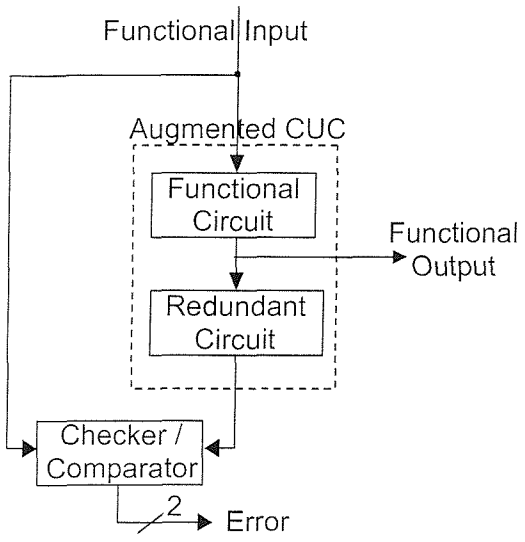


Figure 5.6. Inversion Testing

dant circuit in the inversion case is no longer a replica of the functional one. Rather, it is a circuit that reproduces the original functional input, when suitably fed by the functional output. Clearly, this means that the flow of data throughout the scheme should be as Figure 5.6 shows, i.e. the inverse operation should take place after the functional one, rather than in parallel (compare to Figure 2.27). Other than that, the redundant circuit has to be of approximately the same size as the functional one. This proposition is historically backed by the theoretical fault detection study of [120], analytically proving that for any given system under check, the "detection" logic added to it should be at least as complex as the system itself, if an unrestricted fault model is adopted (i.e. if all possible faults are targeted). In that sense, inversion testing can be considered a member of the family of duplication-related techniques, as loosely defined in §2.2.2.

Clearly, inversion cannot be applied to any arbitrary function. One can think of several examples where there can be no redundant circuit that uniquely reproduces the original inputs, when fed by the functional outputs. Logic functions (AND, OR) are such non-invertible examples. An arithmetic example is the square - square root pair, which is not *uniquely* invertible for signed arithmetic. However, when a unique inverse for the functional output exists, then the scheme is fault secure. Figure 5.7 exemplifies the inversion testing idea and demonstrates fault security, through the simple addition – subtraction pair. In the figure, let a and b be signals of bit-width n (e.g. n=8 or n=16), corresponding to arithmetic values. Signal c equals a+b, while

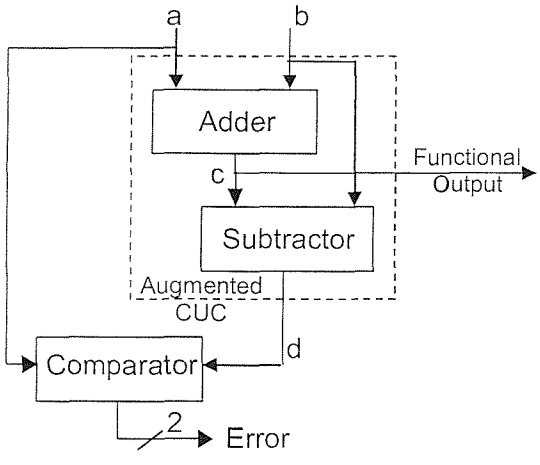


Figure 5.7. Inverting an addition

likewise d is equal to c-b. Basic arithmetic suggests that in the fault free case, signal d will always be equal to input a, and the comparator will verify the correct operation. Any single fault manifesting itself at the output of the adder will result in a corrupted value c′ with c′≠a+b. Due to the 1-to-1 property of subtraction, the subtractor output will now be d′=c′-b≠a and the comparator will detect the fault. Alternatively, if a hardware fault corrupts the operation of the subtractor when it is fed by correct inputs, changing the output to d′′≠c-b, then the comparator will once more be fed by unequal values and detect the fault. Finally, any manifested single comparator fault will clearly result in an error indication, so long as the comparator has been designed to be 2-bit output fault-secure, on the principles of self-checking design (§2.2.2.1). It is thus evident that the scheme is fault secure with respect to single faults, since any non-latent single fault in any part of the scheme will result in an error indication. It should again be stressed that this is clearly a result of the 1-to-1 property of the considered arithmetic functions. It is only under this condition that fault security is guaranteed and only under this condition that inversion defines a valid alternative to duplication.

Simple visual inspection of Figures 2.27 and 5.6 immediately gives rise to the issue of whether inversion can be a *beneficial* choice over duplication. An initial remark is that physically inverting a circuit is expected to be approximately as expensive as physically duplicating it, since the redundant inversion circuit is expected to be at least of the size of the functional one [120]. Further, inversion will be considerably slower, since in Figure 5.6 the functional output is verified after both the redundant circuit and the comparator have performed their operation. In contrast, in the duplication testing of Figure 2.27, the redundant circuit operation is performed concurrently with the functional operation. It can therefore be stated that, even when an inverse function exists and leads to a fault secure scheme, physical inversion of isolated circuits has *no* advantage over physical duplication, and is therefore of no interest.

Inversion becomes interesting only in the context of substantially-sized sequential systems. This is illustrated in the following example. Figure 5.8 depicts a possible DFG realisation of the *Tseng* design. This design was introduced in [121] and has ever since been widely used in the high-level synthesis community for benchmarking purposes. Although not corresponding to any useful functionality, its form is regarded as highly representative of situations typically encountered in high-level synthesis. Hence it is an instructive and

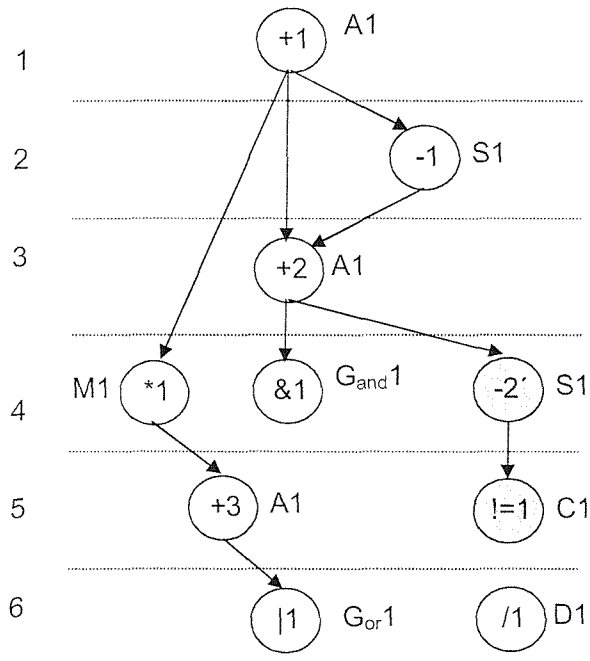


Figure 5.8. Algorithmic inversion for an example DFG (Tseng benchmark)

useful example. Temporarily omitting the two highlighted operations, one can observe that the design includes three additions (+1, +2, +3), one subtraction (-1), one multiplication (*1), one division (/1), as well as two logic functions, a bit-wise AND (&1) and a bit-wise OR (|1). In the present realisation, these operations are scheduled in a total of 6 control steps as shown, and allocated to an adder A1, a subtractor S1, a multiplier M1, a divider D1, an array of AND gates $G_{and}1$ and an array of OR gates $G_{or}1$. All operations have two functional inputs; however, in line with the previous Figure 5.5, several inputs are omitted in Figure 5.8 for the sake of clarity. All inputs and outputs that define internal data dependencies are clearly depicted by arcs, as usual (Appendix B includes a complete VHDL description of the Tseng benchmark). Let us now focus on operation +2, and assume that a self-checking scheme is required for this addition alone. Since there is only one functional adder in the design, applying duplication testing would necessarily result in the introduction of a new adder A2 together with a new comparator C1. If inversion is applied instead, then a self-checking solution for +2 could be configured as the figure shows, by introducing the two highlighted operations. Subtraction -2′ inverts addition +2 using the existing subtractor S1, which is idle during CS 4. Further, the necessary comparison !=1 is conducted during CS 5, on the newly introduced comparator C1. This way, operation +2 is checked by means of *algorithmic inversion* (inversion testing that does *not necessarily* involve physical introduction of a new "redundant" module). With respect to duplication, it is evident that in this particular case algorithmic inversion saves the hardware cost of an adder module. Referring back to Figure 5.8, one can observe that alternative inversion solutions could be considered by moving operations -2′ and / or !=1 in time. For example, the comparison could be moved to control step 4 and chained (§3.2.3) after the subtraction. This

would reduce error latency to a single control step, while it might not require clock speed degradation, since the (probably slow) multiplication *1 is already present in CS 4. Of course, this cannot be determined conceptually here; low-level implementations and the target technology need to be taken into account. In a high-level synthesis environment, such information is readily available in or can easily be calculated from the technology library (§3.1).

In summary, the above example points out that, in DFGs of substantial sizes, there can be cases when algorithmic inversion provides an interesting and beneficial alternative to algorithmic duplication. In that sense, it should be kept as an *additional degree of freedom* when devising self-checking DFGs. The example also shows that the whole problem with all of its parameters and trade-offs is best addressed at the behavioural synthesis level of abstraction.

Other than the historical theoretical study of [120] mentioned above, one can also find two recent publications proposing schemes that remind of inversion self-checking as shown here. In [122], an encoder (compressor) - decoder (decompressor) pair is used for testing purposes in a dependable computing architecture, while in [123] decryption ("inverting") is applied to encrypted data, in order to detect faults in a certain hardware implementation of a cryptographic application. Still, properly defining, analysing and considering inversion in the context of self-checking DFGs, within high-level synthesis, is a novelty and one of the contributions of the research presented in this thesis.

5.2.3 Discussion

Subsection 5.2.1 evaluated algorithmic duplication techniques found in the literature and identified concepts not adequately addressed by them, not simultaneously addressed by them, or at times not addressed by them at all. Subsection 5.2.2 defined inversion and algorithmic inversion. The datapath self-checking design work of this thesis covers the issues left open by previous researchers, while exploiting algorithmic inversion, where it is beneficial. To this end, the goals and properties of the implementation presented in this thesis can be categorized with respect to the following three criteria :

- Fault recovery. Past attempts at fault recovery have yielded application-specific and unproven recovery mechanisms (§5.2.1). In principle, any adopted recovery mechanism

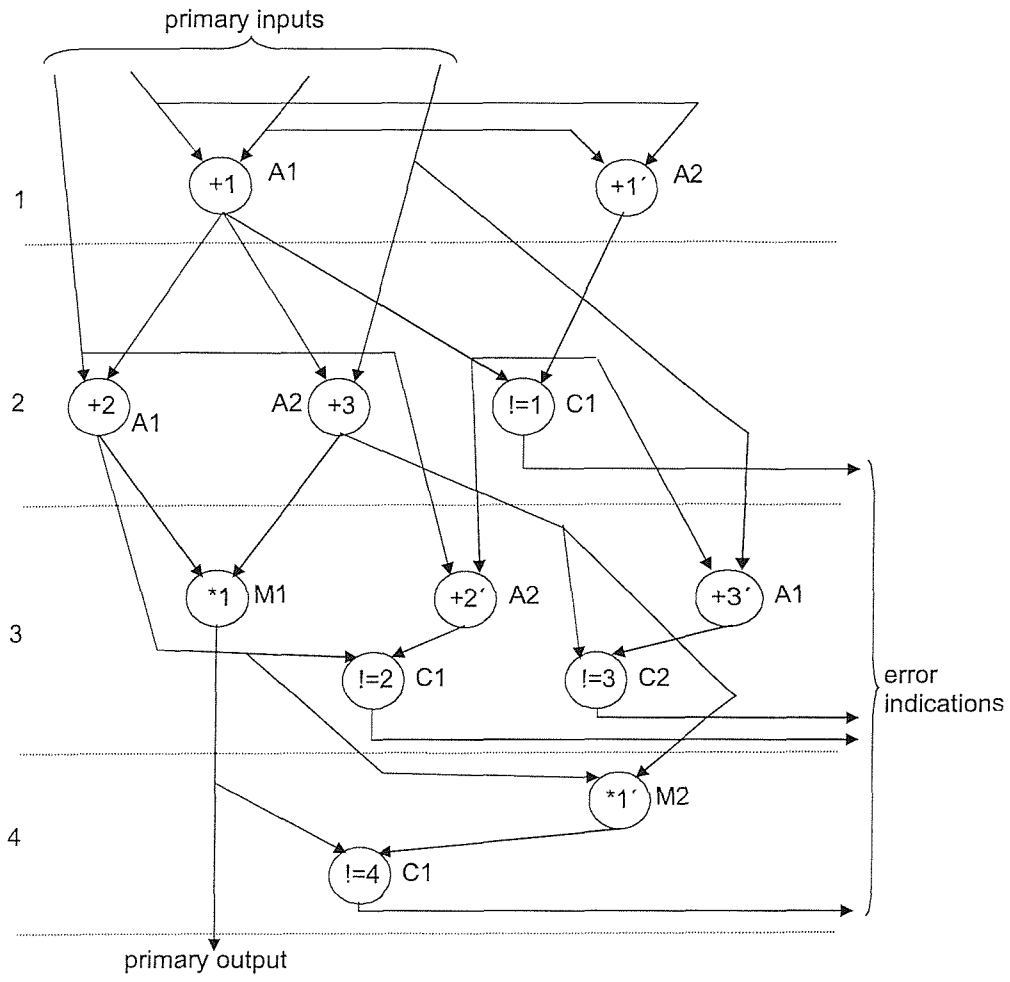


Figure 5.9. Checking all intermediate results for the example of Figure 5.2

will by nature rely on the target technology (e.g. targeting a dynamically reconfigurable FPGA part can reveal interesting opportunities for run-time self-repair). In order to keep this work generic and technology independent, this author makes no assumption regarding the fault recovery technique. This thesis is thus restricted to *timely* and *secure* reporting of faulty circumstances, such that faults can be reported *as soon as possible*, *before* the system primary outputs are corrupted, so that *any* recovery mechanism can react in a timely manner.

- Fault detection. The duplication-based fault detection mechanism applied in this thesis is effectively shaped by the requirement for timely reporting, as stated above. Previous research works overviewed in §5.2.1 mostly employed checking of primary outputs; at times not even all primary outputs were checked [65, 69, 70], while in certain cases selected but limited intermediate results were also checked [60, 23, 61]. The strict error latency requirements stated in this thesis mandate that *every single* intermediate result be

checked. Algorithmic duplication is applied to perform this checking; algorithmic inversion is also used alternatively.

- Synthesis. Addressing the whole self-checking problem at the high-level synthesis level has a number of challenges, implications, as well as inherent advantages over previous pieces of work. Insertion of self-checking resources should ultimately be done automatically by the synthesis tool, without any HDL modification or other intervention of the user to the synthesis process, other than specifying the synthesis constraints. Even further, self-checking insertion and other design optimisation (for area or delay) should be done in a *single* optimisation process, to facilitate efficient design space exploration (Figure 5.3). The choice between algorithmic duplication or inversion in a given situation should also be automatic within this same process. Moreover, both duplication and inversion require fault secure comparators and such comparators do not normally exist in cell libraries by default. The design of fault secure comparator cells, utilisable by the core synthesis system, is therefore an additional challenge. Once these goals have been reached, the resulting integral synthesis for on-line testability tool will be able to take full advantage of existing high-level synthesis benefits. To this end, loops and conditionals will be accommodated painlessly (so long as the original tool supports them), chaining of operations will be a feasible design choice, while independence of technology and support for alternative technologies through existing libraries will also be available by default.

Figure 5.9 shows how Figure 5.2 could be transformed to provide checking of all intermediate operations. The original data flow graph still comprises operations +1, +2, +3 and *1, dependent on each other and scheduled exactly as in Figure 5.2. The duplicate operations receive the same inputs as the respective original ones, and produce outputs that are compared against the original operation outputs through suitable comparison operations, implemented on introduced fault secure comparators. This can be confirmed on the figure, by focusing, for example, on additions +2 and +2′, whose outputs feed comparison !=2, implemented on comparator C1. The original operation output is always also fed to its proper successor operation (e.g. the output of +2 feeds *1). *All* internal arcs are thus verified concurrently with the useful operation. This ensures that all intermediate results are fault-free when they feed their successors, unless an error is indicated (at the right-hand side of the figure). This scheme clearly provides a *constant* monitoring of the health of the system, and detects faults literally as soon as they manifest themselves at the outputs of faulty functional units. For instance, if adder A2 is faulty and corrupts the output of opera-

tion +1', then the fault will be detected at CS 2 rather than at the end of the whole operation (CS 4). This may seem like a modest improvement for such a trivial example; one can however understand the importance of timely reporting in a realistic design with a critical path length of a few tens of cycles. On another note, the figure also depicts the chaining of operations mentioned earlier. Indeed, comparisons !=2, !=3 and !=4 are chained after redundant operations within control steps 3 and 4. Clearly, this is a design option; dedicated control steps could alternatively have been introduced for the comparisons. In a realistic situation, the choice will be made within the optimisation process, taking designer priorities and technology parameters into account.

Now focus on the error indications on the right-hand side of the DFG of Figure 5.9. In this particular example, two 2-bit output comparators C1 and C2 are used. Under the timely reporting assumption, the outputs of these comparators need to be combined and taken to a chip primary output port. This is done here by applying the standard practice of self-checking response compaction, using a two-pair dual-rail checker (§2.2.2.2). Figure 5.10 reminds us of the idea and illustrates its application in the particular context. Modules C1 and C2 in Figure 5.10 represent the comparators found in Figure 5.9. Two flip-flops (effectively constituting a 2-bit register) are attached to each of them. "Write enable" signals

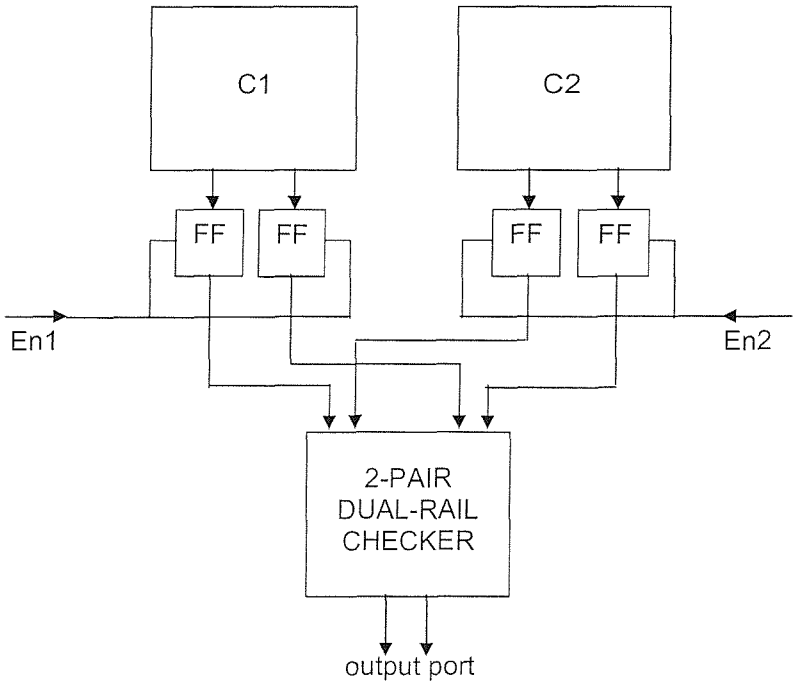


Figure 5.10. Compaction of datapath comparator responses

En1 and En2 are such that the flip-flops register their input values only at the appropriate control steps, according to the DFG. For instance, C2 produces a value of interest only at control step 3 and it is only during that control step that the corresponding flip-flop is enabled to store a value. Naturally, there is also a clock input to the flip-flops, omitted in the figure for clarity. The values stored in the flip-flops asynchronously feed a dual-rail checker, here acting as a response compactor. The output of the response compactor drives the overall chip "health indication" primary output. The system designer can then handle this health indication to trigger any recovery mechanism as desired.

A final observation on Figure 5.9 is that the data-flow graph is *overloaded* with nodes (operations) and especially arcs (operation input / outputs). Indeed, the introduction of several new comparisons and the associated data dependencies create a situation which may require a great many multiplexers, comparators and interconnect to be implemented in hardware. One may think that this overloading will lead to an unacceptably high hardware overhead, possibly higher than physical duplication, characterizing the whole approach impractical. Section 5.3 will experimentally prove that this is not the case if the optimisation potential of high-level synthesis is properly exploited.

5.3 Implementation and Experimental Results

The presentation of this chapter now moves on to the implementation of the concepts outlined in §5.2.3. Implementation involves two interdependent tasks. Firstly, insertion of self-checking resources should be done automatically and transparently, at the designer's request. Secondly, the resulting self-checking system should be optimised for the traditional high-level synthesis objectives, i.e. area and delay. These tasks should ideally be addressed simultaneously. Ultimately this can best be achieved if the self-checking problem is formulated in a manner that a high-level synthesis tool can use. The rest of this chapter details how this is achieved using the MOODS high-level synthesis system (§3.2), and presents experimental results, comparisons and conclusions. It should be noted that the work is by no means restricted to the particular tool. The concepts presented hereafter are generic in their essence; if a different tool was given, then the low-level practical implementation details could be adjusted as applicable to match the idiosyncrasies of the tool at hand. As shown in §3.2 MOODS is a transformational tool; a tool based on constructive

algorithms could alternatively be written. All the theoretical foundation of §5.1 and §5.2 would still be valid, but a different implementation strategy would need to be adopted.

5.3.1 Preliminary experiments

As a preliminary step towards implementing on-line testing within MOODS, a number of manual experiments targeting standard synthesis benchmarks were conducted using the original MOODS system of §3.2. These experiments essentially constituted a feasibility investigation. The manual methodology followed, results obtained and lessons learned are given in the following.

The first benchmark design used was the Tseng datapath, already presented in §5.2.2. With respect to Figure 5.8, the multiplication and the division have been substituted by left and right shifts respectively; this is permitted since their constant operands are powers of 2 (Appendix B), and in fact it leads to particularly economical realizations, since shifters are much cheaper than multipliers. For the purposes of this subsection, self-checking functionality was manually inserted to the design by modifying the original VHDL description. Consider the following simple addition example in VHDL :

```
v8i := v3i + v5i;                    (5.1)
```

where `v8i`, `v3i` and `v5i` are bit vectors representing unsigned integer values. Duplication testing is implemented as :

```
v8i := v3i + v5i;
sc1 := v3i + v5i;                    (5.2)
failed <= sc1 /= v8i;
```

`sc1` is an additional bit vector of the same size as the already existing ones, while `failed` is a single-bit port, responsible for communicating the error indication information. Inversion testing can alternatively be configured for the same example as follows :

```
v8i := v3i + v5i;
sc1 := v8i - v5i;                    (5.3)
failed <= sc1 /= v3i;
```

Both the original and the modified behavioural descriptions of the design were fed to MOODS and optimised using the existing tailored heuristics (§3.2.5.2), with equal priorities for the area and delay criteria and a nominal value for the clock period. The MOODS

RTL output was subsequently further synthesized targeting a standard FPGA part (Xilinx Virtex XCV800) and using a commercial tool (Synplicity Synplify version Pro 6.2 [124]). The final implementation was carried out using the Xilinx Design Manager (version 3.1i [125]). Table 5.2 sums the results of this experimentation. The first column on the table defines the synthesized version of the design. The *original* version refers to the untestable implementation (i.e. without any VHDL modification). The *duplicated* version is the result of applying the duplication modification exemplified in code segment (5.2) in all eight operations of the data-flow graph. The particular version also needed some further manual

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 5.2 : Tseng benchmark preliminary 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 5.3 : 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 5.4 : Diffeq benchmark preliminary 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 5.5 : QRS benchmark preliminary synthesis results

intervention, on top of the tailored heuristics optimisation. The reason is that the automatic optimisation procedure naturally assigns additions `v8i := v3i + v5i;` and `sc1 := v3i + v5i;` to the same functional unit, trying to minimise the overall hardware. Of course, in the case at hand this is not valid, because a functional and a redundant operation need to be executed by disjoint hardware for the self-checking scheme to be meaningful (§2.2.2). A number of manually selected applications of the "unshare" transformation TF13 (§3.2.3) were thus needed to produce valid self-checking output (see also [126] and Appendix A for information about running MOODS in a manual, "console" mode). Referring back to Table 5.2, the *inverted_1* version is the product of modifying operations according to the inversion paradigm of segment (5.3) *where applicable* (§5.2.2), while still retaining duplication where inversion is not applicable. In addition, an *inverted_2* version is given. The difference between this last version and *inverted_1*, is that in *inverted_2* the pairs of functional and redundant operations are not allowed to be chained in the same control step. Chaining is prevented manually in the DFG, by forcibly inserting a control step boundary between the two operations. The MOODS VHDL Reference and Style Guides [127, 128] or Appendix A of this thesis can be consulted for practical details on how this is done.

The rest of the columns in Table 5.2 give the actual numeric results of the synthesis experiments. FPGA resource usage is given in terms of the number of occupied slices [125]. The number of tristate buffers used is also included, for the sake of completeness. These buffers are used for multiplexing. More specifically, this author's design experience, backed by previous research conducted in [100], suggests that multiplexers implemented in FPGAs using standard look-up table based logic are very costly in terms of area (in fact they occupy more area than functional modules, thus rendering hardware sharing a disadvantageous option). It was found that using tristate buffers to implement multiplexers solves this problem, as there is a plethora of normally unused such buffers in a typical FPGA device. The number of buffers used may appear excessive, but this has no negative implications on the design quality, since it is the number of occupied slices that signifies the FPGA area utilisation. Speed parameters of the synthesized designs are reported subsequently; these are the critical path length (measured in number of clock cycles) and the maximum frequency achievable by a given realisation. The hardware and speed overheads are more clearly illustrated next, by means of the percentage of increase in slice usage and the percentage of performance degradation in number of cycles. Finally, for this small ex-

ample (eight operations) it is easy to calculate the average error latency, in cycles; this is reported in the last column of Table 5.2. The error latency is given as infinite in the case of the *original* design, since there is no on-line testing applied to it (i.e. faults are never detected). Note that the hardware usage and frequency statistics on the table are the ones reported by the actual lowest-level implementer tool; therefore, they are as realistic as could be and fully reflect the optimisation contribution of RT-level synthesis. This note applies not only to Table 5.2 but to all tables hereafter.

A simple comparison of the results in Table 5.2 for the *duplicated* and the *inverted_1* versions reveals that *inverted_1* has a smaller hardware overhead. This is consistent with the inversion testing intuition provided through Figure 5.8. Further, Table 5.2 shows that error latency in both cases is 0, since for all instructions in the DFG the functional, redundant and comparison operations are scheduled in the same control step, and thus faults are detected at the same control step as they occur. Performance degradation (in terms of clock cycles) is also the same; however, chaining of functional / inverse operation pairs within the same control step results in the maximum achievable clock frequency being 7 times lower in the *inverted_1* version. Focusing now on the *inverted_2* version, it can be seen that it needs an additional 4 cycles, but the maximum achievable clock frequency is not degraded with respect to the *duplicated* version. The hardware overhead is more than for the *inverted_1* version but is still less than the *duplicated* version. 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. In an attempt to more clearly demonstrate the area savings for this simple but illustrative example, Table 5.3 summarizes the functional module usage of the different Tseng versions. The *duplicated* version naturally features double the number of hardware components with respect to the original one; the *inverted_1* version is shown to include an adder and a subtractor less. In fact it is the absence of these two arithmetic modules that gives rise to a cheaper self-checking solution when using invertion testing. *Inverted_2* has exactly the same functional module usage as *inverted_1* and is thus not included on Table 5.3. The three extra FPGA slices that the unchained *inverted_2* version occupies are due to registers introduced to store the results of original computations across clock cycle boundaries, before being fed to the redundant ones.

The next design tried was a differential equation solver (hereafter Diffeq). It is taken from

[129] and it has also seen extensive usage for benchmarking purposes, also considered representative of more complicated but typical HLS situations. The experiments conducted with Diffeq are shown on Table 5.4. The version names have the same meaning as in the previous example. The self-checking versions were again produced manually and synthesized using equal priorities and targeting a Xilinx Virtex XCV800 FPGA part. The observations are along the same lines as before. The *duplicated* version is the most expensive, but also the fastest both as regards clock cycles required and maximum achievable clock frequency. Chained *inverted_1* is the cheapest with respect to hardware overhead, but suffers severe frequency degradation. Unchained *inverted_2* is moderate in hardware usage and does not cause frequency degradation, but results in a few additional clock cycles in the critical path.

The question that naturally arises in both of the above examples, is which of the on-line testable versions one would choose. As is usually the case when working in high-level synthesis, there can be no definite answer, and the choice is always up to the designer. Considering the results of Table 5.2 as an illustrative example, it can be commented that if cost is the biggest restriction, then the designer may probably choose the (cheapest) chained *inverted_1* version. If the clock frequency degradation imposed cannot be tolerated, maybe they will consider paying the extra price for the non-chained *inverted_2* realization. Still, if the additional clock cycles are unacceptable, maybe they will have to pay even more to have the *duplicated* version. Finally, if the latter is too expensive and reliability is not a first priority, the designer may decide to drop on-line testing completely and go for the *original* untestable version. It is thus in practice demonstrated that the trade-offs and dilemmas of traditional high-level synthesis apply equally to the problem at hand; this time, though, on-line testability acts as an *additional* parameter.

The last experiments of this subsection were conducted on the QRS benchmark [130]. The particular design is actually of substantial size (~70 operations, mainly additions, subtractions, and divisions by powers of 2, implemented by "shift right" modules), and it corresponds to a useful medical electronics application. Table 5.5 presents the obtained results. This table assumes a slightly different form from the previous Tables 5.2 and 5.4. Firstly, a dedicated column shows the particular FPGA targeted in each experiment. The maximum frequency is not reported; this is because there were no significant degradations in its value, since the original untestable QRS designs already feature considerable chaining.

Finally, the average error latency is not reported either, since the number of operations in this design make its manual calculation impractical. For this benchmark, initially a duplicated and an inverted version were configured and synthesized. These are denoted on the table as *dupl_1* and *inv_1*. Two different FPGAs were targeted in two different sets of experiments. The interesting observation is that there are cases when inversion can be more expensive than duplication; indeed, *dupl_1* is cheaper when a Virtex XCV800 is used, while *inv_1* is cheaper for the alternative part XC95288XV. This can be explained as an effect of low-level refinement, or of the place and route algorithms utilized by the final implementation tool. Clearly a design which appears more expensive than another when considered at a high level in the design flow, may at times demonstrate enhanced optimization potential at lower levels, especially in FPGA technology. This observation gives rise to a strong argument for high-level synthesis : it is desirable that the time from the conceptual design to the final solution be as little as possible, so that alternative solutions can be tried fast and efficiently.

A second observation on the table is that *dupl_1* and *inv_1* always experience severe delay degradation (more than 100%). This is a most undesirable effect and it can be explained as follows. Recall that self-checking functionality was added to the design by means of the VHDL modifications of (5.2) or (5.3). In both cases the one-bit signal `failed` was used to store fault indication information. Clearly only one "write" operation can target a signal at a given control step. This means that each of the comparison operations attempting to write to the `failed` signal will need a control step of its own. There are around 70 such comparisons (equal to the number of functional operations), so at least 70 discrete control steps will be needed for the self-checking design. This is indeed confirmed on the table (77 control steps). In effect, the implementation of self-checking as done here hinders the control step merging potential of the data-flow graph. This problem can partly be solved manually, by using multiple `failed` signals. For example, consider the following VHDL code segment describing a duplication-tested subtraction :

```
ecg_dif := ecg1 - ecgm1;
sc1 := ecg1 - ecgm1;                    (5.4)
failed1 <= sc1 /= ecg_dif;
```

A second self-checking operation immediately following should take the form :

```
ecg_dif256 := ecg_dif / 256;
sc2 := ecg_dif / 256;                    (5.5)
failed2 <= sc2 /= ecg_dif256;
```

If the two comparisons in the above code segments were assigned to discrete comparators, and the clock period requirements were not violated, then all operations of (5.4) and (5.5) could be scheduled in the same control step. In order to provide a concise error indication, *n* failed signals are combined through a logic OR :

```
failed <= failed1 or failed2 or ... or failedn;    (5.6)
```

Referring back to Table 5.5, the *dupl_2* and *inv_2* versions were configured for the QRS benchmark, each one implementing the respective self-checking strategies as before, but this time a total of n=7 different `failed` signals were used; the choice of number was random. The table shows that the performance degradation experienced by both designs was much more tolerable, while the inverted version was the cheapest for the particular technology, but marginally slower than the duplicated one. The obvious question in this procedure is if the random value assigned to *n* was the optimal choice, and if there is a way to determine which choice would have been optimal. In effect, different values of *n* would enable exploration of different parts of the overall design space. It would be particularly time consuming to try a good number of alternative choices in this example, since each choice would require modifications throughout the whole length of a substantially sized behavioural VHDL input. The need to *automate* the design space exploration process for self-checking resource insertion is evident.

Concluding this subsection, it is to be noted that the preliminary results presented above do not as such reach the goals of the present chapter, as outlined in §5.2.3. Two of their obvious weaknesses are the need for manual intervention and the use of conventional one-bit output comparators, not adhering to the scheme of Figure 5.10. They do, however, provide some useful insight on the problem of on-line testing within high-level synthesis, as summarized in the following two points :

- It is confirmed that the high-level synthesis considerations and trade-offs are relevant to self-checking resource insertion. Further automation in the design flow is also shown to be required, to facilitate efficient design space exploration for self-checking datapaths.

- Inversion testing appears to be a source of hardware savings, but is likely to lead to slower realisations, either by degrading the maximum clock speed, or by giving rise to additional clock cycles.

The first point provided the encouragement for further automation of the whole process; the second will be constructively used in §5.3.3.2.

5.3.2 Semi-automatic experiments

As §3.2 established, the internal functionality of MOODS involves the application of certain transformations to the design under synthesis, through multiple repetitions of the optimisation loop of Figure 3.7, directed by an automatic optimisation algorithm or by the designer manually interacting with the system, and controlled by a cost function. At the lowest level, it is the transformations that introduce changes to the resulting datapath structure. It is therefore sensible to state that the introduction of new functionality within MOODS has to begin with defining an appropriate set of additional transformations.

5.3.2.1 Self-checking resource insertion software framework

In order for redundancy-based on-line testing schemes to be incorporated within the MOODS environment, three additional transformations were initially implemented. Table 5.6 summarizes them. All three are described as "testing" transformations, thus distinguished from the allocation or scheduling transformations encountered in §3.2.3. A notable innovation in testing transformations is that they introduce new functionality to the design, while the traditional allocation and scheduling ones strictly preserve the circuit behaviour and only change the structural realisation. In that, the present work breaks with high-level synthesis tradition. However, it should be made clear that the original functionality of the design is not affected by the testing transforma-

symbolic name	description	type of transform
TF22	physically duplicate instruction	testing
TF23	physically invert instruction	testing
TF26	remove instruction testing scheme	testing

Table 5.6. Test resource insertion transformations

tions; only redundant instructions are inserted and strictly utilised for self-checking purposes. In that sense, testing transformations can be considered "semantic-preserving".

In order to be exploitable within the optimisation loop of Figure 3.7, each of the transformations of Table 5.6 needs associated "validate", "estimate" and "perform" software functions implemented within the MOODS system. The software development involved was carried out for the purposes of this work, taking up around 2000 lines of C++ code [119]. Detailed descriptions of the transformations of Table 5.6 are now given in the following.

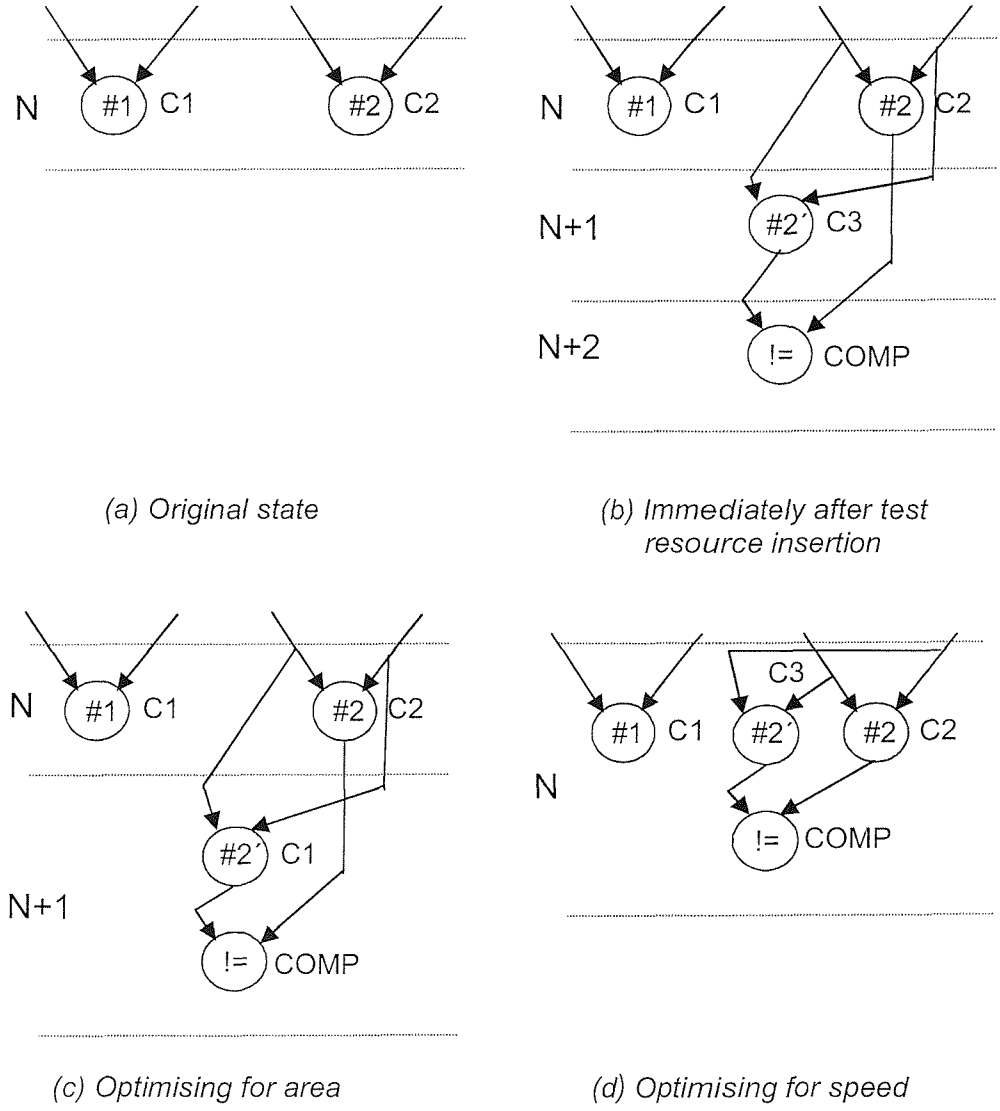


Figure 5.11. Insertion of duplication testing resources and subsequent optimization

Transformation TF22 (*physically duplicate instruction*) targets a given instruction and associates duplication-based self-checking resources to it. Clearly this involves the introduction of two additional operations, a duplicate and a comparison. The net result immediately after the transformation has been performed, is a locally "maximally serial" type of self-checking configuration, wherein a *new* datapath module has been introduced to implement the duplicate operation, together with a *new* comparator; the associated duplicate and comparison instruction also have *newly inserted* control steps dedicated to them. In other words, the transformation as such implements purely physical duplication and does not make any attempt to identify and reuse possibly existing idle modules. This initially appears to be naively expensive; Figure 5.11 depicts the situation and clarifies the benefits of such an approach. Firstly focus on Figure 5.11a. A very simple segment of a DFG features two independent operations of the same abstract type "#", namely #1 and #2. They are scheduled in a single control step N, and assigned to components C1 and C2. The components are assumed to be behaviourally identical. Figure 5.11b depicts the situation immediately after the application of TF22 on #2. The new elements mentioned above can be observed. Indeed, N+1 and N+2 are additional CSs, while a new component C3 implements the duplicate operation #2′ and an introduced comparator COMP implements the comparison !=. At this point remember that optimisation within MOODS consists of a substantial number of repetitions of the optimisation loop of Figure 3.7, effectively leading to the application of a substantial number of transformations. Therefore, the final state of the design does not need to be that of Figure 5.11b since more transformations will follow; Figures 5.11c and 5.11d show two possible mutually exclusive paths that subsequent optimisation steps can lead the design to. The scenario of Figure 5.11c implies that the designer has specified the chip area as a top priority constraint, while delay optimisation is secondary (§3.2.4). An area-oriented algorithm will then be chosen (for instance the heuristic of Figure 3.10b, readily available within MOODS). The hardware sharing transformation TF10 (§3.2.3) will then be applied on operations #1 and #2′. The result is that component C3 is dropped and C1 implements both #1 and #2′. Further, assume that the comparison can be chained after #2′ without affecting the clock period; the CS merging transformation TF8 will then move operation != to CS N+1 and drop CS N+2. Thus the state of the design reaches Figure 5.11c. One can observe that the solution at hand is a relatively cheap self-checking implementation (only the comparator is introduced), but gives rise to a delay degradation of a clock cycle (N+1). Alternatively, if the designer has

specified delay as his or her first priority, then the state of Figure 5.11d will be reached, wherein CSs N and N+1 are merged using TF8, invoked by the heuristic of Figure 3.10a. Subsequently, N+2 is also merged with the other two exactly as before, assuming again that the clock period is long enough. In Figure 5.11d it can be seen that any hardware sharing between C1, C2 and C3 is now impossible, since they are all active simultaneously. Therefore, self-checking is implemented at a high price, but the result is fast, since there is no additional delay degradation. The example overall shows that *naively* applying straightforward physical duplication and then allowing the existing synthesis framework to further optimise leads to a *versatile* design space exploration process, in the sense that the subsequent optimisation automatically follows the designer's directives and, depending on these directives, can take alternative paths. This would not be possible if TF22 immediately lead, for example, to the state of 5.11c, since then delay degradation would be unavoidable, and the requirements of a delay-constrained project less likely to be met. In effect, the initially naïve state of 5.11b is dictated by the nature itself of iterative high-level synthesis.

As all transformations, TF22 also needs a validity check phase. Given a target instruction, the validity check software function first checks if the instruction is a valid datapath operation. If it is, then duplication testing can readily by applied, unless a) a self-checking scheme has already been inserted and associated with the instruction, or b) the instruction itself is the duplicate or the inverse of another functional operation in the DFG.

Transformation TF23 (*physically invert instruction*) is very similar to TF22. It is performed exactly along the lines of Figure 5.11, although naturally in this case an inverse rather than a duplicate of operation #2 would appear in CS N+1 (Figure 5.11b). The trade-offs and design space exploration arguments built around Figures 5.11c and 5.11d equally apply in the inversion case. Once more, the same naïve start leads to a versatile process. The validity check phase is also very similar to that of TF22, with the important addition that, in order for TF23 to be valid, the targeted instruction should be mathematically uniquely invertible (§5.2.2). For this purpose, each instruction type that the tool supports is characterised as either invertible or non-invertible and this information is hard-coded into the tool; checking for invertibility is then a simple table look-up.

Note that both TF22 and TF23 introduce new comparison operations to the datapath. These comparisons need dedicated one-bit registers to preserve their results, and the outputs of the registers should be compacted to produce a concise output, exactly as the VHDL statement (5.6) showed. The test resource framework of this intermediate experiment in fact automatically introduced a statement such as (5.6) in the RTL output file, thus accommodating this need.

The third transformation shown on Table 5.6 is TF26 (*remove testing scheme*). It is the "undo" transformation of both TF22 and TF23. It targets a given instruction, and, as its name suggests, its function is to disassociate it from any self-checking resources that a previous application of either TF22 or TF23 may have inserted. This disassociation of a functional operation from its testing hardware may or may not involve a degree of actual dropping of hardware modules or control steps. As an example, refer back to Figure 5.11. If TF26 is applied to #2 at the state of Figure 5.11b, then operations #2′ and != will be abolished; hardware modules C3 and COMP are only allocated to the abolished instructions, and therefore they will be removed as well. Control steps N+1 and N+2 will also be empty and therefore not needed anymore. In contrast, if TF26 is applied at the design state of Figure 5.11c, then dropping out instruction #2′ should *not* be followed by the abolition of the component implementing it, since the component (C1) is also in use elsewhere (allocated to #1).

The validity check phase of TF26 needs to ensure that the targeted instruction : a) is a valid datapath operation, b) is not in itself the duplicate, inverse or comparison operation of a self-checking scheme, and c) has had self-checking resources associated to it and not yet removed.

As the final remark of this subsection, recall that in the experiments of §5.3.1, at times a certain manual intervention (unit unsharing) was needed, to ensure that MOODS did not assign the same hardware module to the functional and the duplicate operations of a given duplication scheme. As a supplement to the test resource insertion transformations presented here, the validity check function of the existing functional unit sharing transformation TF10 (§3.2.3) was augmented, such that the transformation is considered invalid in case its two target instructions happen to partake in the same self-checking scheme. This

slight modification allows the designer to safely use the tailored heuristic algorithms of §3.2.5.2, without hindering the validity of any previously inserted self-checking schemes.

5.3.2.2 Experimental results

At this point, a number of intermediate experiments were carried out. The objective of these experiments was to validate the software framework of §5.3.2.1, effectively by reproducing the results of Tables 5.2, 5.4 and 5.5; this time, however, no HDL modification was allowed, hoping that transformations TF22 and TF23 would do what the code segments did in §5.3.1. The experiments were conducted as follows. MOODS was invoked in console mode (Appendix A), a cost function chosen and the testing transformations were applied by interacting with the system and manually choosing the type of transformation (TF22 or TF23) and the target instruction. When all instructions in the design were made on-line testable, the existing tailored heuristic optimisation algorithm was applied. As explained in §3.2.5.2, heuristic optimisation automatically follows any of the three paths of Figure 3.10, depending on the designer priorities; in the context of this work, this equivalently means that optimisation of test resources automatically follows either of the paths of Figure 5.11 (or alternates between the two, in case of equal priorities). Implementing on-line testability this way is clearly a much more automated process than the one described in §5.3.1; however, a degree of manual intervention on behalf of the designer is still needed, even if this is through the tool user interface. This is why the approach of this subsection is termed "semi-automatic".

Tables 5.7 – 5.9 summarise the results provided by this set of experiments. All the elements on the tables are familiar from §5.3.1; the same three benchmarks and the same low-level tools were used, while version names also have the same meaning. *Inverted_2* versions this time were produced simply by specifying a very low clock period value, thus effectively disallowing chaining. Qualitatively the results of Tables 5.7, 5.8 and 5.9 match those of Tables 5.2, 5.4 and 5.5 respectively. Indeed, *inverted_1* is again the cheapest option for both the Tseng and the Diffeq designs, while the *duplicated* version is the least hardware-intensive in the QRS benchmark. *Inverted_2* always experiences the least frequency degradation. Some minor numerical mismatches between Tables 5.2 and 5.7, and 5.4 and 5.8, can be attributed to minor modifications to the MOODS system that are not related to this work. When comparing Tables 5.5 and 5.9, one notes that the designs on the

latter are significantly faster as well as more expensive. To understand the reason for this mismatch, recall the discussion of §5.3.1 regarding the dilemma over how many `failed` signals were to be used. Ultimately, a random number n=7 was chosen, and the *dupl_2* and *inv_2* designs of Table 5.5 were thus configured. In the designs of Table 5.9, literally every single self-checking scheme has its own error indication bit, because such bits are introduced together with the comparators, through the defined transformations TF22 and TF23. There is no mechanism to share the introduced error indication bits; therefore, even in the final, optimised design each self-checking scheme retains its unique error indication signal. These signals are equivalent to the `failed` signals defined in the manual experiments of §5.3.1. Since the QRS benchmark includes around 70 operations that all have self-checking schemes attached to them, the situation is equivalent to having around 70 different `failed` signals in the experiments of §5.3.1. In turn, this suggests that the portion of the design space explored by the semi-automatic approach is different from that

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	N/A	N/A	∞
duplicated	164	704	7	35 MHz	19.7	0	0
inverted_1	156	720	7	4 MHz	13.9	0	0
inverted_2	163	752	12	42 MHz	19.0	71.4	1.25

Table 5.7 : Tseng Benchmark semi-automatic experiments
(Target Technology Xilinx Virtex XCV800 FPGA)

Version	Resource Usage		Speed Parameters		Testing Penalty		Average Error Latency (cycles)
	Slices	Tristate Buffers	Cycles	Maximum Frequency	Hardware Overhead (slices %)	Performance Degradation (cycles %)	
original	234	642	13	31 MHz	N/A	N/A	∞
duplicated	344	1106	13	29 MHz	47.0	0	0
inverted_1	328	1106	13	5 MHz	40.2	0	0
inverted_2	404	1154	15	29 MHz	72.6	15.4	0.92

Table 5.8 : Diffeq benchmark semi-automatic experiments
(Target Technology Xilinx Virtex XCV800 FPGA)

Version	Resource Usage		Cycles	Testing Penalty		
				Hardware Overhead		Performance Degradation (cycles %)
	Slices	Tristate Buffers		slices %	tristate buffers %	
original	470	2626	34	N/A	N/A	N/A
duplicated	750	6548	36	59.6	149.4	5.9
inverted	762	6915	37	62.1	163.3	8.8

Table 5.9 : QRS benchmark semi-automatic experiments (Target Technology Xilinx Virtex XCV1000 FPGA)

explored during the manual experiments, and explains the quantitative differences. An automated way to determine the optimal comparison resources is still missing.

Once more, it has to be noted that the work presented in this experimentation round is still incomplete. Again the self-checking schemes lack the fault secure property, while full automation has not been achieved. However, the experiments are successful in that the transformational framework is experimentally validated; the subsequent §5.3.3 builds upon this framework and achieves full automation.

5.3.3 Fully automatic approach

As §3.2 has established, automatic optimisation within an iterative and transformational high-level synthesis tool such as MOODS primarily depends upon the set of available transformations, the form of the cost function constantly monitoring the quality of the system, and the choice of algorithms provided. High-level synthesis for on-line testability as outlined in this thesis has no reason to be different. Subsection 5.3.2.1 already defined three additions to the existing set of transformations. The following §5.3.3.1 will define and explain a metric for on-line testability, to be included in the system cost function. Subsection 5.3.3.2 will choose an algorithmic approach to fully automate test resource insertion and integrate it with subsequent optimisation. Subsection 5.3.3.3 will alleviate the lack of fault security of §5.3.1 and §5.3.2. Two more additional transformations will further be defined in §5.3.3.4. All these additions will create a fully integral and designer-friendly synthesis environment; experimentation results will be given in §5.3.3.5 and comparative comments on §5.3.3.6.

5.3.3.1 A metric for on-line testability

Transformations TF22 and TF23 (Table 5.6) have been shown to give rise to an initially inefficient design, paving the way for subsequent versatile optimisation. Still, the initial application of either of them results in a temporarily huge overhead (Figure 5.11b). Any synthesis system considering them will consult the controlling cost function to determine if they are beneficial or degrading; since for the tool at hand the cost function originally only relies on area and delay, one can conclude that the tool will be highly unlikely to accept TF22 or TF23 in automatic optimisation mode. This is because the area and delay estimation will *only* reflect the penalties but *not* the benefit of applying the transformation, causing it to appear brutally degrading. This not yet reflected benefit is, of course, the improvement in on-line testability. It follows that a metric is needed, to quantify on-line testability and include it in the original cost function (equation (3.3)), so as to bias the system towards introducing on-line testability by means of transformations TF22 and TF23.

The following heuristic on-line testability metric is proposed here :

$$T_{on-line} = \sigma_1 \times P_1 + \sigma_2 \times P_2 \times (1 - P_1) + \sigma_3 \times (\log(L^{-1}) + \sigma_4) \qquad (5.7)$$

where :

P_1% is the percentage of original operations made on-line testable

P_2% is the average (per functional module) idle time availability

L (measured in control steps) is the average error latency per self-checking scheme, where the term error latency refers to the number of clock cycles that elapse between the manifestation and the detection of a fault (equivalently, the number of control steps between the functional operation and the comparison of the self-checking scheme)

σ_1, σ_2, σ_3, σ_4 are weighting constants

$T_{on-line}$ is normalized over its maximum value, obtained for P_1=1 and L=0, and thus ultimately expressed in %. It is well known that $\lim_{L \to 0} L^{-1} = \infty$. As always, in practice infinity is expressed by a pre-defined "sufficiently large" number *INF*. In the context of this work, it was empirically chosen that the *INF* value should correspond to a quantity that cannot possibly appear in the synthesis session of a given design. Given that the largest quantity that can appear in a design is the number *OPS* of operations in the design, it was chosen that *INF*=*OPS*+1. The maximum value of on-line testability is then given by the expres-

sion $T_{on-line,\max} = \sigma_1 + \sigma_3 \times (\log(INF) + \sigma_4)$ and according to the above the normalized on-line testability $\langle T_{on-line} \rangle$ is ultimately given by

$$\langle T_{on-line} \rangle = \frac{T_{on-line}}{T_{on-line,\max}} \times 100\% \tag{5.8}$$

The ideas summarized by equation (5.7) are clarified in the following. Clarifying comments are provided with reference to the DFG of Figure 5.8.

- P_1 is clearly a factor that determines the quality of test, by simply reflecting that the more operations made on-line testable, the more testable the whole circuit is. For example, in the Tseng datapath as shown in Figure 5.8 only one out of eight original operations is on-line testable (addition +2, by means of inversion testing). Therefore P_1=1/8=12.5%.
- The percentage of available idle time is easily calculated for the given state of a design. In the example at hand, subtractor S1 is used in two out of a total of six control steps in the design. Therefore it is idle during 4 out of 6 CSs, yielding the 66.67% value for its idle time availability percentage. The respective percentages for the other modules in the datapath are 50% for adder A1, and 83.33% for multiplier M1, divider D1, comparator C1 and logic gates $G_{and}1$ and $G_{or}1$. Averaging these values yields P_2=76.19%.
- The term (1-P_1) by which idle time availability P_2 is multiplied, initially has the value 1 (because initially P_1=0), and as the design becomes more and more testable, it moves towards 0 (as $P_1 \to 1$). The significance of this, is that idle time can be an advantage in the first optimisation stages, because idle modules can be utilised in future optimisation steps to implement duplicate / inverse computations not yet inserted. As optimisation progresses, less and less idle time is needed, since fewer and fewer duplicate / inverse computations are to be inserted. Therefore, the term $\sigma_2 \times P_2 \times (1 - P_1)$ prevents functional module sharing in the initial stages, and allows it later on, when testing instructions will have been accommodated for, and there will be nothing to be gained by preventing sharing.
- As far as the third term of (5.7) is concerned, clearly faults need to be detected as soon as possible, thus the linearised inverse error latency is present to facilitate merging of control steps that intervene between the original computation and the comparison operation (for instance, CSs 4 and 5 in Figure 5.8). For the Tseng DFG at hand, only one self-checking scheme has been configured (+2, -2′, !=1); its error latency is 2 control steps. Therefore L=2.
- The weighting constant values in (5.7) determine the relative contribution of each term in the overall on-line testability value. They have been set such that the first term contrib-

utes 90% (as being the most important), while the third one contributes 10%. The second term contributes a small .1%. This does *not* practically add up to more than 100%, since the second term comes out of play as the first approaches its maximum value. The exact values used in the experiments of this work for the constants were $\sigma_1=9\times(\log(INF)+\sigma_4)$, $\sigma_2=0.01\times(\log(INF)+\sigma_4)$, $\sigma_3=1$, and $\sigma_4=0.3$. Notice that σ_1 and σ_2 depend on the "*INF*" value defined above; therefore, they are constants for a given synthesis project, since *INF* is a constant for a given design. These values were determined purely empirically, through experimentation and evaluation of the synthesis results produced using them. Notably, the overall contribution of the second term of equation (5.7) is very small. Clearly a higher value of σ_2 would have increased it, but once again experimentation dictated that this was not necessary.

The MOODS cost function now becomes

$$Cost = c_{area} \times area + c_{delay} \times delay + c_{OLT} \times \langle T_{on-line} \rangle \qquad (5.9)$$

Exactly like c_{area} and c_{delay}, c_{OLT} reflects the designer-specified priority of the on-line testability criterion.

Equations (5.7) and (5.8) succeed in providing a visualisation of the previously abstract concept of on-line testability, by identifying and exploiting the parameters that make up a good on-line testable design. Inclusion in the cost function (5.9) informs the synthesis suite of the importance of on-line testability and paves the way for automatic optimisation, through the choice of a suitable algorithm (§5.3.3.2). One subtle difference between on-line testability and the conventional criteria, is that optimising the latter re-

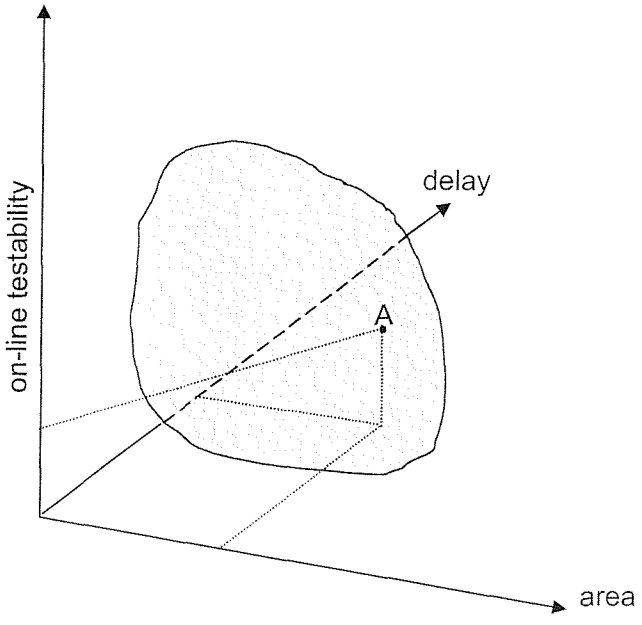


Figure 5.12. 3-dimensional design space (area, delay, on-line testability)

fers to minimisation (of e.g. area or delay), while optimising on-line testability is equivalent to *maximizing* its value. Indeed, if the designer wishes a design that would be "as testable as possible", then he or she should specify the 100% value as the testability optimisation target. For the same reason, c_{OLT} in equation (5.9) should be understood as holding a negative value. Other than that, equation (5.9) is fully consistent with the cost function description of §3.2.4. Further, the introduction of a third user specification effectively gives rise to a *three-dimensional design space*, as Figure 5.12 depicts. The coloured area shows the achievable region (§3.1.2) including the example point A, along with the projections of A on the three axes that define the space (area, delay, on-line testability).

5.3.3.2 Algorithms

The next step towards full inclusion of test resource insertion within the overall iterative optimisation process, is the choice and implementation of one or more suitable algorithm(s), to control the optimisation loop execution. Synthesis experience using MOODS suggests that the tailored heuristic algorithms of §3.2.5.2 are very fast, and normally provide acceptable results, despite the theoretical risk of ending up in a local minimum. The problem is that all versions of the heuristics use only a limited number of transformations; the testing transformations of §5.3.2.1 are not relevant, and there is no obvious way to include testing considerations to the metrics of §3.2.5.2. On the other hand, simulated annealing (§3.2.5.1) is very abstract and thus particularly suitable for optimising anything that can be quantified, regardless of its nature. The disadvantage of simulated annealing is its very slow speed.

In order to exploit the benefits and make up for the weaknesses of both simulated annealing and tailored heuristics, it was decided that a combination of the two should be used, as in the following :

- Step 1 : apply *modified* simulated annealing, using designer defined parameters for the initial and the terminating "temperature", as well as for the rate of temperature decrease per step
- Step 2 : apply the version of tailored heuristics that matches the area and delay design priorities (Figure 3.10)

The "modified" simulated annealing mentioned in Step 1 of the above procedure refers to the standard simulated annealing algorithm already implemented within the MOODS sys-

tem, with TF22 and TF23 included in the set of transformations, and a degree of determinism incorporated. This determinism consists in the following. When the algorithm randomly chooses a transformation from the set, if it happens to be a scheduling or allocation transformation then the algorithm proceeds as usual; if it turns out to be either of the test resource insertion transformations, then its actual type (duplication or inversion) is initially ignored, and which of the two will ultimately be applied is decided based on the following criteria :

- if the target instruction is not invertible, then duplication is applied, else
  - if no inverse module instance is already present in the design, then duplication is applied, else
    - if frequency requirements are relaxed, then inversion is applied, else
      - if delay is a higher priority than area, then duplication is applied, else
      - if area is more important than delay, then inversion is applied, else
      - area and delay are of equal importance; the initial randomly selected transformation (TF22 or TF23) is applied

The criteria upon which the choice of testing technique is made actually connect this discussion to the presentation of inversion testing in §5.2.2 and the manual experiments of §5.3.1. Indeed, remember that inversion testing is practically advocated in §5.2.2 only when idle modules of suitable types already exist in the datapath; if that is the case, then experimental observations in §5.3.1 suggest that applying inversion testing leads to compact designs, but severe degradation in the maximum achievable clock speed. It can therefore be beneficial in situations that do not demand very fast clocks, in other words when frequency requirements are relaxed. The exact numerical correspondence of "relaxed" frequency requirements is to be determined experimentally, and varies from design to design. On the other hand, when frequency requirements are strict and thus chaining is unlikely to occur, then duplication and inversion were found in §5.3.1 to lead to faster and cheaper (respectively) solutions; therefore duplication should be favoured when delay is the top priority, and vice versa. When area and delay have equal priorities, then there can be no certainty as to which choice will lead to a better long-term solution, and so the initial random choice is adopted.

The goal of the above modification is simply to prevent moves that design experience suggests are undoubtedly suboptimal. Although classic simulated annealing is famous for

turning around unfavourable situations and over time balancing at the cost function global minimum, there is no reason why a particular area of the design space cannot be excluded, if it is known a priori that the desired solution does not lie within that area. It is in the light of this statement that the above modifications were decided. The positive result is the acceleration of the simulated annealing algorithm.

Returning to Step 1, it is clear that the designer can specify the duration of the simulated annealing optimisation process through the temperature parameters. The implication is that simulated annealing is used primarily for test resource insertion and secondarily for area and delay optimisation; therefore the designer can experimentally determine parameters that practically apply simulated annealing for as much time as needed for a "sufficient" improvement in testability. Tailored heuristics are employed afterwards (Step 2), optimising the already testable design for the traditional criteria of delay and area. In this way, the abstract nature of simulated annealing is exploited, while its slow speed is compensated for, firstly by the introduction of a degree of determinism, and secondly by fast and efficient heuristics that take over as soon as simulated annealing has fulfilled its primary objective.

5.3.3.3 Fault secure comparators and dual-rail checkers

The concluding remarks of both §5.3.1 and §5.3.2 include mentions to the missing property of fault security. The present subsection presents the development work that solved this problem. Both duplication and, when applicable, inversion are fault secure as separately shown in §2.2.2 and §5.2.2, provided that the checkers / comparators used in the schemes are fault secure by design. This means that the datapath self-checking schemes of this chapter can all be made fault secure, if the conventional, single-bit output comparators §5.3.1 and §5.3.2 are replaced by the standard two-bit output fault secure comparator modules mentioned in §2.2.2.1. Therefore, the task of this subsection is the design of a library of fault secure comparators, and the necessary modifications to the MOODS system to utilise them in the self-checking schemes.

In essence, an n-pair fault secure comparator is composed of an n-pair fault-secure dual-rail checker (§2.2.2.2) and n inverters applied to one of the dual-rail input vectors. In turn, an n-pair dual-rail checker consists of n-1 dual-rail checker cells, such as the one shown in

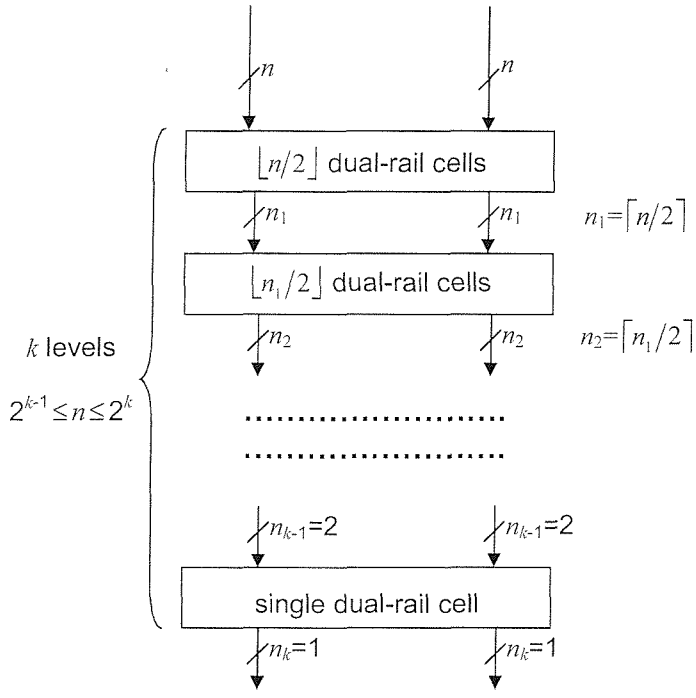


Figure 5.13. Block diagram of an n-bit dual-rail checker

Figure 2.30. Figure 2.31 has exemplified this concept, by showing a 5-pair dual-rail checker. A generic block diagram representation of an n-pair dual-rail checker is shown in Figure 5.13. The figure shows that the checker is effectively composed of k levels of arrays of dual-rail cells. The number of levels k, the number of cells in each array, and the number of intermediate signals between arrays are also analytically defined in the figure. An array fed by an even number of dual-rail pairs effectively applies dual-rail checks to each "pair of pairs" separately, since a dual-rail checker cell is in effect a 2-pair dual-rail checker. In the event that an array is fed by an odd number of pairs, one pair is simply carried to the array output and fed to the lower level array, unaffected. It can easily be verified that for n=5 Figure 5.13 produces Figure 2.31.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity CHK_ARR is
    generic (m: positive := 1);
    port (in1, in2 : in std_logic_vector (m-1 downto 0);
          output: out std_logic_vector ((m + (m rem 2))-1 downto 0));
end CHK_ARR;

architecture structure of CHK_ARR is
begin
    B1: for i in 1 to m/2 generate
      output(m+(m rem 2)-i) <= (in1(m-2*i+1) and in2(m-2*i)) or (in2(m-2*i+1)
and in1(m-2*i));
      output(m-i-m/2) <= (in1(m-2*i+1) and in1(m-2*i)) or (in2(m-2*i+1) and
in2(m-2*i));
end generate;
    B2: if ((m rem 2)=1) generate
      output((m+1)/2) <= in1(0);
      output(0) <= in2(0);
end generate;
end;
```

Figure 5.14 : The CHK_ARR *cell*

Clearly, the first step towards the design of a complete dual-rail checker is the design of dual-rail cell arrays. A generic and synthesisable VHDL description of a dual-rail array component has been written for this purpose; it is shown in Figure 5.14. The VHDL code shows that an appropriate, parameterized number of dual-rail cells are defined through signal assignment statements that follow the behaviour of Figure 2.30.

Using the array component of Figure 5.14, one can easily implement fault secure comparators and dual-rail checkers of any desired bit widths. Figure 5.15 shows the synthesizable VHDL description of a 16-pair dual-rail checker. A fault secure comparator is easily produced from the design of Figure 5.15, by simply substituting the signal assignment

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity NEQ_3_n16 is
    port (in1, in2 : in std_logic_vector(15 downto 0);
          output : out std_logic_vector(1 downto 0));
end NEQ_3_n16;

architecture structure of NEQ_3_n16 is
    signal intermediate_signals : std_logic_vector(61 downto 0);
    component CHK_ARR
              generic (m: positive := 1);
              port (in1 : in std_logic_vector (m-1 downto 0);
                    in2 : in std_logic_vector (m-1 downto 0);
             output : out std_logic_vector ((m + (m rem 2))-1 downto 0));
    end component;

    for all: CHK_ARR use entity work.CHK_ARR(structure);

begin
    intermediate_signals(61 downto 46) <= in1;
    intermediate_signals(45 downto 30) <= in2;
    U1: CHK_ARR generic map (16)
                port map (intermediate_signals(61 downto 46),
                          intermediate_signals(45 downto 30),
                          intermediate_signals(29 downto 14));
    U2: CHK_ARR generic map (8)
                port map (intermediate_signals(29 downto 22),
                          intermediate_signals(21 downto 14),
                          intermediate_signals(13 downto 6));
    U3: CHK_ARR generic map (4)
                port map (intermediate_signals(13 downto 10),
                          intermediate_signals(9 downto 6),
                          intermediate_signals(5 downto 2));
    U4: CHK_ARR generic map (2)
                port map (intermediate_signals(5 downto 4),
                          intermediate_signals(3 downto 2),
                          intermediate_signals(1 downto 0));
    output <= intermediate_signals(1 downto 0);
end;
```

Figure 5.15 : A 16-pair dual-rail checker

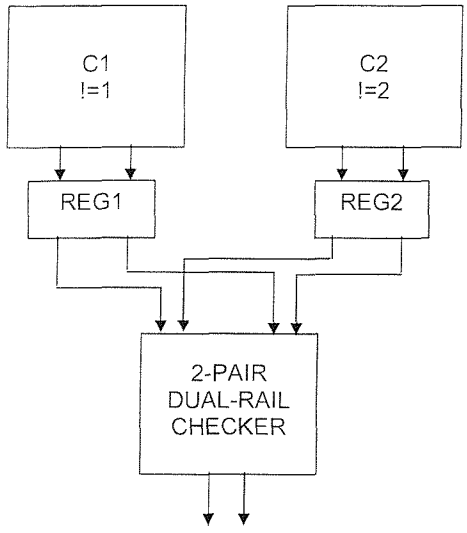
```
intermediate_signals(45 downto 30) <= in2;        (5.10)
```

with

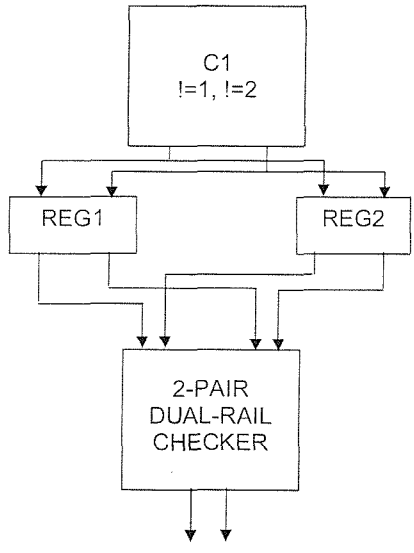
```
intermediate_signals(45 downto 30) <= not in2;    (5.11)
```

Following these analytical structure definitions, a C++ programme was developed, that automatically produced two libraries of VHDL descriptions of dual-rail checkers and fault secure comparators, for all bit widths between 1 and 200. The MOODS core synthesis system was then modified to use fault secure comparators in all self-checking schemes. Further, the interim technique of compacting comparator responses using OR gates as shown in §5.3.1 is no longer relevant. Instead, response compaction has to be done by using 2-bit registers attached to the comparators, and employing a universal dual-rail checker in the standard way of Figure 5.10. This was also accommodated for within MOODS, again by using a cell from the dual-rail checker library.

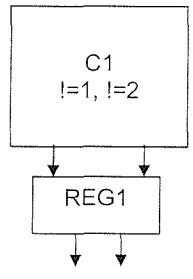
Note that the structure of Figure 5.13 is one out of several possible structures that an n-bit dual-rail checker can have. Such a checker will always use n-1 checker cells, but alternative structures can be configured by applying alternative internal arrangements of the cells within the checker. As explained in §2.2.2.2, different arrangements will need to receive different test sets during their normal operation to ensure the self-testing property. Therefore, if the inputs received during normal operation were known, it would be possible to choose the most efficient arrangement that would provide the self-testing property [58]. However, in the generic tool development context of this work, the inputs cannot possibly be known a priori. A solution that would ensure the self-testing property regardless of inputs would be the embedded dual-rail checker of Figure 2.32 [12, 19, 20]. This design, however, constitutes a very expensive solution, especially taking into account that a generic design can easily include tens of operations of realistic bit widths, that would need tens of long LFSRs if the structure of Figure 2.32 was applied to every single self-checking scheme configured for them. It was therefore decided that a theoretical concession be made, by not explicitly pursuing the TSC goal (notably, none of the previous works on algorithmic duplication pursue it either). The self-checking schemes are still fault secure, and if the chip operates for long enough for each scheme to receive all possible inputs, under Hypothesis 2.1, then they are self-testing too; if certain local conditions within a given design prevent a checker from receiving all possible inputs, then there is a



(a) original state



(b) undesirable post-sharing situation



(c) desired state

Figure 5.16. Sharing fault secure comparators

theoretical risk that faults may escape. Subsection 7.1 elaborates more on these considerations and proves that this risk is practically negligible for reasonably-sized designs.

In the light of the arguments stated in the above paragraph, any arbitrary arrangement of dual-rail cells within the checker would be sufficient for the purposes of this thesis. The structure of Figure 5.13 was consequently devised because it is well-defined, and thus it was possible to automate its design.

5.3.3.4 Auxiliary modifications

Subsection 5.3.3.3 has presented the details of the comparators needed throughout the self-checking designs that the modified MOODS will produce. Although they perform a special function in a specific context, these comparators are normal data path modules, taking up valuable area of the chip. It is therefore desired that they can be shared. In fact, the MOODS framework is readily able to share fault secure comparators, by virtue of the existing hardware sharing transformation TF10 (§3.2.3). However, the presence of the 2-bit registers together with the fact that MOODS has no reliable register sharing mechanism gives rise to suboptimal configurations as exemplified in Figure 5.16. Figure 5.16a is effectively a

simplified version of Figure 5.10, showing two fault secure comparators writing their results to respective registers and the register outputs compacted by a dual-rail checker. Suppose that comparator C1 implements comparison !=1, while comparator C2 implements !=2. If !=1 and !=2 have not been scheduled for the same control step, then the modules implementing them can be shared. Under this assumption, Figure 5.16b shows the situation immediately after merging C2 into C1 using the classic module sharing transformation TF10. It is easy to observe that unneeded logic remains in the system; indeed, there is no reason to keep both registers. In fact, sharing the registers not only saves a register, but also minimises the size of the response compactor. In the particular case, since the response compactor is only a 2-pair dual-rail checker, sharing the registers will enable its full removal; this is the desired state shown in Figure 5.16c.

The above example establishes the need for some limited register sharing functionality to be added to the synthesis system. As always within MOODS, this was formulated in a suitable transformation. To be consistent with the MOODS nomenclature, an "unsharing" transformation was developed too. These two transformations are tabulated in Table 5.10 and explained in detail in the following.

symbolic name	description	type of transform
TF24	share test response register	testing/ allocation
TF25	restore original test response register	testing/ allocation

Table 5.10. Additional transformations

Transformation TF24 (*share test response register*) targets two functional operations having testing schemes attached to them. It redirects the comparator output of the second scheme to the register that stores the comparator output of the first. The register originally attached to the second operation is abandoned. The test phase of TF24 first ensures that the target instructions are valid and suitable for self-checking. Then it checks that they actually both had self-checking schemes attached to them and neither of the schemes has been removed. Finally, it makes sure that the two comparators are under no condition active at the same control step.

Transformation TF25 (*restore original test response register*) is the inverse of TF24. Indeed, it targets a single functional instruction that has had a testing scheme attached to it and its dedicated test response register removed through TF24. It simply reintroduces the original register and redirects the testing scheme output accordingly. The test phase simply checks that the above statements about the targeted instruction are true, i.e. that it has had a self-checking scheme attached to it and the original register has been removed.

It is to be noted that TF24 and TF25 do not provide a proper framework for general-purpose register sharing. Indeed, register sharing generally refers to using a single register to store multiple *distinguishable* functional signals; instead, TF24 effectively implements what could be called "signal sharing". In simple terms, TF24 causes a certain non-functional, auxiliary signal to be fully abandoned (together with the register storing it) and an alternative one to take its place. Clearly this cannot apply to functional signals.

Although the "pseudo" register sharing implemented in this subsection is transformational, the relevant transformations TF24 and TF25 are not as such considered within the simulated annealing step of the automatic on-line test synthesis process (§5.3.3.2). Instead, they are embedded within the hardware sharing (TF10) and unsharing (TF12, TF13) transformations, such that whenever fault secure comparator modules are chosen to be shared or unshared, their respective target registers are shared or unshared as well. Thus, the transformations of this subsection can be seen as a way to "tidy up" the suboptimalities left by the pre-existing MOODS framework when interacting with the additions of this thesis (e.g. Figure 5.16).

One might think that sharing small 2-bit registers is a minor issue that will lead to only marginal improvements. However, recall §5.3.1 and the observation that, in the context of the manual experiments, using multiple `failed` signals produced very different results from using just one (Table 5.5). The need for an automatic way to identify an optimal number of such signals was also highlighted. The semi-automatic experiments (§5.3.2.2) further confirmed this need. In the dual-rail domain of this subsection, the 2-bit comparator outputs and the registers storing them are the equivalent to the single-bit `failed` signals of §5.3.1 and §5.3.2.2. In that sense, defining TF24 and TF25, and embedding them in the usual MOODS hardware sharing transformations provides an automatic solution to this last outstanding problem.

5.3.3.5 Experimental results

The final experimental results validating the automatic datapath self-checking design of this whole chapter are presented here. Given the synthesis framework of §5.3.3.1 – §5.3.3.4, no time-consuming HDL modification or console-mode operation-after-operation handling is needed anymore. When the modified MOODS is invoked, the designer has the chance to specify the cost function, both in terms of the traditional parameters (area, delay, clock period) and in terms of on-line testability. Subsequently synthesis proceeds along the lines of §5.3.3.2, beginning with an initial simulated annealing stage and concluding with a stage of tailored heuristic optimisation. If the designer does not specify an on-line testability specification, then the simulated annealing stage is omitted and an untestable version is produced, by plainly using the original synthesis suite of §3.2. In most cases, synthesis of self-checking designs finishes within *minutes*. This is an important advantage from the design space exploration point of view; indeed, it allows the designer to experiment with different values of parameters fast and painlessly, until a solution that satisfies his or her project needs is reached.

Tables 5.11 – 5.41 show the automatically obtained results. The three benchmarks mentioned in previous subsections (Tseng, Diffeq, QRS) are used; an additional few designs are also tried. Note that some of these benchmarks include loops, conditionals, as well as parallel processes (covered in more detail in §6.1.2). Thus, it is demonstrated that all structures likely to appear in a realistic design scenario can be accommodated. In all experiments, the designer's goals were set to 0 units of area, 0 nanoseconds of delay and, when desired, 100% on-line testability. Of course, these goals were classified as high or low priority, thus resulting in alternative design space exploration paths in each different synthesis run; this classification is always shown on the tables. In fact, on-line testability is always either a high priority or totally omitted. Further, the simulated annealing parameters were always chosen such that all the instructions in the design were secured by a self-checking scheme. Thus, in all experiments targeting on-line testability, P_1 of equation (5.7) ultimately assumes 100%. Together with other design statistics, the tables also report the on-line testability technique used, as well as a value for on-line testability as calculated using equation (5.7). Regarding the technique, the information on the tables only refers to the invertible instructions. Thus, "inversion" on the tables should be interpreted as "invert-

ible instructions are checked by inversion, while non-invertible ones still use duplication". On the other hand, "duplication" simply means "all instructions are checked by duplication". Very often some invertible instructions are checked by duplication and some others by inversion, in the same design. On the tables, this is termed a "mixed" technique, and it automatically arises when there is no deterministic reason to choose one over the other and a random choice is made within simulated annealing (as explained in §5.3.3.2). Regarding the testability value reported, since P_1 is always left to reach 100%, any deviation of $\langle T_{on-line} \rangle$ from its maximum 100% value is an indication of error latency. The desired clock frequency was adjusted between experiments, in order to promote or prevent chaining. Practically, for a given design in a given technology, a clock period value was experimentally identified that allowed unconstrained chaining; this is always shown on the tables as a "relaxed" clock period requirement. A second clock period value was also found, that did not allow any instruction chaining at all. This is termed a "strict" or "very strict" period constraint. In most cases, one or two period values between these two extremes were also tried and classified accordingly (e.g. "moderate").

The first automatic experiments were conducted using the Tseng benchmark, and targeting an FPGA part. Table 5.11 shows the results, highlighting points of particular interest. The least hardware-intensive self-checking version was the one on the second row, using inversion when possible and having a hardware overhead of 29.5%. There were two versions that did not experience any clock cycle degradation; one of them however suffered severe frequency degradation, due to relaxed clock period requirements leading to extensive chaining. The highest maximum frequency value (41 MHz) was achieved at a relatively high price (42.5% in hardware, 57.1% in clock cycles and some error latency, since testability is at 94.8%). The final choice lies with the designer; the goal of tool development, that is efficient design space exploration providing him or her with a variety of choices, is clearly achieved.

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	relaxed	-	146	432	7	48	N/A	N/A	none, 0.0
high	high	relaxed	high	189	752	7	7	29.5	0.0	inversion, 100.0
high	low	strict	high	208	784	11	41	42.5	57.1	inversion, 94.8
low	high	strict	high	197	736	7	38	35.0	0.0	duplication, 100.0
high	high	strict	high	197	752	8	40	35.0	14.3	mixed, 99.6

Table 5.11 : Tseng Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA)

Tables 5.12 – 5.14 sum up the results of the experimentation with the Diffeq benchmark. Three different untestable versions were synthesized, each one shown on a different table, with different clock period requirements. A total of nine self-checking versions were also produced. Notably, two different combinations of specifications can lead to effectively the same result. The second row Table 5.12 and the third row of Table 5.13 are an example of this phenomenon. This simply means that two different optimisation paths may lead to the

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	relaxed	-	234	642	13	31	N/A	N/A	none, 0.0
high	high	relaxed	high	321	962	14	7	37.2	7.7	inversion, 100.0
high	low	relaxed	high	321	962	14	6	37.2	7.7	inversion, 100.0
low	high	relaxed	high	323	962	14	8	38.0	7.7	inversion, 100.0

Table 5.12 : Diffeq Benchmark synthesis results (Target Technology Xilinx Virtex XCV800 FPGA), relaxed clock period requirements

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	moderate	-	234	642	13	31	N/A	N/A	none, 0.0
high	high	moderate	high	331	962	14	28	41.5	7.7	mixed, 100.0
high	low	moderate	high	321	962	14	7	37.2	7.7	inversion, 100.0
low	high	moderate	high	338	1026	14	28	44.4	7.7	duplication, 100.0

Table 5.13 : Diffeq Benchmark synthesis results (Target Technology Xilinx Virtex XCV800 FPGA), moderate clock period requirements

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	strict	-	306	706	19	43	N/A	N/A	none, 0.0
high	high	strict	high	427	1170	28	35	39.5	47.4	mixed, 91.6
high	low	strict	high	429	1202	30	37	40.2	57.9	inversion, 91.2
low	high	strict	high	436	1282	25	38	42.5	31.6	duplication, 92.1

Table 5.14 : Diffeq Benchmark synthesis results (Target Technology Xilinx Virtex XCV800 FPGA), strict clock period requirements

same point in the design space. The tables again highlight the optimum results with respect to different criteria. Hardware overhead can be as low as 37.2%, while clock cycle degradation is in several cases kept as low as a single cycle. The maximum frequency achieved by a self-checking design is 38 MHz, again at a certain hardware overhead and clock cycle penalty price.

The following tables 5.15 – 5.18 present the results of synthesis using the QRS design. This design is of particular significance, both because it corresponds to a useful system rather than a devised benchmark, and because of its substantial size. Each synthesis for on-line testability run with the particular design took approximately 20 minutes of real time, which is a serious time-to-market advantage. Indeed, having written the original VHDL description, the designer can use high-level synthesis to produce a variety of on-line

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	very strict	-	564	2552	66	19.2	N/A	N/A	none, 0.0
high	high	very strict	high	875	6703	69	2.1	55.1	4.5	mixed, 93.8
low	high	very strict	high	794	6511	66	8.5	40.8	0.0	duplication, 94.5
high	low	very strict	high	983	6298	107	2.3	74.3	62.1	mixed, 92.1

Table 5.15 : QRS Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), very strict clock period requirements

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	strict	-	514	2689	45	2.6	N/A	N/A	none, 0.0
high	high	strict	high	774	7221	47	1.1	50.6	4.4	mixed, 95.8
low	high	strict	high	788	7357	43	1.0	53.3	-4.4	duplication, 95.4
high	low	strict	high	829	5936	101	3.1	61.3	124.4	mixed, 93.0

Table 5.16 : QRS Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), strict clock period requirements

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	moderate	-	457	2577	34	9.7	N/A	N/A	none, 0.0
high	high	moderate	high	706	7221	37	1.0	54.5	8.8	mixed, 100.0
low	high	moderate	high	715	7336	33	0.8	56.5	-2.9	mixed, 97.3
high	low	moderate	high	839	5936	100	2.7	83.6	194.1	mixed, 92.9

Table 5.17 : QRS Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), moderate clock period requirements

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	relaxed	-	470	2626	34	3.2	N/A	N/A	none, 0.0
high	high	relaxed	high	764	7164	37	0.6	62.6	8.8	mixed, 97.3
low	high	relaxed	high	732	7227	34	0.9	55.7	0.0	mixed, 100.0
high	low	relaxed	high	839	5936	100	2.6	78.5	194.1	mixed, 92.9

Table 5.18 : QRS Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), relaxed clock period requirements

testable realisations to choose from, within hours of real time. The cheapest on-line testable realisation identified used up 706 FPGA slices, for an overhead of 54.5% with respect to its original untestable design. In general, all but one solutions in this experiment experience a hardware overhead of more than 50%, which is still cheap with respect to straightforward physical duplication and comparison (>100%). There is a huge variation as regards design performance. Indeed, one can notice values between 33 and 107 cycles. This is because of the substantial size of the design, which consequently means the achievable region in the design space is also of substantial size. In turns, this effects in a particularly large number of optimisation paths, once more stressing the importance of being able to traverse these paths quickly. An interesting observation is that on two occasions (Tables 5.16 and 5.17) there exist testable designs that are faster (i.e. take up slightly fewer clock cycles) than their untestable equivalents. This can perfectly well be attributed to algorithm inefficiencies (the tailored heuristic algorithms are, after all, only *heuristics*). A more

elaborate explanation is that the particular versions are shown on the tables to be produced using a "high" delay priority versus a "low" for area, while their corresponding untestable designs have an equal priority for the two criteria.

Table 5.19 shows the experimental results reached for a design not encountered earlier in this thesis; that is an 8-bit viterbi decoder, featuring 72 operations. It is not a standard HLS benchmark, and it is explained in [131]. The full VHDL code can be found in Appendix B. One observation on the table is that in this case it appears rather clear which testable version will most probably be preferred. Indeed, the last row shows a design that is both the cheapest and the fastest in clock cycles, although it experiences a modestly suboptimal degradation in maximum frequency. The most serious observation, however, is that all three synthesized testable designs are rather expensive; indeed, in two cases their hardware overhead greatly exceeds 100%. The explanation for this is that the particular design is composed of parallel VHDL processes, each one using a *single* copy of each of its hardware modules to implement the instructions assigned to it. The only way to perform duplication testing under these circumstances is to physically introduce an additional module of every type, in every process. This very much results in physical duplication; the

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	relaxed	-	174	344	4	37	N/A	N/A	none, 0.0
high	high	relaxed	high	428	936	6	31	146.0	50.0	duplication, 95.3
high	low	relaxed	-	174	344	4	38	N/A	N/A	none, 0.0
high	low	relaxed	high	448	849	7	37	157.5	75.0	duplication, 92.9
low	high	relaxed	-	174	344	4	37	N/A	N/A	none, 0.0
low	high	relaxed	high	314	731	4	33	80.5	0.0	duplication, 100.0

Table 5.19 : 8-bit viterbi decoder synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), relaxed clock period requirements

conclusion is that the particular design is rather unsuitable for duplication testing. Suitable error-correcting codes (§2.2.1) would probably give cheaper, although technology-specific and harder to devise, results. On the positive side, it is important that the synthesis tool was able to come up with *some* solution, even for this pathological design. This proves the *generic* property, highly desired when developing a tool. Besides, an overhead of 80.5% is still below 100% and it may be acceptable in certain applications.

The next design experimented with, was an elliptical filter, taken from [8]. The results obtained when targeting an FPGA part are shown in Tables 5.20 – 5.22, where the best numerical results per parameter are highlighted. It is an interesting benchmark, in that the

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	low	relaxed	-	322	923	32	52	N/A	N/A	none, 0.0
high	low	relaxed	high	541	2052	38	34	68.0	18.8	duplication, 93.8
high	high	relaxed	-	315	1018	17	50	N/A	N/A	none, 0.0
high	high	relaxed	high	408	2626	17	32	29.5	0	duplication, 100.0
low	high	relaxed	high	426	2674	19	32	35.2	11.8	duplication, 99.9

Table 5.20 : Ellip Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), relaxed clock period requirements

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	low	moderate	-	322	923	32	52	N/A	N/A	none, 0.0
high	low	moderate	high	497	2019	36	34	54.3	12.5	duplication, 94.3
high	high	moderate	-	315	1018	17	50	N/A	N/A	none, 0.0
high	high	moderate	high	437	2562	20	32	38.7	17.6	duplication, 99.9
low	high	moderate	high	446	2450	23	34	41.6	35.3	duplication, 97.4

Table 5.21 : Ellip Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), moderate clock period requirements

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	low	strict	-	322	923	32	52	N/A	N/A	none, 0.0
high	low	strict	high	516	2020	39	34	60.2	21.9	duplication, 93.6
high	high	strict	-	315	1018	17	50	N/A	N/A	none, 0.0
high	high	strict	high	467	2642	21	32	48.3	23.5	duplication, 98.3
low	high	strict	high	441	2579	21	33	40.0	23.5	duplication, 97.4

Table 5.22 : Ellip Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), strict clock period requirements

range of variation in the statistics is particularly broad. For example, the hardware overhead for test resource insertion ranges from a modest 29.5% to 68%. Some synthesis sessions have clearly failed. Indeed, on all three tables, the second row corresponds to a self-checking design synthesized with a high priority for area optimisation and a low priority for delay optimisation. However, the heuristics totally failed in these cases, since the results are both the most expensive and the slowest when compared to the other synthesis runs. Ultimately, design space exploration leads to a very good result, shown on the fourth row of Table 5.20, requiring a minimum hardware overhead of 29.5% with no additional clock cycles and a maximum frequency value of the same order as all other self-checking results.

The last benchmark design that tested the datapath self-checking synthesis system targeting FPGA technology was a Greater Common Divider module (GCD), found in [129]. Tables 5.23 – 5.25 summarise the results. On these tables one can observe the same phenomenon already seen on Tables 5.15 – 5.17, that is, on-line testable design that are

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	relaxed	-	81	292	7	26	N/A	N/A	none, 0.0
high	high	relaxed	high	137	748	7	29	69.1	0.0	duplication, 99.7
high	low	relaxed	high	121	580	6	25	49.4	-14.3	duplication, 100.0
low	high	relaxed	high	124	644	6	22	53.1	-14.3	duplication, 100.0

Table 5.23 : GCD Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), relaxed clock period reuiqrements

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	moderate	-	82	276	8	26	N/A	N/A	none, 0.0
high	high	moderate	high	140	668	8	33	70.7	0.0	duplication, 99.7
high	low	moderate	high	127	580	7	33	54.9	-12.5	duplication, 100.0
low	high	moderate	high	126	596	7	32	53.7	-12.5	duplication, 100.0

Table 5.24 : GCD Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), moderate clock period requirements

Synthesis constraints and priorities				Hardware usage		Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability	slices	Tristate buffers	clock cycles	maximum frequency (MHz)	hardware (slices %)	speed (cycles %)	
high	high	strict	-	84	228	9	42	N/A	N/A	none, 0.0
high	high	strict	high	144	652	9	37	71.4	0.0	duplication, 99.7
high	low	strict	high	144	716	8	32	71.4	-11.1	duplication, 100.0
low	high	strict	high	151	716	8	34	79.8	-11.1	duplication, 100.0

Table 5.25 : GCD Benchmark synthesis results (Target Technology Xilinx Virtex XCV1000 FPGA), strict clock period requirements

faster than their corresponding untestable versions. Again, this can be regarded as a sign of inefficient performance of the heuristic algorithm when synthesizing the untestable design. In fact, it is likely that the untestable versions ended up in a cost function local minimum. When on-line testability was applied, a degree of simulated annealing helped the synthesis process escape the local minimum, while at the same time the introduction of self-checking resources created an overall very different design for the heuristic algorithms to optimise. The results experimentally prove that the overall strategy was successful. Regarding the area overhead reported on Tables 5.23 – 5.25, this is in most cases relatively high, but it can be kept at as little as just below 50% (49.4% on the third row of Table 5.23).

Thus the experiments conducted to target Xilinx FPGA parts finished. One of the benefits of high-level synthesis mentioned throughout this thesis is the technology-independence of the core synthesis system, and the ability to optimise for alternative technologies if suitable technology libraries are provided (§3.1). In order to experimentally validate the point of technology independence, and evaluate the performance of the ideas of §5.2.3 in a different technology, development work was undertaken that produced a MOODS technology library targeting an Alcatel CMOS .35μm technology. This effectively allowed the

duplication of all the experiments of Tables 5.11 – 5.25 for this alternative technology. Very much like in the FPGA case, the RTL output of MOODS was fed to a low-level tool for register-transfer level synthesis. The tool in this case was version 2002e.16 of Mentor Graphics LeonardoSpectrum [132]. The results for different synthesis runs are shown in the following Tables 5.26 – 5.41. The only difference with respect to the previous tables in this chapter is that hardware usage is now naturally reported in terms of logic gates required. Once more, the results on the tables are the ones reported by the low-level tool, so they are as accurate as possible.

Table 5.26 summarizes the experimentation for the Tseng benchmark. One can easily observe that Table 5.26 accurately follows the lines of Table 5.11, in that the same choices of priorities are needed to produce e.g. the cheapest or the fastest result.

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	relaxed	-	1799	7	63.7	N/A	N/A	none, 0.0
high	high	relaxed	high	2308	7	15.2	28.3	0.0	inversion, 100.0
high	low	strict	high	2830	12	52.8	57.3	71.4	inversion, 94.0
low	high	strict	high	2367	7	50.2	31.6	0.0	duplication, 100.0
high	high	strict	high	2644	9	51.1	47.0	28.6	mixed, 97.2

Table 5.26 : Tseng Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI)

Tables 5.27 – 5.29 show the alternative solutions produced for the Diffeq benchmark. The lowest hardware penalty required for self-checking is identified to be 33.9%. There exist several versions that only impose a single clock cycle of delay degradation, while notably the fastest testable design produced does not need any additional cycles. Finally, the results on Table 5.29 can achieve very high frequencies at a high area price and additional cycles; if high frequency is an issue in a given project, then the second and fourth rows of Table 5.28 may be the best candidates, since they experience a modest frequency degradation with good area and delay statistics. Once more, there is satisfactory consistency with Tables 5.12 – 5.14.

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	relaxed	-	3535	13	40.3	N/A	N/A	none, 0.0
high	high	relaxed	high	4759	14	14.6	34.6	7.7	inversion, 100.0
high	low	relaxed	high	4739	14	13.8	34.1	7.7	inversion, 100.0
low	high	relaxed	high	4734	14	12.8	33.9	7.7	inversion, 100.0

Table 5.27 : Diffeq Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), relaxed clock period reuiqrements

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	moderate	-	3535	13	40.3	N/A	N/A	none, 0.0
high	high	moderate	high	4909	13	36.2	38.9	0.0	mixed, 100.0
high	low	moderate	high	4734	14	12.8	33.9	7.7	inversion, 100.0
low	high	moderate	high	4784	14	35.6	35.3	7.7	duplication, 100.0

Table 5.28 : Diffeq Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), moderate clock period requirements

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	strict	-	4111	19	40.3	N/A	N/A	none, 0.0
high	high	strict	high	6552	27	40.2	59.4	42.1	mixed, 91.6
high	low	strict	high	6990	30	40.5	70.0	57.9	inversion, 91.2
low	high	strict	high	6018	25	41.0	46.4	31.6	duplication, 92.1

Table 5.29 : Diffeq Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), strict clock period requirements

Tables 5.30 – 5.33 summarize the experiments conducted for the QRS benchmark in VLSI technology. An immediate observation is that the hardware penalty is relatively high, never dropping below 72%. This can be compared against Tables 5.15 – 5.18, where overheads around 55% were often achievable. Other than that, once more a substantially

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	very strict	-	7559	56	43.1	N/A	N/A	none, 0.0
high	high	very strict	high	13747	56	21.6	81.9	0.0	mixed, 94.0
low	high	very strict	high	13278	51	23.4	75.7	-8.9	duplication, 94.7
high	low	very strict	high	14813	101	32.0	96.0	80.4	mixed, 92.2

Table 5.30 : QRS Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), very strict clock period requirements

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	strict	-	7137	39	19.7	N/A	N/A	none, 0.0
high	high	strict	high	12759	40	3.2	78.8	2.6	mixed, 97.9
low	high	strict	high	12959	37	3.6	81.6	-5.1	duplication, 96.0
high	low	strict	high	12953	91	8.3	81.5	133.3	mixed, 93.1

Table 5.31 : QRS Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), strict clock period requirements

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	moderate	-	6849	35	9.2	N/A	N/A	none, 0.0
high	high	moderate	high	12574	34	2.9	83.6	-2.9	mixed, 98.7
low	high	moderate	high	12390	32	3.0	80.9	-8.6	mixed, 93.1
high	low	moderate	high	12953	91	8.3	89.1	160.0	mixed, 93.1

Table 5.32 : QRS Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), moderate clock period requirements

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	relaxed	-	6936	34	8.9	N/A	N/A	none, 0.0
high	high	relaxed	high	11927	32	3.0	72.0	-6.0	mixed, 100.0
low	high	relaxed	high	12063	31	2.7	73.9	-8.8	mixed, 98.7
high	low	relaxed	high	12953	91	8.3	86.8	167.6	mixed, 93.1

Table 5.33 : QRS Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), relaxed clock period requirements

sized design such as QRS once again has a particularly broad design space; this is verified on the tables by the variety of different results. Further, the phenomenon that certain on-line testable designs are faster than their untestable counterparts can once more be observed.

Table 5.34 briefs the experiments for the 8-bit viterbi decoder. The encouraging observation is that in VLSI technology the hardware overheads are generally much more tolerable than the FPGA ones of Table 5.19.

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	relaxed	-	2062	4	116.2	N/A	N/A	none, 0.0
high	high	relaxed	high	4589	5	85.8	122.6	25.0	duplication, 95.3
high	low	relaxed	-	3262	5	106.9	N/A	N/A	none, 0.0
high	low	relaxed	high	4734	7	127.4	45.1	40.0	duplication, 93.3
low	high	relaxed	-	2060	4	113.7	N/A	N/A	none, 0.0
low	high	relaxed	high	3421	5	92.5	66.1	25.0	duplication, 100.0

Table 5.34 : 8-bit viterbi decoder synthesis results (Target Technology Alcatel CMOS .35 VLSI), relaxed clock period requirements

The elliptical filter experiments are shown in the following Tables 5.35 – 5.37. The tables show a number of points in the 3D design space that can be considered neighbouring, in that most of them have a critical path length of 17 or 18 clock cycles, are composed of around 6500 – 6900 logic gates (minimum 6589 for an overhead of 48.2%), and can achieve frequencies in most cases around 35 – 40 MHz. The optimal values with respect to each of these criteria are highlighted separately on the tables, while designs for which a parameter is outside these ranges are rather unlikely to be favoured by the designer.

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	low	relaxed	-	4174	30	49.3	N/A	N/A	none, 0.0
high	low	relaxed	high	7678	37	40.9	83.9	23.3	duplication, 93.6
high	high	relaxed	-	4446	17	50.0	N/A	N/A	none, 0.0
high	high	relaxed	high	6706	17	41.6	50.8	0.0	duplication, 100.0
low	high	relaxed	high	6862	19	34.8	54.3	11.8	duplication, 99.9

Table 5.35 : Ellip Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), relaxed clock period requirements

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	low	moderate	-	4174	30	49.3	N/A	N/A	none, 0.0
high	low	moderate	high	8015	37	43.5	92.0	23.3	duplication, 93.6
high	high	moderate	-	4446	17	50.0	N/A	N/A	none, 0.0
high	high	moderate	high	6887	18	37.0	54.9	5.9	duplication, 99.9
low	high	moderate	high	6589	18	40.2	48.2	5.9	duplication, 100.0

Table 5.36 : Ellip Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), moderate clock period requirements

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	low	strict	-	4174	30	49.3	N/A	N/A	none, 0.0
high	low	strict	high	8015	37	43.5	92.0	23.3	duplication, 93.6
high	high	strict	-	4446	17	50.0	N/A	N/A	none, 0.0
high	high	strict	high	6897	21	41.2	55.1	23.5	duplication, 99.9
low	high	strict	high	6589	18	40.2	48.2	5.9	duplication, 100.0

Table 5.37 : Ellip Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), strict clock period requirements

Notably, the hardware overhead was not found possible to drop as low as the best choice of the equivalent FPGA-targeting experiment of Table 5.20.

The GCD benchmark synthesis experiments for VLSI technology are summarized in Tables 5.38 – 5.40. The observation in this experiment with respect to Tables 5.23 – 5.25 is

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	relaxed	-	1041	7	40.7	N/A	N/A	none, 0.0
high	high	relaxed	high	1198	8	40.1	15.1	14.3	duplication, 100.0
high	low	relaxed	high	1471	8	40.3	41.3	14.3	duplication, 100.0
low	high	relaxed	high	1489	6	36.6	43.0	-14.3	duplication, 100.0

Table 5.38 : GCD Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), relaxed clock period requirements

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	moderate	-	1041	7	40.7	N/A	N/A	none, 0.0
high	high	moderate	high	1315	8	43.9	26.3	14.3	duplication, 99.7
high	low	moderate	high	1564	8	37.5	50.2	14.3	duplication, 99.7
low	high	moderate	high	1418	6	36.2	36.2	-14.3	duplication, 100.0

Table 5.39 : GCD Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), moderate clock period requirements

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
high	high	strict	-	978	9	60.6	N/A	N/A	none, 0.0
high	high	strict	high	1378	8	43.9	40.9	-11.1	duplication, 100.0
high	low	strict	high	1450	9	43.9	48.3	0.0	duplication, 97.5
low	high	strict	high	1324	8	43.1	35.4	-11.1	duplication, 100.0

Table 5.40 : GCD Benchmark synthesis results (Target Technology Alcatel CMOS .35 VLSI), strict clock period requirements

that most overheads appear lower than in the FPGA scenario. Indeed, the hardware penalty can be as low as 15.1%, while performance degradation is either non existent or tolerable. Maximum frequency values in self-checking versions are usually of the same order as in the original equivalents, with the exception of Table 5.40 where a maximum frequency drop of roughly 16 MHz can be observed.

A simple statement that can be given regarding the proportional overheads of designs implemented in VLSI compared to the same designs implemented on FPGA parts, is that no safe assumption can be made about the relative overheads of the latter from the experiments targeting the former, and vice versa. That is, if on an FPGA part a design requires a certain hardware overhead to be made self-checking, the same design in VLSI may require much lower, much higher or roughly the same. This is expected, since the relative sizes of different RTL components greatly vary from technology to technology. It is, for example, well known that logic gates are expensive on an FPGA, while arithmetic modules are comparatively more expensive in VLSI. Therefore, experimentation is the only way for a secure conclusion, and this further stresses the importance of facilitating such experimentation through high-level synthesis.

As a final experiment, Table 5.41 shows the results of two synthesis runs for a 32-bit viterbi decoder in VLSI. This design comes from [131] together with its 8-bit counterpart presented earlier. It is however much bigger; it comprises 288 operations and 32 parallel processes, which are both considerably bigger than anything presented in the algorithmic duplication literature before. An original untestable design was synthesized first, followed

Synthesis constraints and priorities				Hardware usage (gates)	Performance		Overheads		Testability (technique, value %)
area	delay	clock period	on-line testability		clock cycles	maximum frequency (MHz)	hardware (gates %)	speed (cycles %)	
low	high	relaxed	-	15606	4	79.6	N/A	N/A	none, 0.0
low	high	relaxed	high	20361	5	49.9	30.5	25.0	duplication, 95.3

Table 5.41 : 32-bit viterbi decoder synthesis results (Target Technology Alcatel CMOS .35 VLSI)

by a self-checking version. The penalties related to test resource insertion can be regarded as moderate (30.5% in area, a clock cycle in delay, and some necessary degradation in maximum frequency). Notably, the automatic synthesis run for the self-checking version took about 24 hours of real time. The explanation is that the increase in the number of operations in the design results in a considerable increase in the number of different random choices of transformations and data within simulated annealing, thus lengthening the synthesis time. To understand this, refer, for example, to transformation TF10 of §3.2.3

("share functional unit"), and consider a design comprising a total of *OPS* operations. In the initial state each operation is allocated to a dedicated datapath unit. Further consider a fully testable realisation, with dedicated units for each redundant and comparison operation. This will give a total of $3\times OPS$ functional units. Since TF10 is fed by two functional units, the total number of combinations the algorithm can choose from is given by

$$\binom{3\times OPS}{2} = (3\times OPS)!/2!(3\times OPS-2)! = (3\times OPS-2)\times(3\times OPS-1)/2 \tag{5.12}$$

It is clear that the number of choices increases rapidly as the complexity of the system increases. Taking into account that similar increase is also experienced by the other allocation, scheduling and testing transformations, the consequent increase in the overall computational time is evident. Such long run-time may appear impractical at first and be used as an argument against simulated annealing; however one has to take into account the time-to-market savings if datapath self-checking is applied in an industrial environment. Indeed, in such an environment, 24 hours of *effortless* automatic synthesis is still much more efficient than days of designer effort to manually configure self-checking schemes for hundreds of instructions in the original HDL code, then again manually synthesize with special care to map the functional and checking parts of the schemes on disjoint hardware, and maybe conduct multiple synthesis runs and further HDL modifications to try alternative solutions. On the other hand, it can be predicted that considerably more complex designs than the 288-operation, 32-bit viterbi decoder will require prohibitively long synthesis run-time; it is therefore sensible to state that the biggest designs the proposed technique can practically handle would be composed of around 300 operations. This is still the most complicated ever presented in the self-checking design literature.

5.3.3.6 Discussion

Subsection 5.2.1 critically evaluated the algorithmic duplication literature material and identified points not adequately covered therein. Based on that, the approach of this thesis was defined and implemented. The present subsection conducts an a posteriori evaluation of the numerical data of §5.3.3.5 with respect to results presented in the algorithmic duplication literature. An important word of note is that no "strict" arithmetic comparisons can be drawn, since each past publication uses a different technology, and at times even out-

dated generations of technologies are quoted. The idea that no reliable comparisons can be given is not only sensible, but also advocated by the results in this thesis, showing overhead disagreements between different technologies. For that reason, the comparisons given here are only *roughly indicative* of the quality of considered results.

The rollback and recomputation technique of [60] mostly reports results in the form of RTL functional modules used. This is not an accurate metric, since the area of multiplexers and registers is ignored. A single result is given for a fully implemented VLSI chip; this experiences a hardware overhead value of approximately 170%, which is overwhelmingly more expensive than the vast majority of the results in this thesis. On the other hand, by nature rollback and recomputation imposes strict performance constraints; therefore no additional clock cycles are permitted.

The differentiation-related techniques of [62, 63] report overheads equal to physical duplication [63] or slightly less [62]. Interestingly, in [62] the elliptical filter benchmark was used, also used in this thesis. When 17 clock cycles were used in the DFG, the result of [62] imposed a hardware penalty of about 77%. This value is at times comparable to but still higher than the results herein (indeed, Table 5.17 quotes 29.5% on an FPGA and Table 5.32 gives 50.8% in VLSI).

Introspection [64] gives minimal hardware overhead (always less than 5%). However, bear in mind that the particular technique totally rejects the idea of introducing redundancy for self-checking purposes and purely utilises any naturally existing idle time. At times only a small number of operations are checked (in the formulation of this thesis, P_1 of §5.3.3.1 is at times well below 50%). Therefore, this technique is probably not meaningfully comparable at all with the present material, since in this thesis the goal is *first* full self-checking and *then* as much area saving as possible.

The work of Lakshminarayana et al [23] has already mentioned in §5.2.1 as probably the overall best developed in the background literature. For 9 different benchmarks used, overheads of roughly between 25% - 85% were reported. The results are therefore comparable to those of the present thesis.

The semiconcurrent error detection scheme of [65] reports hardware overheads roughly between 26% and 100% for a checking periodicity of 2 [65], but no performance statistics. Two of the benchmarks used are the elliptical filter and the differential equation solver, also familiar in this thesis. A huge overhead of just over 100% is reported in [65] for the former, while the latter is at the lower end of the overhead range, roughly at 26%. This thesis has given better results for the elliptical filter in both technologies used, and not as good but still comparable (around 35% in Table 5.25) for the differential equation solver. Further recall that the results of [65] were obtained *manually*. It can therefore be stated that the high-level synthesis for on-line testability technique of this thesis *automatically* achieves at times cheaper and much more testable (§5.2) results than those manually derived in the literature.

References [69, 70] always achieve below 30% in hardware overhead. However, testability is greatly reduced since only a percentage of the produced results are checked, and that includes no intermediate results. Further, synthesis is conducted manually and automation is not even mentioned as a future goal. In that sense, comparisons are probably not meaningful.

Finally, [66, 67, 68] also do not concern tool development. Still, the manually obtained results range approximately between 10% and 60%, and are thus on average cheaper, but still comparable to the ones automatically produced herein.

An inspection of all background literature reveals that the value of the operating maximum clock frequency is never reported. However, there should always be a frequency degradation associated with test resource insertion. Even merging two existing functional modules requires multiplexers; this increases the delay of operations, since it lengthens the path that input signals have to traverse before reaching module outputs. This delay degradation necessarily results in clock speed degrading. As the tables of §5.3.3.5 have shown, this work not only acknowledges this frequency degradation, but also fully treats the clock speed as a design parameter, by trading off clock speed through chaining, to devise low cost self-checking solutions that provide a valid option in low frequency projects. This approach is adopted for the first time.

5.4 Summary

In conclusion, in this chapter we have presented a fully automatic integral high-level synthesis for datapath on-line testability approach. The realisation of this approach within the MOODS high-level synthesis system involved :

- implementing five additional transformations that were included in the pre-existing MOODS set of transformations
- developing an elementary software tool for the automatic production of a VHDL library of fault-secure dual-rail checkers and comparators
- defining and formalising a metric for on-line testability, effectively giving rise to a 3-dimensional design space

The particular approach is the first to include *all* of the following :

- test resource insertion is done fully automatically within the HLS optimisation loop; no input HDL modification or other designer interaction is needed
- still, the designer's requirements are taken into account, through his or her choice of priorities; design space exploration is fast and efficient, thus allowing experimentation for alternative priorities
- loops, conditionals and parallel processes are fully accommodated
- instruction chaining is aggressively utilised
- the inversion testing idea is defined and exploited as an alternative to duplication
- alternative technologies are accommodated
- the duplication / inversion self-checking schemes are made fault secure
- all intermediate results are checked; this ensures minimal error latency, and timely reporting of faulty hardware

All this is offered at a hardware overhead and delay degradation that are comparable to and at times cheaper than the experimental results of previous publications.

Chapter 6

Controller Self-checking Design

As high-level synthesis systems become more and more powerful and able to provide solutions for more and more complicated designs, comprising conditional operations, loops and parallel structures, the controllers they produce become more and more complicated and occupy more area on the final chip. Hence, the RTL output of such a system cannot be considered reliable unless an on-line testing scheme for the control path is included in the system. In this context, in addition to the traditional self-checking of data paths, covered in chapter 5 of this thesis, controller checking has recently attained considerable importance as mandatory practice for ensuring the correct operation of controller / datapath pairs, such as the designs output by high-level synthesis systems that are considered throughout this thesis (Figure 3.1). In this chapter, emphasis is given to the self-checking design of the controller part.

The chapter is organised as follows. Section 6.1 reviews the target architecture, states the problem, and briefly describes previously proposed solutions. Section 6.2 examines how parity-based self-checking (§2.2.1.1) can be utilised for controller self-checking purposes, and highlights its properties and limitations. In section 6.3, "1/n" self-checking (§2.2.1.2) is considered as an alternative solution. Section 6.4 discusses the problem in the specific context of the MOODS high-level synthesis system (§3.2), outlines the implementation, presents the obtained experimental results, and gives comments on them. Finally, section 6.5 concludes the chapter.

6.1 Problem statement

This section describes the target architecture and comprehensively states the controller self-checking problem.

6.1.1 Encoded vs. one-hot implementations

Figure 6.1 revisits the typical architecture of a controller / datapath pair. The figure is highly reminiscent of Figure 3.11 – indeed, the same DFG example is used for the datapath. Figure 6.1, however, further reveals the typical controller block structure. In principle, the controller consists of state flip-flops constituting the *state register*, and a block of next-state logic responsible for producing the next-state vector that is to be loaded

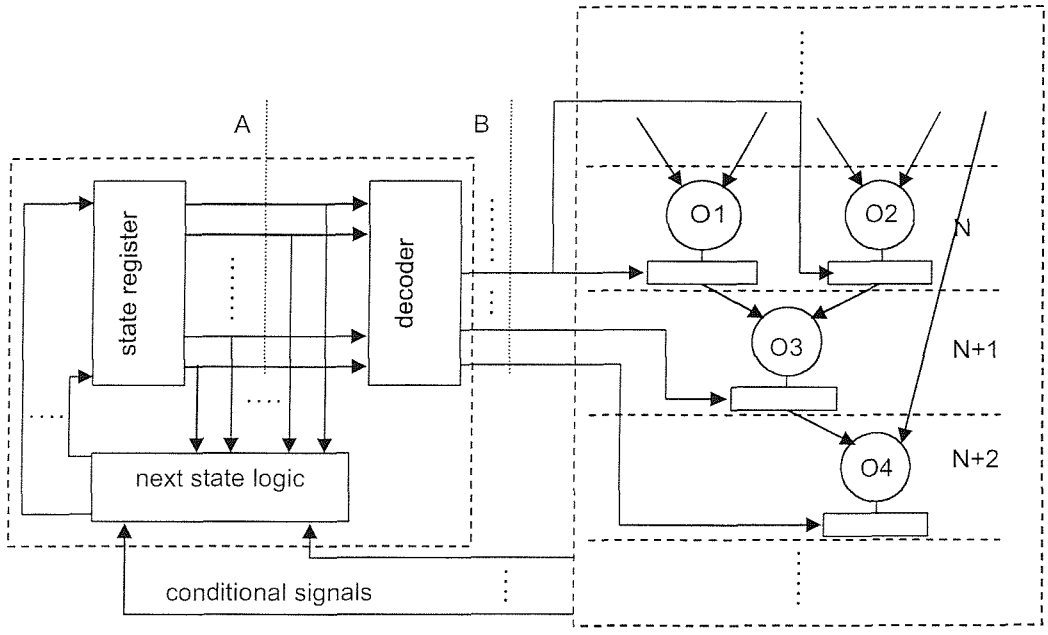


Figure 6.1 : Controller / datapath architecture

onto the state register. Any possibly existing conditional signals also contribute to the production of the next-state vector. The datapath consists of hardware modules that implement instructions scheduled over several control states. Intermediate results are stored in appropriate registers, and are thus preserved across control state boundaries. The analysis of subsection 3.2.6 has established that the controller outputs / control signals (point B of Figure 6.1) should by necessity be one-hot encoded in order for the state transitions to be properly realised. Since the state register contents can, in general, be encoded according to a variety of encoding schemes, a *decoder* is applied (also shown in Figure 6.1) to produce the one-hot control signals. The state register, together with the decoder and the next state logic constitute the overall controller, depicted on the left-hand side of the figure by a dashed rectangle.

From the FSM testability point of view, typically [38, 133, 37, 22, 23, 134] the state signals are encoded according to some coding scheme with enhanced error detection and / or correction capabilities, such as parity [22, 23, 133], Hamming encoding [37], constant Hamming distance [38], or even controller physical duplication [134]. All checking and correcting takes place at the actual state register outputs (point "A" of Figure 6.1). If this is applied in a sequential datapath configuration such as the one at hand, then any possible faults in the decoder are not considered, and are therefore likely to corrupt the actual decoded control signals, resulting in an erroneous sequence of control states, which cannot in principle be detected by datapath hardware module self-checking schemes. Further, the more complicated the encoding scheme, the more complicated the decoding logic, and naturally the more possibilities that a fault may corrupt it. Consequently, if robust reliability properties are to be maintained, it is highly desirable that controller testing take place *after* the decoding operation, that is on the raw one-hot control signals (at point "B" of Figure 6.1). This idea is not only preferable as regards the stated testability concerns, but also disconnects the controller self-checking problem from the controller encoding and controller synthesis problems, allowing the designer to make use of any proposed self-checking solutions regardless of his or her control path synthesis flow (in some cases with some restrictions that will be mentioned in §6.2). For example, Hellebrand et al [133] propose a novel approach that decomposes a long control unit into a collection of shorter ones, communicating among themselves in a pipeline fashion. The approach significantly speeds up the controller. There is no obvious reason why such control path improvement techniques cannot be combined with self-checking solutions discussed in this chapter.

6.1.2 Concurrency

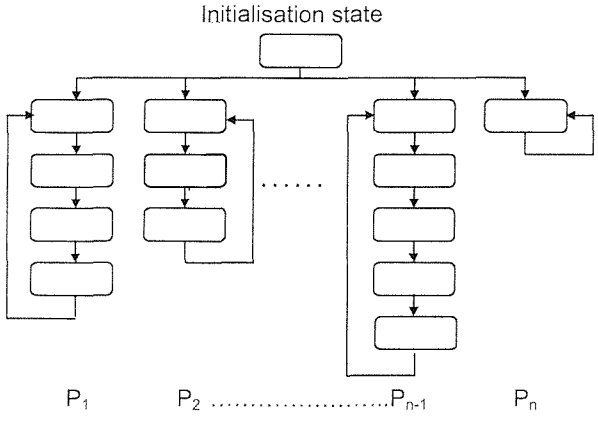


Figure 6.2 : Highly parallel design

Further to the target architecture, when complex digital systems are implemented, it is often the case that they comprise several communicating controller / datapath designs such as the one of Figure 6.1; when the implementation is the result

of a VHDL-based high-level synthesis process, then these structures originate from several synthesisable VHDL concurrent *processes*. These structures often share a single *initialisation* control state, which forks out to several "sub-controllers", as Figure 6.2 depicts. In the figure, rounded rectangles correspond to control states, while vertices naturally show the flow of control, in a manner that closely resembles Petri-nets (§3.1.1). n control paths (P_1-P_n) are shown. Observe the correspondence between Figure 6.1 and Figure 6.2. Each rectangle in 6.2 signifies a separate control state; therefore, a unique control signal (decoder output in 6.1) is dedicated to it. In 6.2, the data path is not shown, and the emphasis is on illustrating the concept of concurrency; in fact, each of the constituent concurrent designs of 6.2 is implemented according to the paradigm of 6.1.

VHDL processes can be arbitrarily long and complex, or they can include as few instructions as can fit within a single control state. The latter is usually the result of a process that simply updates system primary outputs. At any system reset, the initialisation state becomes active, simply meaning that the control signal associated with it assumes the "1" value, while all other control signals throughout all other concurrent designs are at "0". One clock cycle later, control passes to the actual concurrent control paths. From this point onwards (and until the next reset), exactly n (as in Figure 6.2) control signals are at "1". Observe that even single-state control paths are synthesised to comprise *two* states, since they share the common initialisation state with all other control paths in the overall system (e.g. P_n in the figure). Therefore, the 1-hot (in this case, 1-out-of-2) controller output model explained in §6.1.1, is equally applicable regardless of the critical path length of the given design.

While there can be slight variations, the control flow model of this subsection is typical of highly parallel hardware designs. It will therefore be assumed throughout the rest of this thesis. Further, the VHDL "concurrent processes" term will hereafter be used to refer not only to the conceptual descriptions, but also to the resulting communicating controller / datapath pairs that constitute parallel designs.

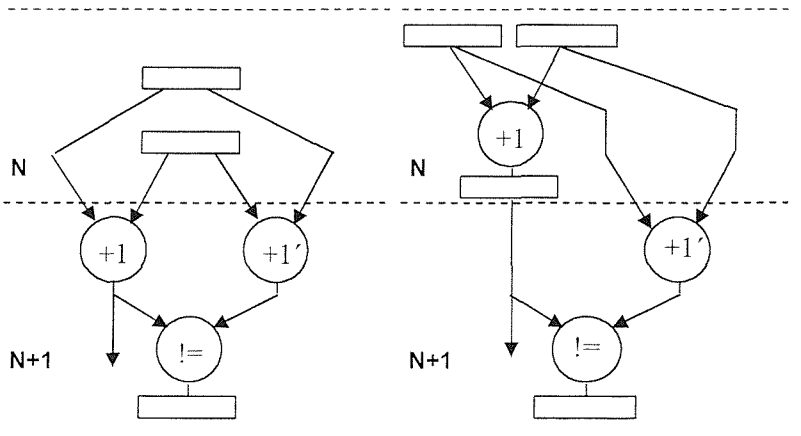
6.1.3 Datapath self-checking constructs reuse

The problem of realising self-checking datapaths through high-level synthesis was comprehensively addressed in chapter 5 of this thesis. Every effort was taken to minimise

hardware penalties; however, even at their minimum, such penalties are inevitable, and sometimes severe. An additional self-checking solution for the control path would involve extra hardware. To this end, it would be desirable to reuse existing datapath self-checking constructs, for controller self-checking purposes. This can be done when (and if) *controller* fault effects are observable in the *datapath.* This is not a new concept; indeed, [135] is a representative example of making controller faults observable in the datapath in the context of the off-line testing of a controller / datapath architecture. In [63], effective controller duplication is proposed and exploited for the same purpose in a self-checking datapath. However, to the best of this author's knowledge, it is the first time that a *low-cost* combined approach is pursued for the on-line, self-checking design problem.

6.1.3.1 Intrinsically Secure states

Consider Figure 6.3a. A portion of a DFG-like representation is shown. A functional operation (addition +1) has been scheduled for control step (CS) N+1. A duplicate operation of the same type, with the same inputs (addition +1′) is also scheduled for parallel execution during the same CS, while the outputs are fed to a fault secure comparator, responsible for verifying correct operation or signalling the presence of a fault. As chapter 5 established, self-checking datapaths can be constructed out of such duplication (and related) testing configurations. Further recall that, assuming a long enough clock period, the additions and the comparison can be scheduled in a single CS (N+1); thus, self-checking is provided at no error latency. It should also be recalled that in the context of a DFG the



(a) Zero-error latency duplication-based self-checking design

(b) Accepting an error latency of a single clock cycle

Figure 6.3 : Securing a control state by accepting datapath error latency

comparator output is also (synchronously) stored in a register, and the register contents (asynchronously) compacted by a dual-rail checker, together with outputs from all other similar comparators present in the design.

Let us move on to Figure 6.3b. In this case, the functional operation has been scheduled a control step earlier, at CS N. Thus, its output is stored in an appropriate register, and the duplicate and comparison operations are executed one clock cycle later. Any fault at the functional operation output will be detected with an error latency of one clock cycle. In the context of this chapter, the following observation is more important than a single clock cycle error latency.

Observation 6.1 : If an induced fault corrupts the control signal that activates state N+1 (i.e. enables the loading of respective registers), such that the said signal behaves as a stuck-at-1, then N+1 will be activated prematurely (i.e. before N, therefore before +1 is executed and its output stored appropriately). Consequently, the comparison operation will not compare the values it has been designed to compare, but two random values (in principle unequal), and therefore it is likely to produce an error indication. Thus, a *controller* fault will be detected through the existing *datapath* self-checking scheme.

There is always a possibility of fault escapes, if the random values mentioned above coincide. This will be ignored for the moment. For the time being, the following definition is provided.

Definition 6.1: A control state is referred to as *Intrinsically Secure* (*IS*), if the comparison (checking) part of a datapath self-checking scheme has been scheduled in it, but at least one of the functional or redundant parts of the scheme has been scheduled in previous states.

In other words, a state in the situation of CS N+1 in Figure 6.3b is IS by definition. The discussion up to now has been restricted to duplication testing; however, the same concepts can be applied to any self-checking scheme that can have its computation and checking parts separated across the boundary of two different control states. This includes inversion testing; therefore, the IS-states idea is fully compatible with the implementations of chapter 5. Also, note that in Figure 6.3, control step N pre-existed in the design, and some

operations were probably scheduled in it. Therefore there was no actual delay degradation by moving the operation and securing CS N+1. In the context of a realistic design, this may not always be possible (due to data dependencies) and some delay degradation may need to be accepted, but it is expected to be in principle tolerable. It is further worth observing that moving +1 to CS N, necessarily (re-)introduces a register to store the result across the CS boundary. This means a hardware overhead; it is therefore likely that at times the hardware savings due to the simplification of the controller self-checking scheme (shown later in §6.2, §6.3) will be cancelled out by the register overhead. On the other hand, there are cases where Intrinsically Secure states appear in self-checking designs naturally, and therefore exploiting their controller self-checking potential is free.

The concept of control states that are Intrinsically Secure according to Definition 6.1 is a particular contribution of this thesis. The area and delay overhead concerns stated in the above paragraph can only be answered through experimentation individually for any given design. The experiments of §6.4 will investigate a number of designs and, among others, provide an insight on this issue.

6.1.3.2 The possibility of fault escapes

Let us go back to Figure 6.3b and comment on the probability of a $\overline{D}$ (0/1) type error [1] on the control signal corresponding to control state N+1 to remain undetected, due to the possibility that the output of the duplicate operation may coincide with the contents of the (improperly loaded) register that stores the functional operation result under fault-free operation.

Assume that the bit-width of operation "+1" in Figure 6.3b is w. Then 2^w different words can appear in the left hand side input of the comparator. Assuming that all words have the same probability, this probability for a particular word is equal to $1/2^w$. Therefore, given the value that "+1′" prematurely computes during N+1, and the functional operation bit-width w, the probability of a fault escape can be estimated as $p_e=1/2^w$. For example, for w=3, p_e=12.5%, which is unacceptably high. Based on the above, Definition 6.1 can be updated as follows :

Definition 6.1' : A control state is referred to as *Intrinsically Secure* (*IS*), if the conditions of Definition 6.1 hold, and in addition the bit-width of the functional operation is higher than a defined threshold value t.

A sensible value for the threshold would be, e.g., at least t=7, which gives $p_e \approx 0.8\%$. This choice is motivated by the usual convention of traditional testing, whereby a testing scheme is considered successful when it detects 99% of the modelled faults [1]. Of course, in the context of the problem at hand it makes sense to differentiate between single-bit logic operations and multi-bit arithmetic operations. While it would be unwise to speak about Intrinsically Secure states when referring to the former (as these would have an escape probability of 50%), such states can be defined for arithmetic operations, experiencing escape probabilities of 0.4%, 0.002% and 2×10^{-8}% for the usual choices of 8-, 16- and 32-bit arithmetic respectively.

A practical precaution which can be applied in order to minimise the possibility of fault escapes in IS states, is to reset the register that carries the functional output value across the CS boundary, to a value that is highly unlikely to occur, as soon as its functional content is not needed anymore. Such a typical value can be the all-1s pattern for unsigned arithmetic operations. Normally the appearance of this pattern during normal operation is an indication of (potential) overflow, and it should not appear if careful design has been applied.

6.1.4 Discussion

Subsections 6.1.1, 6.1.2, and 6.1.3 define the backbone of the problem at hand. In summary, the controller self-checking problem addressed in this chapter has the following characteristics :

- self-checking should be applied to the decoded 1-hot controller outputs
- multiple concurrent processes should be handled efficiently
- the idea of Intrinsically Secure states can be exploited, in an attempt to minimise overheads
- generally, the controller self-checking scheme should be as economical as possible, given the penalty related to the (assumingly existing) datapath self-checking; at the same

time, consistency with self-checking design theory is desirable (totally self-checking property, §2.2.1)

Given the 1-hot encoding restriction, solutions that impose particular encodings (such as Hamming in [37]) are not applicable. Duplicating the controller [134] is rejected, since it is expected to give very expensive results; indeed, the hardware sharing potentials exploited in chapter 5 do not exist in the control path case (i.e. there is no equivalent to "idle hardware module cycles", and no ground for "algorithmic" duplication in the control path). Observe, though, that a 1-hot encoded n-bit signal maintains odd parity. Further, parity-based self-checking (§2.2.1.1) is known to be the cheapest among error-detecting solutions; it has already been proposed for control path self-checking in [23], albeit only a short note was dedicated to this issue. It is considered in detail in this thesis in the following §6.2. 1-out-of-n and / or m-out-of-n self-checking would also appear to be feasible solutions for the given problem. At a first glance, one would expect them to be more expensive than parity; in §6.3 we discuss this issue.

6.2 Parity-based self-checking

In this section, parity-based controller checking in the context of highly parallel synthesized controller / datapath designs is addressed. Recall that parity checking of a bit vector detects all faults in the system producing the vector, that result in single- or odd-multiplicity logic errors in the vector. Regarding the problem at hand, and referring back to Figure 6.1, a parity checker at point "B" will detect all controller faults that give rise to a single or an odd number of corrupted control signals. Combined with Hypothesis 2.1 (faults occur one at a time), this means that the controller has to be designed such that no single fault in it can result in an even number of corrupted bits at the controller output. Normally the easiest and most straightforward way to achieve this, is to disallow logic sharing between the logic cones that produce each one of the control signals, replicating some logic operations in the next state logic and decoder blocks if necessary [37, 22]. Other than that, the techniques presented in this section are generic, and applicable to any controller encoding and synthesis approach. The approach itself normally is dictated by the target technology and any particular constraints. More specifically, if the state register is designed to be one-hot as such (for example, as in MOODS, §3.2.6), then the next state logic block is simple, while a decoder is not needed. The implementation is fast; however,

the large number of flip-flops needed may lead to expensive realisations. The number of flip-flops is dramatically reduced if suitable encoding is applied; the complexity of next state logic and decoder are, however, increased. The resulting controller is also considerably slowed down. In addition, there are technologies for which a plethora of storage element resources are available (e.g. some FPGAs [106]), therefore the direct one-hot implementation may not always be as expensive as it first appears.

In the rest of this section, and unless otherwise stated, it will be assumed that the controller has been designed taking into account the above note about odd error multiplicity.

6.2.1 Per process parity-based self-checking

Consider a design like the one of Figure 6.2, consisting of n concurrent processes ($P_1,\ldots,P_n$), each one consisting of m_i ($0<i\leq n$) control states, plus the common initialisation state, hereafter state-0. Parity-based self-checking design can be straightforwardly imple-

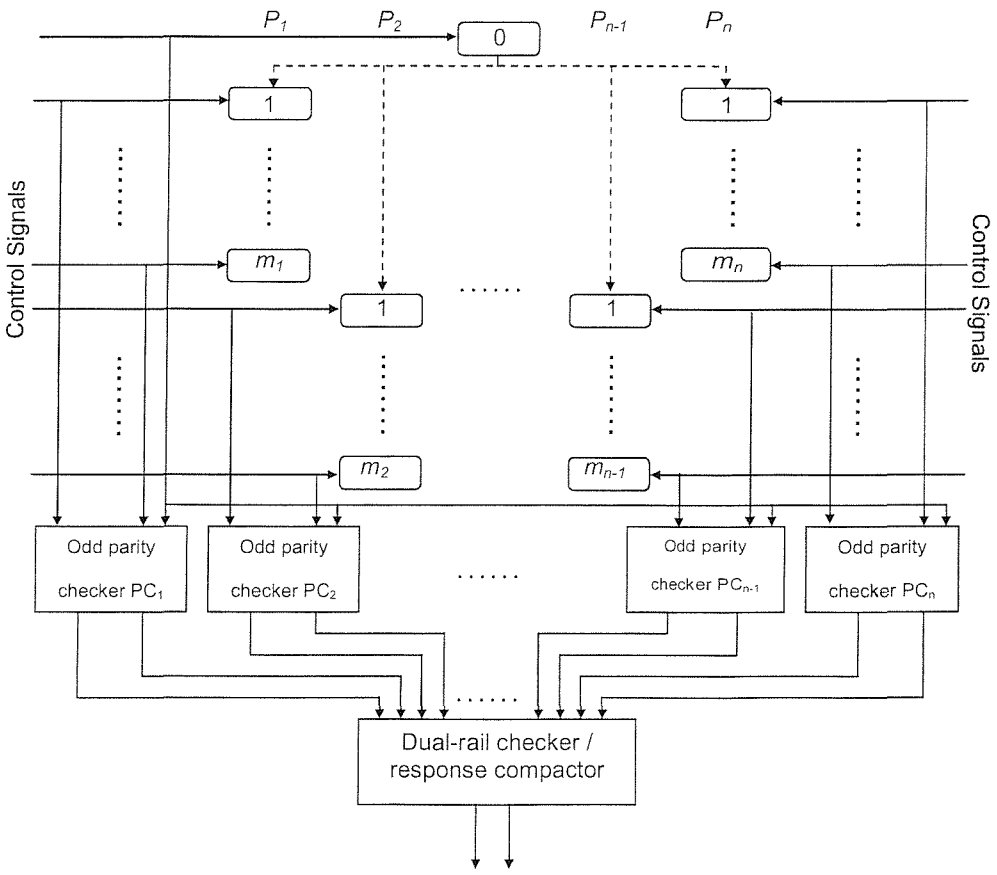


Figure 6.4 : The CTRL_1 self-checking scheme

mented as Figure 6.4 shows. Dashed lines in the figure correspond to the control flow, while solid lines represent actual signals coming from the controller block. In order to understand Figure 6.4, notice its correspondence with Figures 6.1 and 6.2. The system concurrent processes are shown in 6.4 effectively in the same fashion as in 6.2. Clearly, a unique control signal from the controller of Figure 6.1 corresponds to each state, as established in §6.1. This is graphically depicted in Figure 6.4 by a single signal line shown to end at each rectangle representing a control state. Signal lines also fan out to appropriate checkers, thus showing the considered self-checking scheme. As can be observed, every process has its control signals checked by a separate odd parity checker, and all responses are compacted by a dual-rail checker, as is the usual practice in self-checking design (§2.2.2.2). The control signal corresponding to the state-0 becomes active only upon system reset, and is fed to all parity checkers. Thus, at any given point of time each parity checker receives a one-hot signal at its input, and therefore detects any single- or odd-multiplicity errors. This scheme will hereafter be referred to as the *CTRL_1* self-checking scheme.

The actual odd parity and dual-rail checker structures are not detailed yet; for the moment, let us make the assumption that all checker components are double-output, composed of two-input gates only. This assumption is in absolute agreement with the usual checker designs presented in §2.2.1.1 and §2.2.2.2, and it implies that the usual 2-input XOR gates and dual-rail checker cells are used for the parity and dual-rail checkers respectively. No assumption is, however, made at this point regarding the arrangement of gates and cells within the checkers. This approach will be adopted for the moment and until §6.2.7, where a few structure-related considerations are given.

Based on the above assumption, the hardware cost of CTRL_1 can easily be estimated as follows. n parity checkers ($PC_1,\ldots,PC_n$) are used, each one consisting of two XOR trees, to ensure the fault secure property [5]. Any random checker PC_i has m_i+1 inputs (all states in the corresponding process, plus the common state). Every k-input parity tree is composed of k-1 XOR gates, therefore PC_i consists of m_i-1 XOR gates. Further, the dual-rail checker has n input pairs; therefore it consists of n-1 dual-rail checker cells (§2.2.2.2), which yields 6×(n-1) AND/OR gates. In total, the hardware cost for this technique is given by the following expression :

$$Cost_{CTRL\_1} = \left[\sum_{i=1}^{n}(m_i - 1)\right] \times Cost_{XOR} + 6 \times (n-1) \times Cost_{AND/OR} \quad (6.1)$$

where $Cost_{XOR}$ and $Cost_{AND/OR}$ refer to the hardware costs of respective gates, and the implicit assumption is that under the particular target technology an AND and an OR gate have the same cost. When this is not true, the above expression can easily be suitably amended.

Let N_s be the total number of states in the design. Clearly

$$N_s = \sum_{i=1}^{n}(m_i) + 1 \quad (6.2)$$

Further

$$\sum_{i=1}^{n}(m_i - 1) = \sum_{i=1}^{n}(m_i) - n = N_s - (n+1) \quad (6.3)$$

Equations (6.1) and (6.3) yield :

$$Cost_{CTRL\_1} = [N_S - (n+1)] \times Cost_{XOR} + 6 \times (n-1) \times Cost_{AND/OR} \quad (6.4)$$

Equation (6.4) gives the hardware cost of the CTRL_1 self-checking scheme for the design, as a function of the number of processes, the total number of control states, and of the target technology and specific gate implementations.

6.2.2 Self-checking using a single parity checker

Using parity checking necessarily results in a number of XOR gates that is of the order of N_s as defined above, and cannot be dramatically decreased. However, the dual-rail checker may be considered redundant if the checking scheme of Figure 6.5 is used. In this approach, all control signals are led to a single parity checker. At reset only the state-0 control signal will have a logical 1 value; at any other point of time the number of 1s will be equal to the number of processes, n. If n is odd by design (n=2k+1), then odd parity is naturally maintained at all times. If n=2k, then a single-state "dummy" process is inserted. No instruction is executed in this process; an additional control signal is, however, generated by the controller for it, and so odd parity is maintained for the controller output. This is the *CTRL_2* self-checking scheme.

The odd parity checker has N_s inputs. The hardware cost is given by :

$$Cost_{CTRL\_2} = (N_s - 2) \times Cost_{XOR} \quad (6.5)$$

Equation (6.5) is accurate only for n=2k+1; otherwise it is approximate. Particularly, it ignores both the area overhead of introducing the dummy control state to the design, and the corresponding additional input to the parity checker. However, in the usual case that N_s>>1, the overhead contribution of these two elements can sensibly be considered negligible.

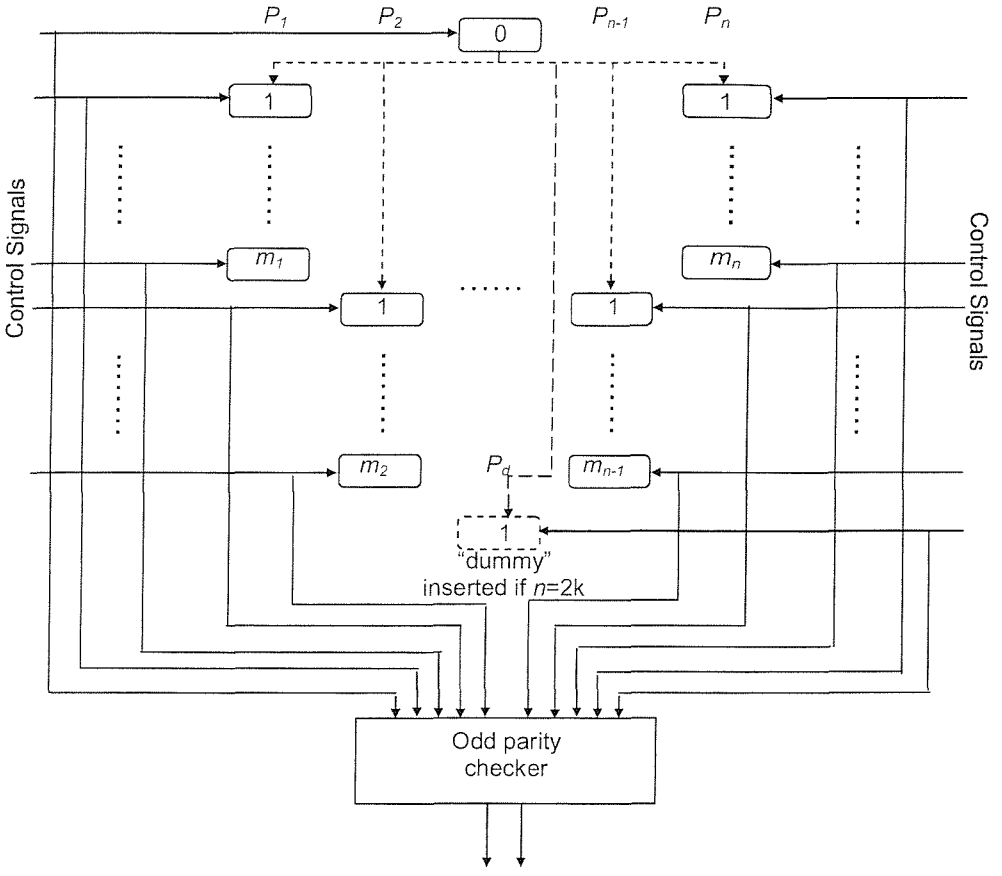


Figure 6.5 : The CTRL_2 self-checking scheme

It is interesting to note that if n=1, then equation (6.4) and equation (6.5) yield the same value $(N_s\text{-}2)\times Cost_{XOR}$. This is expected, since it is obvious by simple comparative inspection of Figures 6.4 and 6.5 that for a single process both CTRL_1 and CTRL_2 correspond to a single parity check.

6.2.3 Utilising Intrinsically Secure states in a single process

In this subsection, as well as in the next two ones §6.2.4 and §6.2.5, it is assumed that the design datapath has been synthesized such as to demonstrate self-checking properties (for

example, as in chapter 5). With this assumption in mind, the self-checking resources inserted for the purpose of datapath checking are identified to be utilisable for the purpose of providing cheaper self-checking for the control signals as well, by exploiting the Intrinsically Secure states concept introduced in §6.1.3. The motivations for this approach lie in Observation 6.1 which was made on designs produced in chapter 5 of this thesis. However, they are generic enough to be equally applicable in alternative environments and design flows.

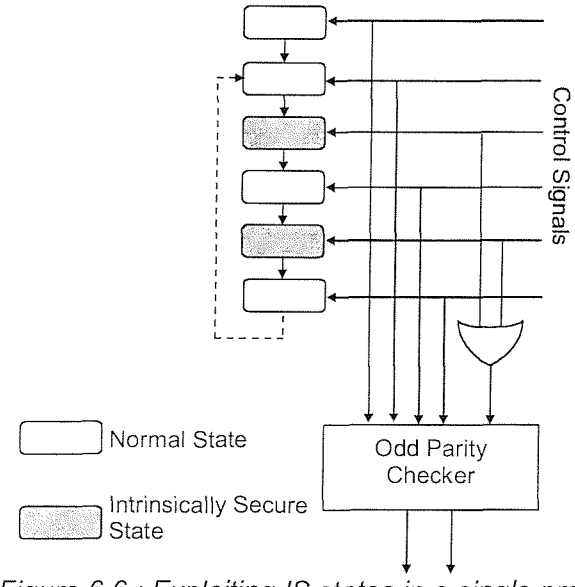


Figure 6.6 : Exploiting IS states in a single process with parity-based controller self-checking

Figure 6.6 focuses on a single process, possibly by isolating any of the concurrent processes of Figure 6.2. It is further assumed that a number of IS states (in the sense of definition 6.1′) are identifiable within this process. The figure exemplifies two such states, clearly distinguishing them from the non-IS states. Control signals are shown in a manner similar to Figures 6.4 and 6.5. A scheme for the utilisation of IS-states for the purposes of the problem at hand is further shown in Figure 6.6. Particularly, control signals from IS states are compacted using an OR gate, and the resulting signal is fed to an odd parity checker, together with the control signals corresponding to non-IS states.

Theorem 6.1 : The configuration of Figure 6.6 detects all single control signal faults, while providing the fault-free indication under fault-free operation.

Proof :

a) Consider the case when one of the IS state control signals is active :

a1) Under fault-free operation, since one of the IS state control signals is active (logic 1), the OR output is a logic 1; since the controller is one-hot, all control signals corresponding to non-IS states are 0. Therefore, the parity checker is fed by a one-hot pattern, and correctly detects odd parity.

a2) If the control signal of an inactive IS state assumes the $\overline{D}$ value (chapter 2 and [1]), therefore erroneously becoming a 1 when it should have been a 0, then the parity checking scheme of Figure 6.6 does not detect the fault; however, since the said state is Intrinsically Secure, the fault is detected by the corresponding checker in the data path (Figure 6.3).

a3) If the control signal of the active IS state erroneously fails to take the active (logic 1) value, and is stuck-at-0 instead (D value [1]), then the OR gate output is at logic 0. At the same time, all non-IS control signals are 0, and the parity checker detects the erroneous (even) parity.

a4) If a non-IS control signal takes the value $\overline{D}$, then since the OR gate outputs 1, the checker is fed by a two-hot type input, which is of even parity, and therefore detects the fault.

b) Now consider the case when one of the non-IS signals is active.

b1) Under fault-free operation, the OR gate outputs logic 0, since all IS control signals are inactive. Therefore only one of the parity checker inputs is 1. The parity is odd and correct operation is confirmed.

b2) If an IS state control signal assumes the $\overline{D}$ value, then the OR gate output erroneously changes to 1. Therefore the parity checker (being fed by a two-hot type signal) detects the fault. Further, since the state is Intrinsically Secure, the data path checker also detects the fault. This double-check property increases the dependability of the system.

b3) If a non-IS state control signal assumes the $\overline{D}$ value, then there are two 1s in the checker input, both coming from the non-IS control signals, since the OR gate outputs 0. The parity is even, and the fault is detected by the checker.

b4) Finally, if the active non-IS state control signal fails to take the logic 1 value, and assumes the D value instead, then the checker is fed by a 0 from the OR gate, and by all-zeros from the non-IS states. Once more, the parity is even and the fault is detected. ▲

The key point in the above proof, that in fact clarifies the benefit of exploiting IS states, is a2 : parity *fails* to detect the fault, but this *does no harm*, since error detecting capabilities for the considered type of fault exist in the datapath. Therefore, the controller checking scheme is simplified, through the abolishing of error detection capabilities that are not needed, resulting in some hardware savings. This is achieved by dropping a number of

XOR gates that are used within the checker when straightforward parity checking is applied, and using an OR gate with a suitable number of inputs. An additional (and in fact more important) benefit of this approach, is that errors of *any* multiplicity in control signals can be detected, provided that one of them corrupts an IS state signal. Thus, the odd-multiplicity error detection limitation of parity is overcome. Backtracking to the odd multiplicity-related note in the beginning of §6.2, it can now be understood that, by utilising IS states as shown above, the designer can allow hardware sharing between control signal cones of logic, *provided* that at least one IS state can be identified among the signals for whose logic cone sharing is applied. This is expected to be another source of hardware savings.

6.2.4 Per process parity-based self-checking exploiting Intrinsically Secure states

Based on the material of §6.2.3, an overall self-checking scheme for a parallel design can

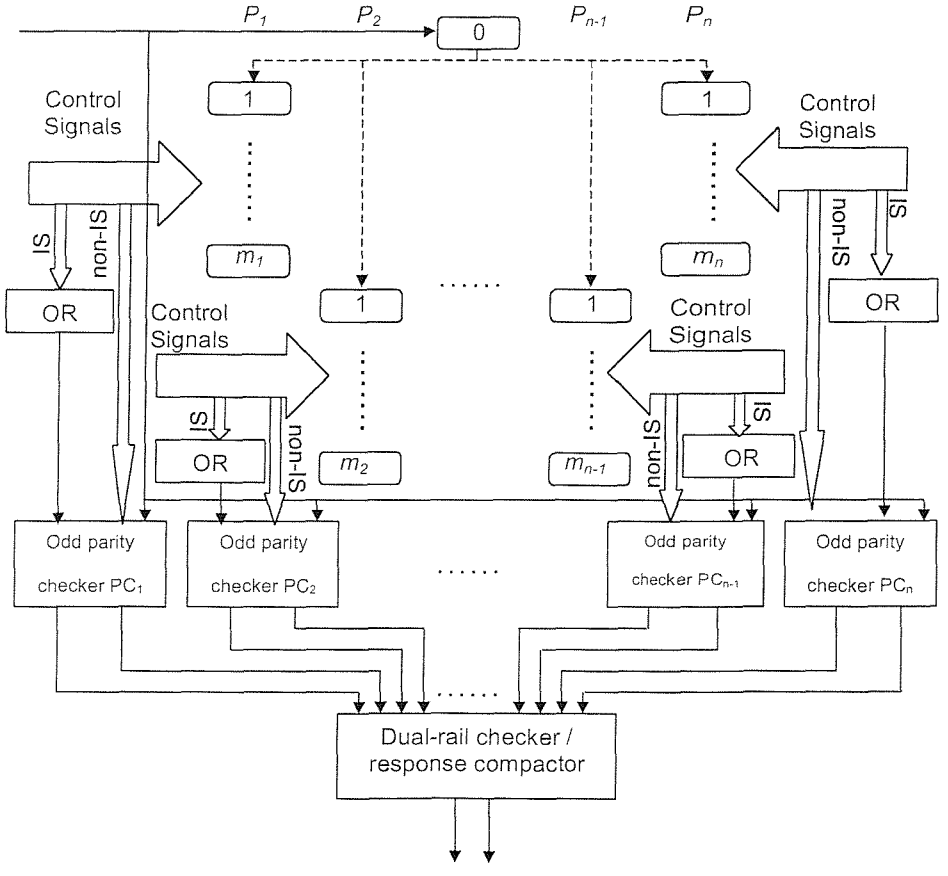


Figure 6.7 : The CTRL_3 self-checking scheme

be configured as Figure 6.7 shows. The figure follows the notations of the previous Figures 6.4 and 6.5; however, here signal buses have been substituted by block arrows for convenience. Control signals reaching each process are separated into two groups, corresponding to IS and non-IS state control signals, with each group separately treated as §6.2.3 suggests. Indeed, the IS-state group signals are ORed, and the result feeds the process parity checker, where it meets all other signals from the non-IS group. The initialisation state signal is once more fed to all parity checkers, since no actual operations take place during initialisation, and therefore it cannot possibly be Intrinsically Secure. The scheme of Figure 6.6 is thus separately applied to every process; parity checker responses are naturally compacted by a dual-rail checker. If no IS states can be identified in a given process, then the OR gate is redundant, and theoretically replaced by a constant logic 0. Since a constant 0 does not change the parity of the overall signal, it is safely omitted. The overall configuration will hereafter be referred to as the *CTRL_3* scheme.

Exactly like in the purely parity-based schemes, an estimation of the CTRL_3 hardware cost is attempted here. For this purpose, let us define $m_{i,S}$ and $m_{i,N}$ as the number of IS and non-IS (respectively) states of process P_i. Clearly $m_{i,S}+m_{i,N}=m_i$, as defined in §6.2.1. Each parity checker PC_i has $m_{i,N}+2$ inputs (therefore $m_{i,N}$ XOR gates) if $m_{i,S}\neq 0$, and $m_{i,N}+1$ inputs (therefore $m_{i,N}$-1 XOR gates) if $m_{i,S}=0$. Further define $n_{IS}\leq n$ as the number of processes that include at least one IS state. The total number of XOR gates needed will be equal to

$$\sum_{i/m_{i,S}\neq 0}(m_{i,N})+\sum_{i/m_{i,S}=0}(m_{i,N}-1)=\sum_{i=1}^{n}(m_{i,N})-(n-n_{IS}) \tag{6.6}$$

Further, all OR gates are of $m_{i,S}$ inputs. Also, the response compactor compacts n input pairs, for a hardware cost of 6×(n-1) 2-input AND/OR gates. Overall, the cost is given by

$$Cost_{CTRL\_3}=\left[\sum_{i=1}^{n}(m_{i,N})-(n-n_{IS})\right]\times Cost_{XOR}+6\times(n-1)\times Cost_{AND/OR}+\sum_{i=1}^{n}C_{OR}(m_{i,S}) \tag{6.7}$$

where function $C_{OR}(k)$ denotes the hardware cost of a k-input OR gate.

If there is no IS state in any process, it can easily be verified, by comparison of Figures 6.4 and 6.7, that CTRL_3 becomes equivalent to CTRL_1. This can also be seen in equation (6.7), substituting $m_{i,S}=0$, and $m_{i,N}=m_i$ for all i. In this case, and taking into account the definition of N_s through equation (6.2), equations (6.4) and (6.7) yield the same value.

6.2.5 Parity-based self-checking using a single parity checker and exploiting Intrinsically Secure states

The next controller self-checking design scheme presented here is naturally a combination of CTRL_2 and CTRL_3. It is depicted in Figure 6.8, and will be called *CTRL_4*. As is obvious from the figure, all non-IS states from all concurrent processes, plus all OR gate outputs compacting IS state control signals are fed to a single odd parity checker.

Lemma 6.1 : The configuration of Figure 6.8 detects all single control signal faults, while providing the fault-free indication under fault-free operation.

Lemma 6.1 is a generalization of Theorem 6.1, and it can be informally verified as follows. During reset, only the initialisation state is active, thus a one-hot signal reaches the parity checker, and the correct operation is confirmed. During all subsequent control states, each process will contribute a logic 1 either because of one of its non-IS state signals, or as the output of corresponding OR gates. So a total of n 1s will feed the parity checker. Therefore, exactly as in the CTRL_2 technique, a single-state "dummy" process is inserted to ensure odd parity, in case n=2k. The above statements apply during fault-free operation, verifying that in that case the scheme produces the fault-free indication; under a single fault in any control signal of any process, the process at hand will either

- erroneously contribute an additional 1 (see a4 and b3 in the proof of Theorem 6.1), thus accumulating an even number of 1s (2k+2) in the checker input (the checker will therefore detect the fault), or
- fail to produce its corresponding 1 (cases a3 and b4 as above), again leading to an even number of 1s (2k) fed to the checker, thus again asserting the faulty indication, or
- produce its fault-free control signal, *but* signal a fault at its data path (case a2), or even
- produce *both* an additional 1 at its control signals and an erroneous signal at the data path (b2), thus giving a double alarm.

The validity of Lemma 6.1 is thus verified.

The odd parity checker inputs are all non-IS state control signals of the design (a total of $\sum_{i=1}^{n}(m_{i,N})+1$ bits, including state-0), plus one signal for every process that has at least one IS state (as defined above, there are n_{IS} such processes). Based on this observation, the fol-

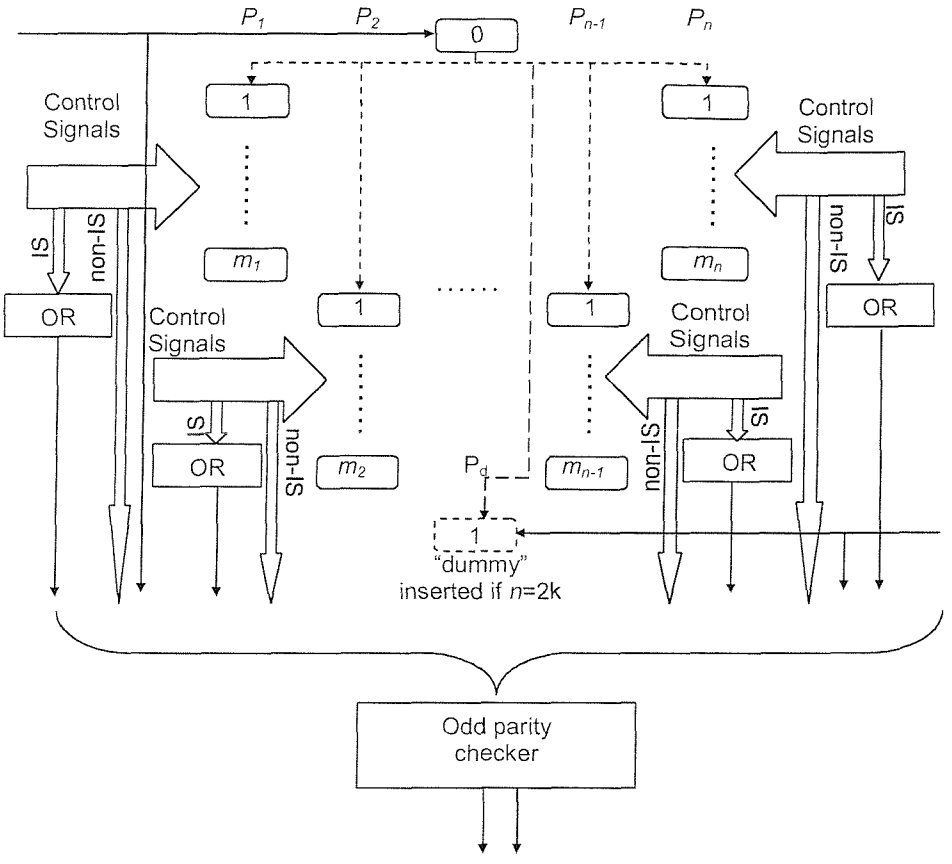


Figure 6.8 : The CTRL_4 self-checking scheme

following expression can be derived, giving the hardware cost estimation for the CTRL_4 scheme.

$$Cost_{CTRL\_4} = \left[\sum_{i=1}^{n}(m_{i,N}) + n_{IS} - 1\right] \times Cost_{XOR} + \sum_{i=1}^{n} C_{OR}(m_{i,S}) \qquad (6.8)$$

Equation (6.8) takes into account the overhead from both the XOR gate-based checker and the OR gates relevant to IS states, but, like (6.5), it ignores overheads associated with the dummy state insertion, in the case of an even n.

6.2.6 Hardware costs

This section attempts a comparison of the four techniques presented in the previous sections in terms of their hardware cost, assuming CMOS VLSI target technology [136]. In this technology, it is known that typically $Cost_{AND/OR}$=6 transistors (implemented as a 2-input NAND/NOR followed by an inverter), and $C_{OR}(k)=2\times(k+1)$ transistors (imple-

mented as a k-input NOR followed by an inverter). It is also assumed that the XOR gates are implemented as transmission-gate XORs, thus yielding $Cost_{XOR}=6$ transistors [136]. It can be argued that the transmission-gate XOR, although particularly cheap, is not the best implementation of an XOR function; indeed, the realisation using three NAND gates and two inverters is usually prefered by most designers. Likewise, with respect to the k-input OR realisation, for high values of k, k-input NOR gates may be too slow for a particular technology, and implementations using multiple 2- or 3-input NORs may be preferred instead. However, the present cost assumptions are purely for the purpose of illustrating the relative theoretical costs derived for the corresponding schemes and they are useful as such.

Table 6.1 summarizes the CMOS transistor count estimations for some sets of values of the associated parameters, for all four schemes, as given by equations (6.4), (6.5), (6.7), and (6.8). To facilitate easier understanding of the figures in the table, the meanings of parameter symbols defined in the previous sections are repeated in the following.

N_S total number of control states in the design (including initialisation)

n total number of concurrent processes

n_{IS} number of processes that include at least one Intrinsically Secure state

$m_{i,N}$ number of non-Intrinsically Secure states in process i

$m_{i,S}$ number of Intrinsically Secure states in process i

Parameters					Checker transistor count			
N_S	n	n_{IS}	$m_{i,N}$, for $1\le i\le n$	$m_{i,S}$, for $1\le i\le n$	CTRL_1	CTRL_2	CTRL_3	CTRL_4
50	1	0	49	0	288	288	288	288
20	1	1	15	4	108	108	100	100
30	3	0	13, 15, 1	0 for every i	228	168	228	168
151	15	15	5 for every i	5 for every i	1314	894	1134	714

Table 6.1 : Self-checking hardware cost estimations

The first row of the table corresponds to a single-process design, with no Intrinsically Secure states. The expected result is that all techniques yield the same cost, since they all lead to a simple parity check. The second row corresponds to another single-process design; this time, however, it is possible to identify four Intrinsically Secure states within the process. The result is a slightly cheaper checker when CTRL_3 or CTRL_4 are used, on top of the increased error detection capabilities mentioned in §6.2.3. Naturally, CTRL_1 and CTRL_2 effectively lead to the same design, and so do CTRL_3 and CTRL_4. In the

third row, a parallel (3-process) design with no Intrinsically Secure states is considered. Significant hardware savings are noticeable when the single-parity checking schemes CTRL_2 and CTRL_4 are used. Finally, the fourth row depicts the most complicated case of a highly parallel (15 processes) design, with identifiable Intrinsically Secure states in all processes. In particular, the assumption is that exactly half (5/10) of the states in each process are IS. Such cases can appear in realistic, useful designs, implementing complex digital signal processing algorithms. Comparing the estimations for the CTRL_1 and the CTRL_4 schemes, an improvement of the order of 45% can be noticed.

It should be stressed that the estimations of this subsection are *not* experimental results; rather, they are an attempt to evaluate the theory of this section in the light of some hypothetical but possible design scenarios. They simply give a flavour of the expected properties of the self-checking choices presented so far. Experimental evaluation is still necessary, especially given that the results of Table 6.1 ignore the presence of the datapath, and the effect on the datapath area that each controller self-checking choice may imply. Such implementation results are given in §6.4.

Nevertheless, the above estimations verify that controller checking using a single checker can lead to more compact implementations (CTRL_2, CTRL_4). Naturally, the higher the degree of concurrency (n), the more significant the improvement. Noticeable savings (~26%) appear in Table 6.1 even for n=3 processes. However, recall that the hardware cost of the data path is not shown in the table. Realistically, it can be estimated that the hardware savings will become really important for a number of concurrent processes of the order of $n \approx 10$. As regards the schemes exploiting IS states (CTRL_3, CTRL_4) versus their pure parity counterparts (CTRL_1, CTRL_2), Table 6.1 suggests that the hardware savings associated with them are rather modest; therefore, the improved reliability, stemming from overcoming the odd multiplicity fault detection limitation, should be kept as their main advantage.

6.2.7 Achieving the totally self-checking goal

All four techniques considered in this section employ parity checking to a greater or lesser extent. Since parity checking properties have seen extensive theoretical investigation

(§2.2.1.1), it is desirable to evaluate the presented schemes with respect to self-checking theory as well.

As a first word of note, parity checking for a 1-hot encoded signal is not – strictly speaking – code-disjoint (Definition 2.4). Indeed, an n-bit 1-hot encoded signal demonstrates odd parity, *but* so does an n-bit signal with three (or any other $2k+1>1$ number of) 1s among its bits. Such a signal may be the result of a highly hostile environment, causing two (or an even number of) faults on the signal, and consequently resulting in a situation where a non-code (i.e. not 1-hot) checker input produces a code (fault-free indication) output. However, the underlying single-fault Hypothesis 2.1, backed by the comments of §6.2 regarding separate cones of logic for every controller output, rules out such a situation. Parity checking for the 1-hot controller outputs can, therefore, in this background, be loosely regarded as a code-disjoint operation. Fault-security (Definition 2.2) can likewise be confirmed.

Regarding the self-testing property (Definition 2.1) also required for the totally-self-checking goal to be achieved, recall Lemmas 2.1 and 2.2 (§2.2.1.1 and [17, 18]). According to them, a parity-based self-checking scheme is guaranteed to be self-testing if the checker receives either

- 75% of all possible code words, or
- the rows of a $4\times n$ matrix, whose distinct columns have exactly two 1s and two 0s.

In contrast to the previous paragraph, "code words" here refers to all odd parity encoded n-bit words, rather than to all 1-out-of-n words. The words of an n-bit 1-hot code are always n, while there are $2^n/2=2^{n-1}$ different odd parity encoded words in total. Clearly it is $2^{n-1}\times 75\%>n$ for all $n>3$, therefore the first condition cannot be true in the case at hand, except for the trivial case $n=3$. Moreover, there can be no two different available code words that have a 1 at the same bit position. This means that a matrix such as the one of the second condition cannot be constructed from the available code words of the considered case. One is therefore forced to conclude that the presented schemes at their current form are not self-testing; consequently, they are not totally self-checking either.

Recalling §2.2.1.1, the solution to this problem is self-exercising checker design. This is directly applicable in this case, by simply substituting the conventional 2-input XOR gate-based parity checkers implied in Figures 6.4, 6.5, 6.7 and 6.8 with the LFSR-based struc-

ture of Figure 2.12, repeated (slightly modified) in Figure 6.9 for convenience. The dashed rectangle in the figure outlines the overall checker structure to be used in the CTRL_1, CTRL_2, CTRL_3 and CTRL_4 schemes. Recall that only the even parity code is a linear code. In practice, this means that the LFSR of Figure 6.9 must "internally" be based on even parity encoding. This is in contrast to the situation at hand, where checker inputs demonstrate odd parity. Therefore, in line with Tarnick's advice [12], two inverters are applied to the structure of Figure 2.12, as Figure 6.9 depicts. The first one is applied to an arbitrary bit of the checker input (in this case, the input LSB), so that the LFSR is fed with the required even parity words. The second one is applied to the (even parity) LFSR output, once again to an arbitrary bit (again the LSB in the figure), to produce the odd parity

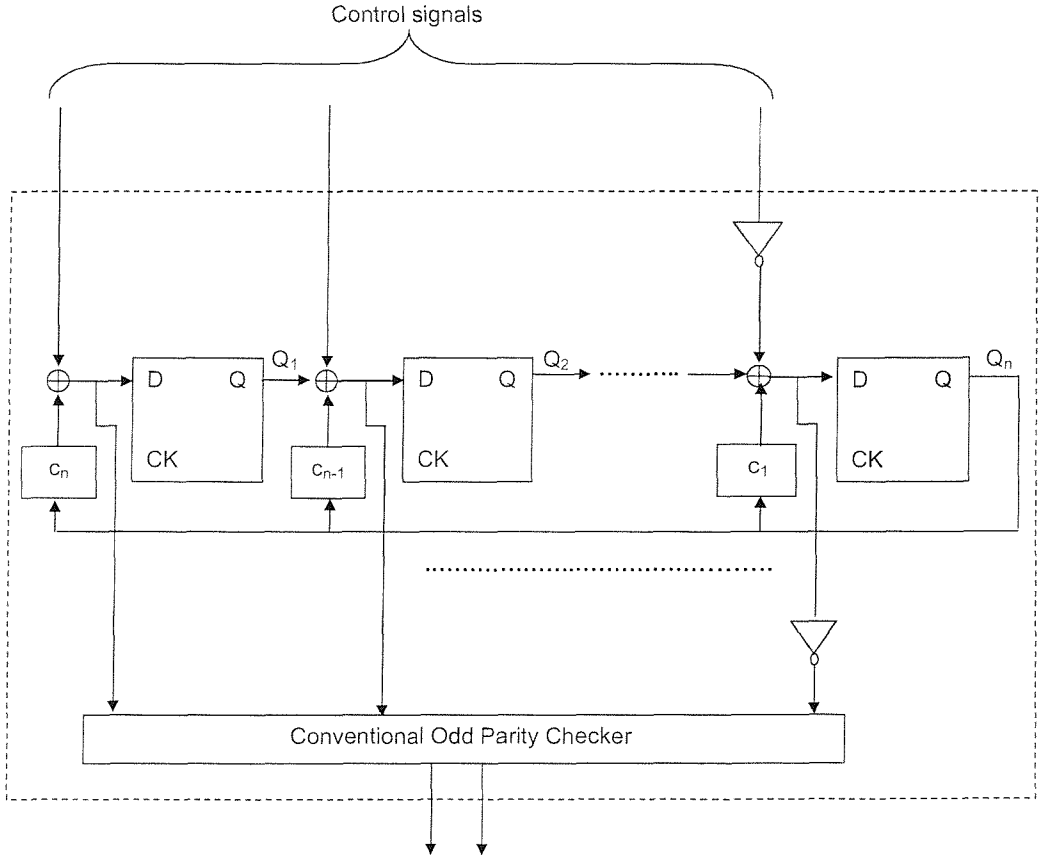


Figure 6.9. TSC parity checker, to be used in CTRL_1, CTRL_2, CTRL_3, CTRL_4

encoded code word that feeds the conventional odd parity checker. The n-bit LFSR itself can be designed by obtaining its characteristic polynomial as outlined in §2.2.1.1, based on Theorem 2.1, and consulting the literature for tables of primitive polynomials (see for example [137, 138]).

As mentioned in §2.2.1.1, the presence of the LFSR has the effect that the conventional parity checker embedded within the overall structure of Figure 6.9 receives all possible code words, as long as two different code words appear in its inputs (which is always true in the case at hand). Again according to Lemma 2.1, this means that any arrangement of 2-input XOR gates in the disjoint parity trees that constitute the conventional checker, will lead to a TSC solution.

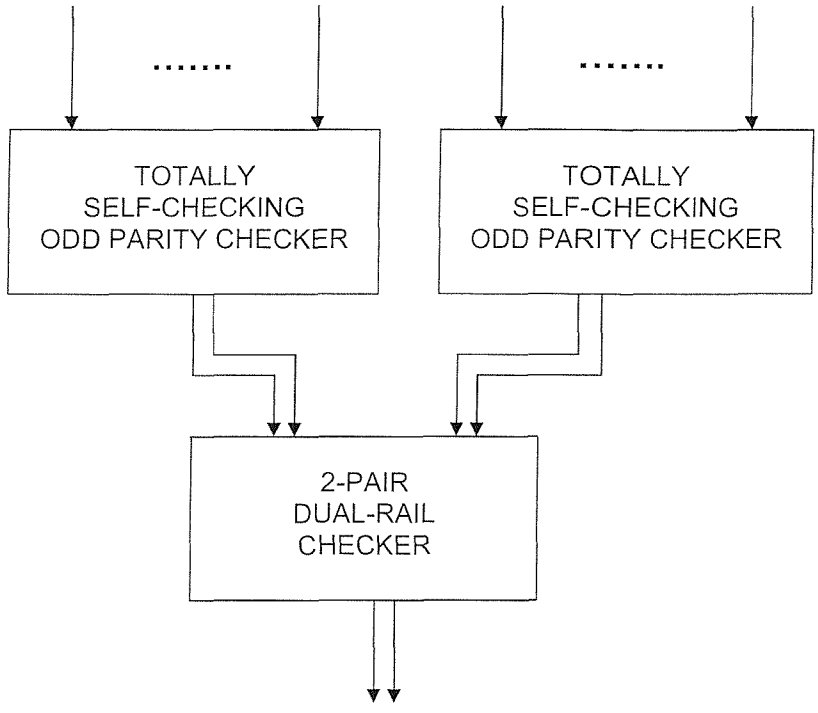


Figure 6.10. Compacting the outputs of two TSC parity checkers

Let us now examine the dual-rail checkers of Figures 6.4 and 6.7 with respect to the totally self-checking goal. First of all, consider the simple example of Figure 6.10 and assume fault-free operation. The figure implies that only two processes exist in the system, and their control signals are checked using two TSC odd parity checkers like the one of Figure 6.9, applying either CTRL_1 or CTRL_3. The outputs are naturally compacted using a two-pair dual-rail checker (in effect a single dual-rail checker cell), as shown. Since the corresponding conventional parity checkers receive all code words, they are also able to produce both possible code outputs (01, 10). The exact state of each control path (i.e. the exact signal fed to the TSC odd parity checker), together with the internal state of the corresponding LFSR determine which of the two possible outputs will be produced. In prin-

ciple, the two processes are independent, while the two LFSRs are always independent. It is therefore sensible to state that, over time, all possible code input combinations {(01, 01), (01, 10), (10, 01), (10, 10)} will appear at the input of the dual-rail checker. Generalizing for an n-process design, with n TSC parity checkers producing both their code outputs, it can be understood that *all* possible 2^n input combinations will be fed to the n-pair dual-rail checker. As explained in §2.2.2.2, only four well-selected code inputs are enough to guarantee the self-testing property for an n-bit dual-rail checker of a given structure. The fact that all possible code words are applied to the checker in the case at hand, suggests that *any* structure (i.e. any internal arrangement of the n-1 dual-rail checker cells) will result in a TSC design.

At this point, the issue regarding the actual checker structures of Figures 6.4, 6.5, 6.7, and 6.8, left open in §6.2.1, has been answered. To summarise, all parity checkers in the figures are implemented using the configuration of Figure 6.9, where the conventional parity checker shown, is composed of two 2-input XOR gate-based parity trees, with arbitrary distribution of gates between the trees, and arbitrary arrangement of the gates within the trees. In addition, the dual-rail checkers of Figures 6.10 are composed of an arbitrary arrangement of 2-pair dual-rail cells.

Clearly, the result of utilising the TSC checker of Figure 6.9 in the controller self-checking schemes, is that the estimations of equations (6.4), (6.5), (6.7) and (6.8) are no longer valid, since they were derived assuming simple parity checkers, and do not take the LFSR hardware overhead into account. They can, however, easily be suitably augmented as follows. Let $Cost_{LFSR\_1}$ be the constant hardware cost of an LFSR cell, that is the cost of a flip-flop, plus the cost of the XOR gate. Clearly, the n-bit LFSR of Figure 6.9 will cost $n \times Cost_{LFSR\_1}$. This is not totally accurate, since the XOR gate can be a 2 or a 3-input one, depending on the absence or presence of a feedback tap, so $Cost_{LFSR\_1}$ should not be a constant. The two inverters are also not taken into account. Let us, however, ignore these negligible details, and accept this approximation for the purposes of this discussion. In each of the presented schemes, the number of LFSR cells is equal to the total number of inputs of all parity checkers. Based on this :

- in the CTRL_1 scheme (Figure 6.4), checker PC_i has m_i+1 inputs, adding up to a total of

$$\sum_{i=1}^{n}(m_i+1)=N_s+n-1 \tag{6.9}$$

inputs in all parity checkers (equation (6.2) has been used in the above). Equation (6.4) can now be updated as

$$\begin{aligned} Cost_{CTRL\_1} &= [N_S-(n+1)]\times Cost_{XOR}+6\times(n-1)\times Cost_{AND/OR}+ \\ &+(N_S+n-1)\times Cost_{LFSR\_1} \end{aligned} \tag{6.4'}$$

- in the CTRL_2 scheme (Figure 6.5), the single odd parity checker receives N_s inputs. Equation (6.5) can now simply be rewritten as

$$Cost_{CTRL\_2}=(N_s-2)\times Cost_{XOR}+N_s\times Cost_{LFSR\_1} \tag{6.5'}$$

- in the CTRL_3 scheme (Figure 6.7), as already seen in §6.2.4, each process P_i with at least one Intrinsically Secure state ($m_{i,S}\neq 0$) feeds an $m_{i,N}+2$ input parity checker, while whenever no IS state can be identified ($m_{i,S}=0$) the checker has $m_{i,N}+1$ inputs. The relationship giving the total number of parity checker inputs is shown in the following to be analogous to equation (6.6)

$$\sum_{i/m_{i,S}\neq 0}(m_{i,N}+2)+\sum_{i/m_{i,S}=0}(m_{i,N}+1)=\sum_{i=1}^{n}(m_{i,N})+n+n_{IS} \tag{6.10}$$

Equation (6.7) now becomes

$$\begin{aligned} Cost_{CTRL\_3} &= \left[\sum_{i=1}^{n}(m_{i,N})-(n-n_{IS})\right]\times Cost_{XOR}+6\times(n-1)\times Cost_{AND/OR} \\ &+\sum_{i=1}^{n}C_{OR}(m_{i,S})+\left[\sum_{i=1}^{n}(m_{i,N})+n+n_{IS}\right]\times Cost_{LFSR\_1} \end{aligned} \tag{6.7'}$$

- finally, in the CTRL_4 scheme (Figure 6.8), the total number of checker inputs (§6.2.5) has been shown to be $\sum_{i=1}^{n}(m_{i,N})+n_{IS}+1$, which updates equation (6.8) to

$$\begin{aligned} Cost_{CTRL\_4} &= \left[\sum_{i=1}^{n}(m_{i,N})+n_{IS}-1\right]\times Cost_{XOR}+\sum_{i=1}^{n}C_{OR}(m_{i,S})+ \\ &+\left[\sum_{i=1}^{n}(m_{i,N})+n_{IS}+1\right]\times Cost_{LFSR\_1} \end{aligned} \tag{6.8'}$$

An inspection of the updated equations reveals that the relationships between the hardware costs of the different schemes still hold; indeed, comparing, for example, equations (6.4')

and (6.5′), is enough for one to realise that the more the degree of parallelism n, the more the hardware savings achieved, not only due to the absence of AND / OR gates, but also due to the reduced number of LFSR cells. This is totally consistent with the observations already made in §6.2.6. It can therefore be claimed that, although the numeric results of Table 6.1 are now even farther from being accurate, the qualitative insight they provide is still relevant.

On the other hand, the increase in area imposed by the design of Figure 6.9, can be unacceptable in the case of realistic designs. Recall, for example, chapter 5, where even designs with a critical path of the order of 100 states were shown. This clearly implies that *parity checking in the considered context is not always as cheap as it first appears*. The situation calls for an alternative approach; self-checking using 1/n checkers is therefore considered in the following section §6.3.

6.3 1/n based self-checking

This section investigates the possibility of directly applying 1/n self-checking to the controller outputs. The reader is reminded that an m/n checker (§2.2.1.2) detects all single, as well as multiple *unidirectional* faults in its inputs. The implication of this on the controller structure (Figure 6.1), is that the next state logic and decoder blocks have to be designed such that no single internal controller fault can under any circumstances give rise to a bidirectional multiple fault on the controller output. This problem has been addressed in [139], based on the following definition.

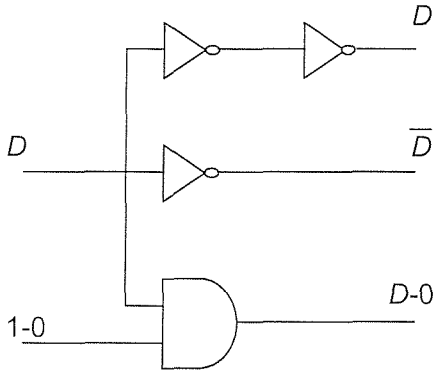


Figure 6.11. An example of fanout branches with different inversion parities

Definition 6.2 : The *inversion parity* of a logic path is the number of inversions in the path modulo 2.

Given a block of logic, a fault on an internal wire will only lead to unidirectional faults on the output of the block, if all paths on the fanout of the wire have the same inversion parity. The trivial but illustrative circuit of Figure 6.11 clarifies this proposition. The figure

shows a stuck-at-0 wire, assuming the D value when its source tries to drive it to 1. The wire has a fanout of 3 branches. The top and the bottom branches have a 0 inversion parity (2 and 0 inversions respectively). If faults reach the circuit outputs through both of these branches, then they result in the same logic error. Indeed, in the figure, if the second input to the AND gate is logic 1, then fault propagation through both paths leads to D-type faults. Of course, a fault does not necessarily make its way to the circuit output, an example being the scenario where the second AND gate input is a logic 0, leading to a fault-free 0 value in the output. In contrast, the middle branch has an inversion parity of 1 (a single inversion). Clearly, this produces a $\overline{D}$ on the respective circuit output; in combination with the top and bottom branches, the D-type faulty input wire causes bidirectional faults on the output lines.

Returning to the controller self-checking problem, it is such situations that need to be excluded when designing the controller blocks, in order for m/n checking to be safely applicable. It is therefore to be noted, that, in contrast to the parity checking situation of §6.2, hardware sharing between the logic cones of controller outputs *is* permitted, so long as it does not lead to fanout branches with different inversion parities. Clearly, this is a less restrictive constraint than that of §6.2; it can, therefore, lead to more compact control path realisations. In the following, it will be assumed that this constraint is satisfied, and under this sole assumption the presented techniques are generically applicable.

6.3.1 Selection of a 1-hot checker

Several 1-out-of-n checker designs have been proposed (§2.2.1.2). Unlike the parity checking case, where XOR trees dominate the field, there seems to be no clear winner as far as 1/n checking is concerned. This subsection states the desired properties of the 1/n checker to be used, then revisits the techniques presented in §2.2.1.2, and finally justifies a particular choice.

6.3.1.1 Checker specifications

The checker needed for the current problem should have the following characteristics :
(a) It must be *hardware-efficient.* This comes directly out of the problem statement of §6.1.4; as already mentioned therein, the assumption is that the designer has already paid a

significant penalty for datapath self-checking (chapter 5). It is therefore desirable to keep any controller-related extra overhead as low as possible.
(b) It must be *generic*, applicable to as wide a range of the bit width n as possible. This stems from the whole context of this work : any solution should be able to easily lend itself to high-level synthesis, where the length of the process critical paths cannot possibly be known a priori. Applicability implies that the checker should be not only constructable, but also consistent with theory (totally self-checking) for as many values of n as possible (§6.1.4).
(c) It must be *technology-independent*. Incorporating technology-independent solutions in high-level synthesis is a virtue, since it takes full advantage of the largely technology-independent nature of the synthesis process, and maintains its ability to be easily tuned to alternative technologies.
(d) It must be relatively *simple* in its description, so that it can easily be coded in an HDL and incorporated in an Electronic Design Automation (EDA) flow.
(e) In contrast, speed is *not* a critical factor. To understand this, once again consider Figure 6.1. The minimum clock period achievable by the synchronous design is determined by the datapath, while controller self-checking is done in parallel to the datapath operation. As chapters 3 and 5 have established, it is often the case that several data path operators or multiplexers operate in series within the same control step. This guarantees ample time for the (normally faster) single operation of 1/n checking to be completed.

The above characteristics add up to a simple sentence : a checker that demonstrates requirements (a)-(d) can be allowed to perform suboptimally as regards speed.

6.3.1.2 1/n checkers revisited

A critical summary of §2.2.1.2 is provided here. The reader is reminded of all proposed solutions for the 1-hot self-checking problem, and these solutions are evaluated in the light of points (a)-(e) of §6.3.1.1.

Recall Anderson and Metze's m/n checker (Figure 2.13 and [10]). A 1-hot checker is effectively implemented as part of it, using a simple code translator, followed by a k/2k checker implemented using majority functions. The design is inherently based on logic gates, so it is technology-independent, while the majority functions could be described in

an HDL and therefore their design automated. However, the TSC property cannot be satisfied for all values of n (n=7 is a characteristic problematic situation mentioned in [10]); the design is therefore unsuitable for the situation at hand.

The adder-based k/2k checkers proposed by Paschalis et al (Figure 2.14 and [24]) can be used in Anderson and Metze's scheme instead of the majority function based one. One advantage is that the design becomes cheaper compared to [10] as k increases; it is also independent of technology, since it uses full and half adders as building blocks. On the other hand, it is rather complicated to describe it in a generic HDL form, since its TSC property strongly relies on the arrangement of adders within the blocks of Figure 2.14. Therefore, a behavioural description is not possible, and a structural one is rather hard to parameterise (so as to make it generically utilisable at a high level of abstraction). The TSC property is achieved for the k/2k checker, *if* it receives all its code words; however, the translator of Figure 2.13 is known to not always provide all code words. Consequently, it is not guaranteed that such a combination provides a TSC solution for every bit width of interest.

The checker of Tao et al [30] is also based on a configuration similar to Figure 2.13. The structure is based on elementary logic functions, and its design is described algorithmically; it could therefore fit within an HDL-based design flow, had it been more generically applicable. Indeed, the TSC property is not achieved for some common values of n (such as 7, 9, 11).

CMOS technology specific designs [28, 33] are cheap and generic. They are, however, unsuitable for the problem at hand, clearly due to their total dependence on target technology, and their irrelevance to high-level HDL-based design flow.

Khakbaz's 1-hot checker ([29] and Figures 2.16, 2.17, 2.18) is an interesting option. Its hardware cost is reported to be comparable with [10] and [24], it can be applied for every bit width, except the well-known problematic 1-out-of-3 case, it is technology-independent, and it can be described in a behavioural HDL through equations (2.4), and using a common dual-rail checker description (§5.3.3.3). In the literature it is criticised as being slow [30], but as argued in §6.3.1.1 this may not necessarily harm. Clearly, it is a tempting choice.

Finally, the accumulator-based sequential structure of Stroele and Tarnick (Figure 2.20 and [34]), although easily implementable, particularly suitable for high-level description, and totally technology independent, does not conform to the usual self-checking theory, in that it experiences fault latency of unpredictable length, and it can even experience fault escapes.

The above discussion singles out Khakbaz's 1/n checker design as the most suitable candidate. The following subsections §6.3.2, §6.3.3 describe the controller self-checking solutions instrumented using it.

6.3.2 Per process 1/n-based self-checking

Based on the selected 1/n checker structure, Figure 6.12 shows how an overall self-checking solution for the controller of a generic highly parallel conceptual design can be configured. The technique is directly analogous to the CTRL_1 method, as simple comparison of Figures 6.4 and 6.12 suggests. This time, however, the parity checkers have been substituted by 1/n checkers, implemented as in Figure 2.16. Responses from all checkers corresponding to all processes are naturally compacted by the usual dual rail checker. This scheme is hereafter referred to as the *CTRL_5* self-checking scheme.

Figure 6.12 also shows how the problematic 1-out-of-3 checker case is dealt with. Let us concentrate on process P_n. Without loss of generality, it is assumed to comprise 2 states. Together with state-0, this dictates the need for a 1-out-of-3 checker. As a first word of note, it has to be stated that such short processes are rather trivial, and not frequently encountered in controller / datapath architectures. The only realistically meaningful service that a 2-state process normally has to offer, is the updating of outputs or internal signals, concurrently with other, useful operations performed by the rest of the processes in the system. Typically, this involves brief periods of activity, and extended periods during which the short process simply waits. Further, the overall system critical path, being the critical path of the longest process, is highly unlikely to be any relevant to the 2-state process length. It would therefore do no harm to add a "dummy" state to the short process (as shown in process P_n in the figure), thus eliminating the need for a 1-out-of-3 checker, and performing 1-out-of-4 checking instead. Essentially, having control of the synthesis task, the designer can avoid the problematic 1/3 situation.

One is tempted to think that the same principle of inserting dummy states to avoid problematic codes could be used more extensively, and an alternative checker adopted instead of Khakbaz's one. For example, Tao's checker (§6.3.1.2) could be used, and dummy states inserted whenever 1/7, 1/9 or 1/11 codes were encountered. This is, however, not so, since processes that are 6, 8, or 10 states long typically perform useful tasks and often determine the critical path; therefore, lengthening them is very likely to hinder performance and partly cancel out the benefit of the HLS critical path length optimisation effort.

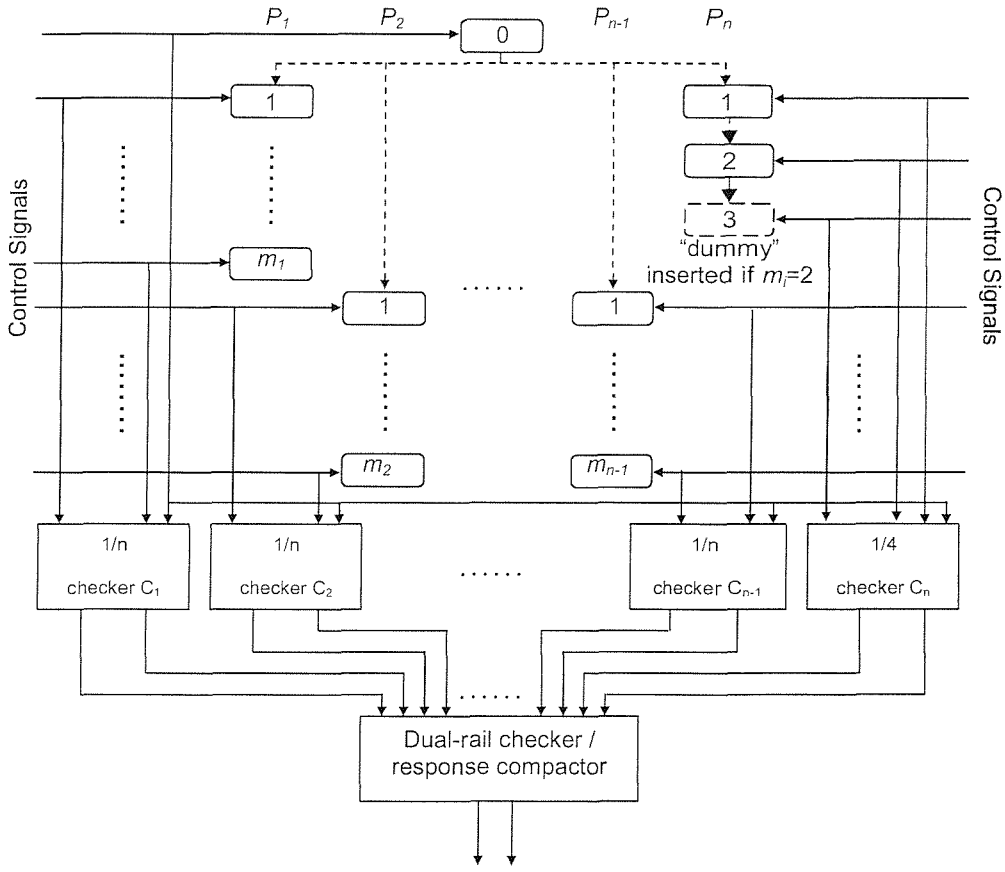


Figure 6.12 : The CTRL_5 self-checking scheme

It also has to be noted that the combined 1/3 self-checking approach of [31] (Figure 2.19) would also provide an acceptable TSC solution, since one expects at least one more process (therefore at least one more checker output) in the system control path, and definitely a number of 2-bit comparator outputs coming from the datapath (chapter 5). Inserting a

dummy state as above was, however, preferred, for being more standalone and independent of the system context, also simpler in concept and easier to incorporate in synthesis.

The CTRL_5 scheme is TSC overall. Indeed, the 1-out-of-n checkers receive all their code inputs and are all TSC (as proved in [29]), especially so now that the 1-out-of-3 issue has been resolved. The dual-rail checker is code-disjoint and fault-secure by construction (§2.2.2.2), and it also receives all its code inputs, since the arguments built around Figure 6.10 are equally applicable here. It is therefore self-testing for any internal arrangement of dual-rail checker cells. This makes both the dual-rail checker and the overall scheme totally self-checking. Notably, this does *not* require any costly LFSR-based design, in contrast to the parity-based techniques.

Finally, no hardware cost estimation prediction is given here. Firstly, the nature of the code translator that forms part of the 1/n checker (Figure 2.17) allows for hardware sharing and optimisation, without affecting the TSC property (notably, in contrast to parity checkers). This will be made clear in the implementation part of this chapter (§6.4.2). This optimisation often depends on the synthesis tool in use. Therefore, any prediction on a purely theoretical basis would likely be misleading. Secondly, such a prediction would only serve the purpose of comparison between 1/n based schemes and their parity based counterparts, e.g. in this case CTRL_5 and CTRL_1, through equation (6.4'). This last equation is, however, highly dependent on target technology (because of $Cost_{LFSR\_1}$ being dependent on technology). It is, therefore, best to leave any such comparison for the experimental section §6.4.

6.3.3 Per process 1/n-based self-checking exploiting Intrinsically Secure states

Figure 6.13 is clearly analogous to Figure 6.6, and shows how any existing Intrinsically Secure states can be exploited within a single process, when 1/n checking is applied. As the proof of Theorem 6.1 has shown, any single fault in the control signals leads to either the all-0s pattern, or a 2-out-of-n word, or even a 1-out-of-n word, plus an alarm from the datapath. In any of these cases, the 1/n checker serves just as well as the parity checker. Moreover, given that 1/n checkers can detect not only single, but also multiple unidirectional faults, it would be interesting to consider such faults here as well.

Theorem 6.2 : The configuration of Figure 6.13 detects all unidirectional control signal faults, while providing the fault-free indication under fault-free operation.

Proof:

The proof proceeds on the footsteps of the proof of Theorem 6.1 :

a) Consider the case when one of the IS state control signals is active :

a1) Under fault-free operation, since one of the IS state control signals is active (logic 1), the OR output is a logic 1; since the controller is one-hot, all control signals corresponding to non-IS states are 0. Therefore, the 1/n checker is fed by a 1/n pattern, thus signalling correct operation.

a2) Let us consider k≥1 $\overline{D}$ type faults. If all of them appear on IS-state control signals, then the datapath produces k error indications. If all of them appear on non-IS state control signals, then the 1-out-of-n checker is fed by a (k+1)-out-of-n word, and signals a fault. If some of the faults are in IS and some in non-IS state control signals, then both the 1/n checker, and an appropriate number of datapath checkers produce error indications.

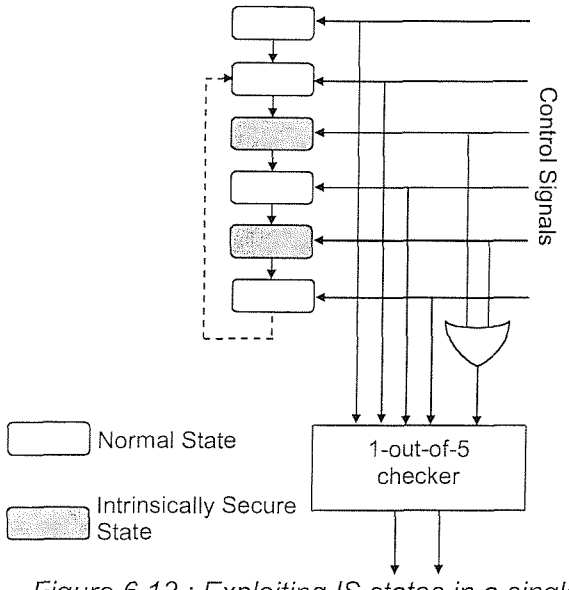


Figure 6.13 : Exploiting IS states in a single process with 1/n controller self-checking

a3) There can only be a single *D* type fault, since there is only one active signal in the design (the one corresponding to an IS state). If this faulty situation occurs, then the 1/n checker is fed by the all-0s pattern and thus detects the fault.

b) Now consider the case when one of the non-IS signals is active.

b1) Under fault-free operation, the OR gate outputs logic 0, since all IS control signals are inactive. Only one of the 1/n checker inputs is 1, so fault-free operation is confirmed.

b2) Consider k≥1 $\bar{D}$ type faults. The situation is evaluated exactly as in a2 above and no further explanation is required.

b3) Once again, there can be no multiple D type fault, and under the presence of a single one, the checker is fed by all-0s and naturally detects the fault. ▲

Interestingly, consider a double bidirectional fault, that is, a fault affecting two controller outputs, such that one assumes the D value, and the other assumes $\bar{D}$. Such a fault would escape detection in the environment of Figure 6.12; *however*, in Figure 6.13, if the signal taking the $\bar{D}$ (0/1) value happens to correspond to an IS state, then the data path signals a

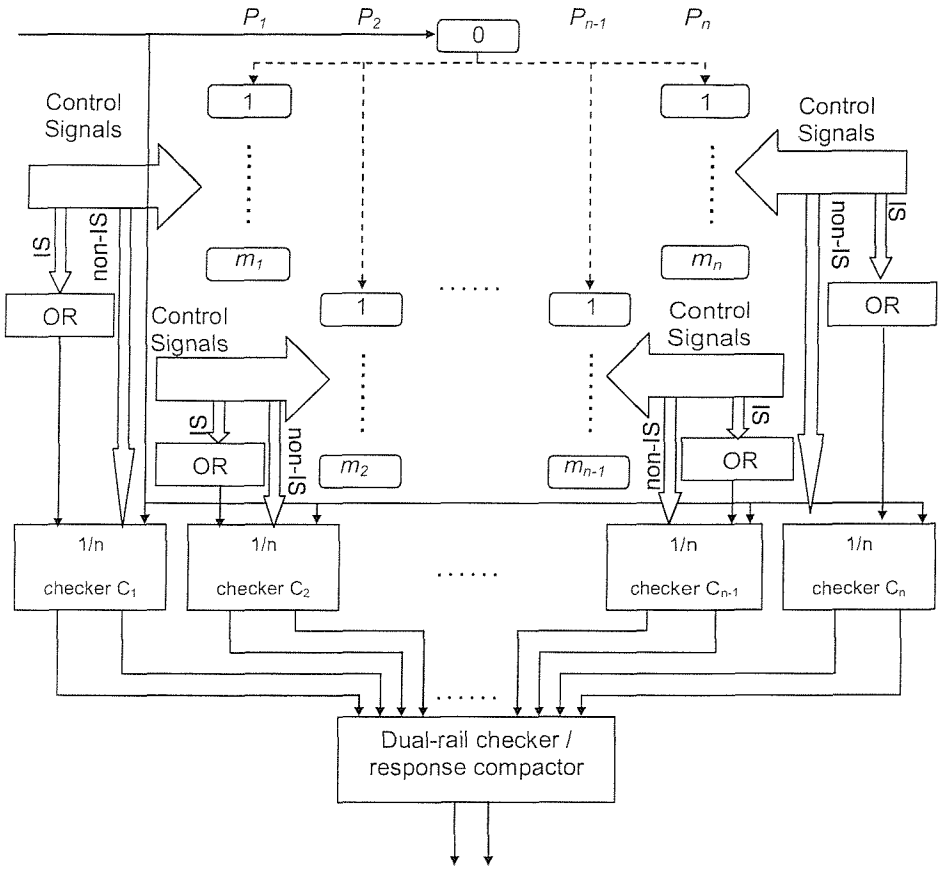


Figure 6.14 : The CTRL_6 self-checking scheme

fault and the fault is detected. This shows that, once more, when utilising Intrinsically Secure states within a process, in the manner of Figure 6.13, the overall self-checking scheme demonstrates enhanced fault detection capabilities, in that now bidirectional faults can also be detected, provided that they corrupt any IS state towards a $\bar{D}$ value. The analogy with the corresponding parity-based scheme is evident.

An overall scheme for the self-checking design of the control path utilising 1/n checkers, and taking Intrinsically Secure states into account, is now proposed in Figure 6.14. It is clearly the 1/n "equivalent" to CTRL_3, and will be called *CTRL_6*. In line with §6.3.2, if any of the checkers C_i is originally fed by 3 inputs, an additional dummy state is provided to the corresponding control path, to resolve the problematic 1-out-of-3 situation (although no such situation is depicted in the figure). Moreover, no theoretical cost estimation is attempted here either, although CTRL_6 can be expected to be somewhat cheaper than CTRL_5, due to utilising cheaper checkers.

Finally, note that no technique analogous to CTRL_2 or CTRL_4 is proposed, i.e. there is no attempt to utilise a single m/n checker. The reason for this, is that there is no generic TSC m/n checker, for arbitrary n and m>1. Most designs mentioned §2.2.1.2 are rather restricted to the area around the k/2k checker, which is not useful for the purposes of this research.

6.4 Implementation and Experimental Results

The discussion in this section focuses on the MOODS High-Level Synthesis Suite (§3.2), and precisely on the implementation of the ideas of §6.2, §6.3 within MOODS. Some essential properties of the MOODS controller are first established (§6.4.1), then implementation details are given (§6.4.2, §6.4.3), and finally the obtained experimental results are presented, together with relevant comments (§6.4.4).

6.4.1 MOODS-generated controller revisited

Sections 6.2 and 6.3 established that controllers need to satisfy certain properties, in order for the respective techniques to be applicable. As a reminder, in order to apply parity-based schemes, one needs to design the controller such that any single internal fault can affect an odd number of output signal bits, while if 1/n-based techniques are desired, the designer needs to make sure that any internal fault may only lead to unidirectional faults on the output. Favourable exceptions exist (§6.2.3, §6.3.3), but the above statements are in principle correct. Observe that, if a controller has the property that every modelled internal fault may lead to one and only one corrupted output bit, then both of the above require-

ments are satisfied. In the following, it is shown that the control path generated by MOODS indeed possesses this characteristic by construction, and therefore all six considered techniques can safely be utilised in its environment.

First of all, recall the generic controller model given in Figure 6.1, consisting of a state register and two combinational logic blocks, namely the next state logic and decoder blocks. Compare this model against the MOODS-specific control path implementation of Figures 3.12 and 3.13. The comparison reveals that in the MOODS implementation no decoder is present. This is expected, since as mentioned in §3.2.6, there exists exactly one general control cell (one flip-flop) for every control state in the system. The D-flip flops found within the general control cells effectively constitute the state register, and any sin-

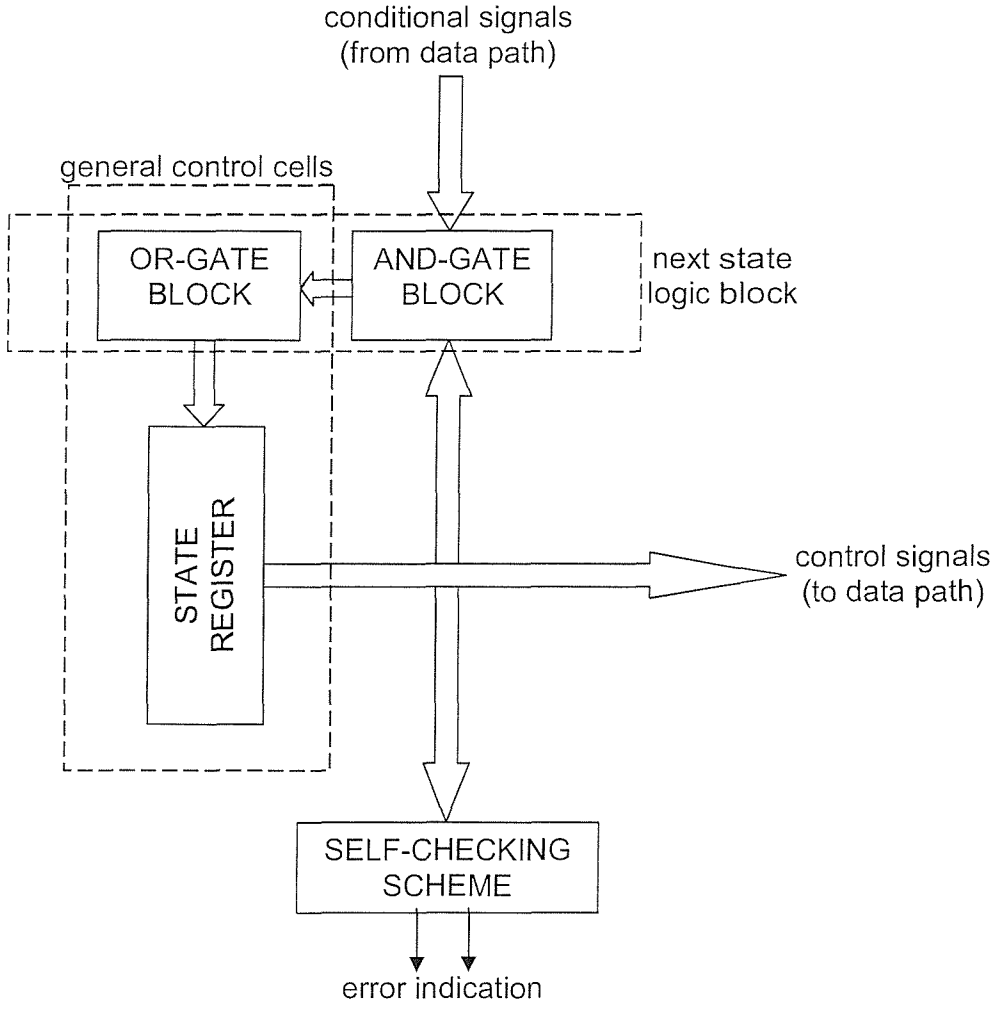
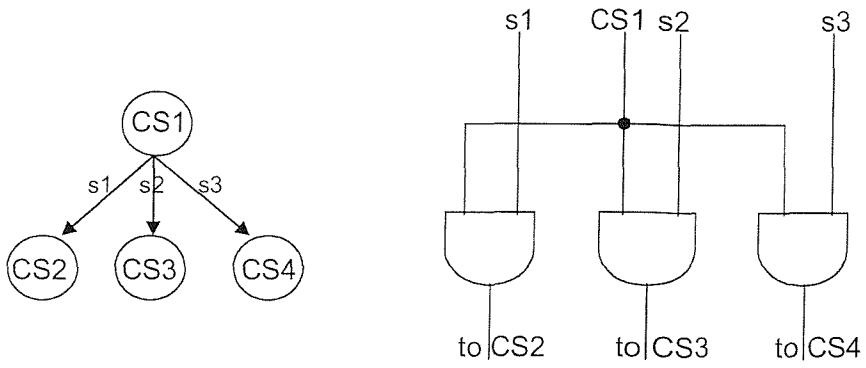


Figure 6.15. The MOODS controller supplemented with self-checking capabilities



(a) control path *(b) hardware implementation*

Figure 6.16. Conditional control flow

gle fault in any of them (in any of their ports : D input, Q output, set, reset, clock) may only affect a single controller output.

Let the discussion now concentrate on the "next state" logic block. Once more, comparison of Figures 6.1, 3.12 and 3.13 reveals that the next state logic in the MOODS implementation effectively comprises the AND gate block (Figure 3.13) and the OR gates found within the general control cells (Figure 3.12). For convenience, this idea is depicted in Figure 6.15. The figure is just an alternative view of Figure 3.13, except for the self-checking scheme block included here. Other than that, the general control cells of §3.2.6 have been decomposed into the flip flops constituting the state register, and a separate OR-gate block. The above mentioned next state block model is also shown. As Figure 3.12 established, OR gates are dedicated to flip-flops; this means that the output path from any gate in the block only leads to one flip-flop. In other words, any fault in any gate of the OR-gate block can only affect one flip-flop, therefore it can only result in a *single* corrupted bit in the control signals.

The AND-gate block requires some more attention. The particular block implements conditional control flow, originating in conditional and loop statements in the system VHDL description. An illustrative example of conditional control flow is provided in Figure 6.16a, together with its hardware implementation in 6.16b. In the example, control step CS1 is followed by CS2, CS3 or CS4, depending on the values of conditional signals s1, s2 and s3 (notably, exactly one of them is "true" when CS1 is active). This is implemented in the AND-gate logic block in the simple manner that Figure 6.16b depicts. The outputs of the AND gates are directed to the general control cells / flip flops that correspond to the

succeeding CS2, CS3 and CS4 (at times through suitable OR gates in the respective block). Since there is exactly one flip flop in the state register for every control step, the logic path from the output of each AND gate leads to exactly one state flip flop. Indeed, any variation from this would lead to functionally meaningless situations, for example VHDL `case` statements where two different branches are activated simultaneously. In turns, this means that any single fault in the AND-gate block can propagate to a single flip flop, and thereby affect a *single* control signal.

Thus the discussion of this subsection concludes. The last few paragraphs established that any single fault in any part of the MOODS controller (state register, OR-gate block, AND-gate block) may corrupt a single output bit. In fact, this is an inherent property of direct one-hot encoding of the control signals. All six controller self-checking schemes of §6.2 and §6.3 can thus safely be applied.

6.4.2 Self-checking design cell libraries

Subsection 6.4.1 established that controller self-checking as addressed in this chapter is perfectly applicable to the control path model of designs synthesized by MOODS. As is obvious from the discussion so far, in principle controller self-checking has no direct relevance to the synthesis tasks of chapter 3, as the checking hardware is always just an add-on to the normal design (see for example Figure 6.15). Therefore, all that is needed for the implementation of the considered techniques within HLS is a simple post processing step. Such a post processing step should take into account the self-checking technique that the designer chooses for a particular experiment (CTRL_1, CTRL_2, CTRL_3, CTRL_4, CTRL_5 or CTRL_6), identify any Intrinsically Secure states (if applicable) and then add a synthesisable VHDL description of the checking block to the tool output code, taking care of the proper connections of control signals to the inputs of the self-checking logic block. This logic block comprises conventional parity checkers, LFSR structures, dual-rail checkers, and / or 1-out-of-n checkers, as applicable. Such components are not available within the standard MOODS cell library (§3.2.7); a dual-rail cell library was however developed and used for the pur-

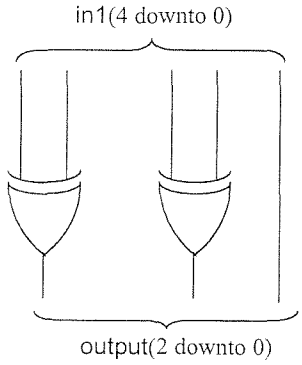


Figure 6.17. A 5-bit XOR array

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity XOR_ARRAY is
    generic (m: positive := 1);
    port (in1 : in std_logic_vector (m-1 downto 0);
          output: out std_logic_vector ((m/2 + (m rem 2))-1 downto 0));
end XOR_ARRAY;

architecture structure of XOR_ARRAY is
begin
    G1: if m>1 generate
      output(m/2+(m rem 2)-1 downto 0) <= in1(m-1 downto m/2+(m rem 2)) xor
in1(m/2+(m rem 2)-1 downto (m rem 2));
end generate;
    G2: if (m rem 2)=1 generate
      output(0) <= in1(0);
end generate;
end;
```

Figure 6.18 : The `XOR_ARRAY` *cell*

poses of chapter 5 as explained in §5.3.3.3. As a reminder, a relatively simple C++ programme was written that automatically created a VHDL `package`, comprising synthesisable descriptions of dual-rail checkers, receiving anything between 1 and 200 pairs of inputs. Parity checkers are known to have a very similar structure, except that instead of dual-rail checker cells they consist of 2-input XOR gates ([2, 5] and §2.2.1.1 of this thesis). As explained in §6.2.7, an n-bit conventional parity checker employed in this work as Figure 6. 9 has shown, can have an arbitrary arrangement of its constituent 2-input XOR gates without loss of the TSC property. A straightforward symmetrical arrangement was therefore chosen, and a C++ programme was used to automatically create the respective parity checker VHDL `package`. Initially an XOR array cell is defined (analogous to the checker array of §5.3.3.3). Figure 6.17 shows an example 5-bit XOR array, while Figure 6.18 shows the VHDL description of the generic m-bit XOR array cell. Parity trees are then composed of XOR arrays. Figure 6.19 gives an example 10-bit parity tree. The figure depicts the block diagram structure of the implementation; of course some XOR arrays are very simple structures, for example a 2-bit "XOR array" only consists of a single XOR gate.

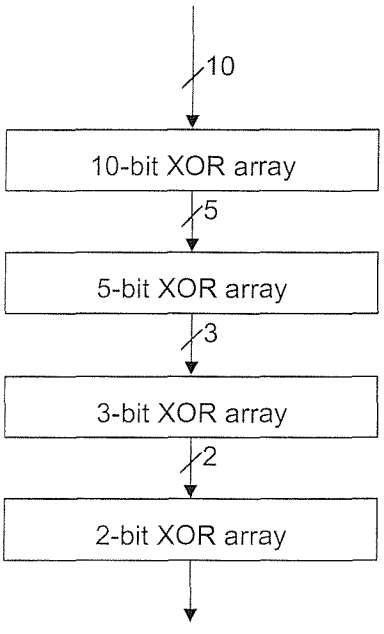


Figure 6.19. Block diagram of a 10-bit parity tree

A generic n-bit parity checker consists of two disjoint parity trees, of widths $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ respectively. Figure 6.20 shows the block diagram of a 21-bit parity checker. The overall parity checker library contains the XOR array of Figure 6.18, structural descriptions of parity trees of bit width values in the range 1 – 100, and based on them, structural descriptions of parity checkers of bit width values in the range 1 – 200. If needed, a trivial modification of the generator C++ programme could produce checkers of even wider inputs.

A library of LFSR cells was implemented similarly. Firstly a 1-bit LFSR cell was defined. The cell is shown in Figure 6.21, while its synthesisable VHDL description is provided in Figure 6.22. The cell is used as a building block for the LFSR structure of Figure 6.9. By connecting a constant 0 to the "feedback" input port, one can model the absence of a feedback tap (indeed, logic synthesis tools typically optimise out the feedback input in such cases). Similarly, by connecting a constant 0 to the `shift_inp` input typically causes this input to be optimised out and creates a 1-bit LFSR cell like the leftmost cell of Figure 6.9. The VHDL `generic rst_val` determines whether the "rst" input will be connected to the "set" or to the "reset" (as in Figure 6.21) port of the D-flip flop. This pro-

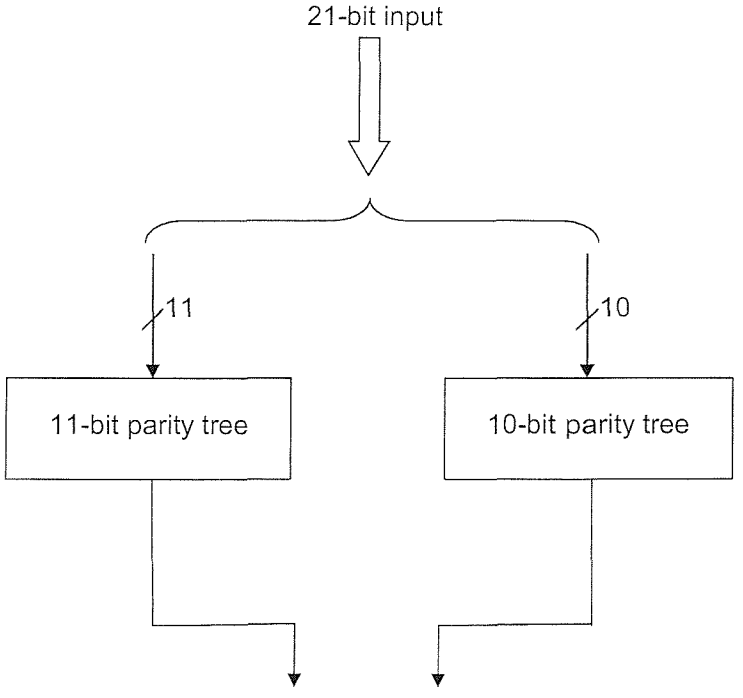


Figure 6.20. Block diagram of a 21-bit parity checker

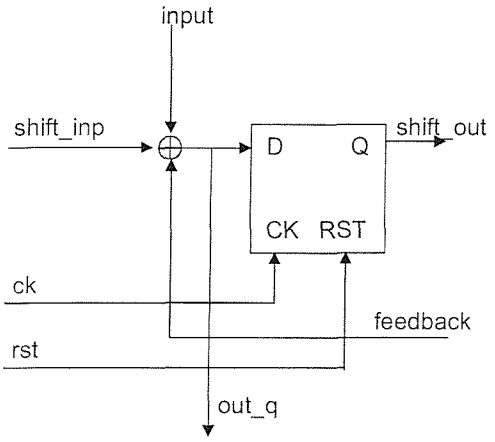


Figure 6.21. The 1-bit LFSR cell

vides a convenient mechanism for the initialisation of the LFSR with the desired value. The positions where feedback taps are to be added in an n-bit LFSR are determined as §2.2.1.1 explained, based on Theorem 2.1 and Figure 2.12. In particular, for a given n, an (n-1)-degree primitive polynomial is chosen, from the tables of primitive trinomials and pentanomials of degrees between 2 – 100 provided in [137]. The chosen polynomial is multiplied by the generator polynomial of the even parity code g(x)=x+1, thus creating the n-bit characteristic polynomial of the LFSR to be designed. The reader is reminded that such a procedure guarantees that the resulting LFSR will produce all non-zero even parity code words, if initialised with a non-zero even parity encoded pattern (Theorem 2.1). A C++ programme is thus written, that "knows" the characteristic polynomial corresponding to every n. It automatically generates an output VHDL `package` that consist of the `LFSR_1_bit` cell, followed by synthesisable descriptions of suitable LFSRs of bit-widths between 2 and 100. When outputting the description of each LFSR, the generator programme provides constant 0s or suitable signals to inputs as appropriate to model feedback taps, proper `rst_val` values to initialise to an even parity non-zero word, and also cares for the inverted input and output values needed to accommodate the odd parity considered in this thesis, according to Figure 6.9. As an example, Figure 6.23 shows the VHDL description of the 4-bit LFSR. The 3-bit primitive trinomial chosen from [137] was $d(x)=x^3+x+1$, thus determining

```
library ieee;
use ieee.std_logic_1164.all;
entity LFSR_1_bit is
     generic (rst_val : integer);
     port (input : in std_logic;
                feedback : in std_logic;
           shift_inp : in std_logic;
           ck : in std_logic;
           rst : in std_logic;
           shift_out : out std_logic;
           out_q : out std_logic);
end LFSR_1_bit;

architecture structure of LFSR_1_bit is
  signal internal_sig : std_logic;
begin
  internal_sig <= input xor shift_inp xor feedback;
  out_q <= internal_sig;
  process (ck, rst)
    begin
      if rst = '1' then
        if rst_val = 0 then
          shift_out <= '0';
        else
          shift_out <= '1';
        end if;
      elsif rising_edge(ck) then
        shift_out <= internal_sig;
      end if;
    end process;
end;
```

Figure 6.22 : The `LFSR_1_bit` cell

```
library ieee;
use ieee.std_logic_1164.all;
entity LFSR_n4 is
    port (input : in std_logic_vector(3 downto 0);
          ck : in std_logic;
          rst : in std_logic;
          output : out std_logic_vector(3 downto 0));
end LFSR_n4;
architecture structure of LFSR_n4 is
    signal shift_out_v : std_logic_vector(3 downto 0);
    signal shift_inp_v : std_logic_vector(3 downto 0);
    signal feedback_v : std_logic_vector(3 downto 0);
    signal neg_in : std_logic;
    signal neg_out : std_logic;
component LFSR_1_bit
     generic (rst_val : integer);
     port (input : in std_logic;
               feedback : in std_logic;
           shift_inp : in std_logic;
           ck : in std_logic;
           rst : in std_logic;
           shift_out : out std_logic;
           out_q : out std_logic);
end component;
    for all: LFSR_1_bit use entity work.LFSR_1_bit(structure);

begin

  feedback_v(0) <= shift_out_v(3);
  shift_inp_v(0) <= '0';
  neg_in <= not input(0);
  L0: LFSR_1_bit generic map (1) port map (neg_in, feedback_v(0),
shift_inp_v(0), ck, rst, shift_out_v(0), neg_out);
  output(0) <= not neg_out;

  feedback_v(1) <= '0';
  shift_inp_v(1) <= shift_out_v(0);
  L1: LFSR_1_bit generic map (1) port map (input(1), feedback_v(1),
shift_inp_v(1), ck, rst, shift_out_v(1), output(1));

  feedback_v(2) <= shift_out_v(3);
  shift_inp_v(2) <= shift_out_v(1);
  L2: LFSR_1_bit generic map (0) port map (input(2), feedback_v(2),
shift_inp_v(2), ck, rst, shift_out_v(2), output(2));

  feedback_v(3) <= shift_out_v(3);
  shift_inp_v(3) <= shift_out_v(2);
  L3: LFSR_1_bit generic map (0) port map (input(3), feedback_v(3),
shift_inp_v(3), ck, rst, shift_out_v(3), output(3));
end;
```

Figure 6.23 : A 4-bit LFSR

the LFSR characteristic polynomial $p(x)=x^4+x^3+x^2+1$. On the figure it can be confirmed that the description provided indeed implements p(x), that the LFSR is initialised to the "0011" value, and that the 0-bit position input and output are inverted.

The last building block needed for the implementation of the controller self-checking schemes of this chapter, is the 1/n checker of [29]. As Figure 2.16 revealed, this is composed of a (1/n)-to-(dual-rail) code translator, followed by a dual rail checker, implemented either symmetrically if n is a power of 2, or using three dual-rail checkers in the configuration of Figure 2.18 otherwise. Clearly, the dual-rail checkers can perfectly well be the ones implemented in §5.3.3.3. The task at this point is, therefore, to implement the

code translator. For this purpose, it is enough to express equations (2.4) using the VHDL syntax. The equations are repeated in the following.

$$J_j = \overline{\sum_i x_i}\text{, for all i} : I(p-j)=1 \tag{6.11a}$$

$$K_j = \overline{\sum_i x_i}\text{, for all i} : I(p-j)=0 \tag{6.11b}$$

The summation symbol in this context represents a logic OR; thus the inverse summations of equations (6.11) are in fact NOR functions. The reader is reminded that x_i ($1 \le i \le n$) is the ith-position bit of the checker input, $I(k)$ is the kth-position bit of the binary representation of integer i, integer p is calculated as $p = \lceil \log_2 n \rceil$, and the above two equations are defined $\forall j : 1 \le j \le p$, giving a total of $2 \times p$ equations. Every (J_j, K_j) pair is then complementary, thus constituting a p-pair dual-rail encoded word. The translator equations are analytically well-defined and depend solely on the value of n; therefore, a relatively simple C++ programme was written to automate the production of yet another VHDL `package`, comprising descriptions for the translator equations and corresponding translator-based 1/n checkers, with bit-widths between 2 and 100, *excluding* the problematic n=3 case. Once more, an extension of the library to values over 100 is perfectly feasible through simple modifications of the generator C++ programme. As an example, the VHDL description of the 1/8 to 3-pair dual-rail translator (Figure 2.17) and the resulting checker is shown in Figure 6.24. Notably, it is a particularly compact description. The equivalence of the translator assignment statements to equations (6.11) can easily be verified. Also observe the utilisation of the 3-pair dual-rail checker (`component NEQ_3_n3`, produced as in §5.3.3.3). Figure 6.25 depicts a somewhat harder situation, where the 1/7 checker is im-

```
library ieee;
use ieee.std_logic_1164.all;
use work.neq_3_cells.all;

entity ONE_HOT_CHK_n8 is
    port (in1 : in std_logic_vector(7 downto 0);
          output : out std_logic_vector(1 downto 0));
end ONE_HOT_CHK_n8;

architecture structure of ONE_HOT_CHK_n8 is
    signal J,K : std_logic_vector(2 downto 0);
    component NEQ_3_n3
            port (in1 : in std_logic_vector (2 downto 0);
                  in2 : in std_logic_vector (2 downto 0);
           output : out std_logic_vector (1 downto 0));
    end component;

    for all: NEQ_3_n3 use entity work.NEQ_3_n3(structure);

begin
    J(2) <= not (in1(0) or in1(2) or in1(4) or in1(6));
    K(2) <= not (in1(1) or in1(3) or in1(5) or in1(7));
    J(1) <= not (in1(1) or in1(2) or in1(5) or in1(6));
    K(1) <= not (in1(0) or in1(3) or in1(4) or in1(7));
    J(0) <= not (in1(3) or in1(4) or in1(5) or in1(6));
    K(0) <= not (in1(0) or in1(1) or in1(2) or in1(7));
    N1: NEQ_3_n3 port map (J, K, output);
end;
```

Figure 6.24 : The 1/8 TSC checker

```
library ieee;
use ieee.std_logic_1164.all;
use work.neq_3_cells.all;

entity ONE_HOT_CHK_n7 is
    port (in1 : in std_logic_vector(6 downto 0);
          output : out std_logic_vector(1 downto 0));
end ONE_HOT_CHK_n7;

architecture structure of ONE_HOT_CHK_n7 is
    signal J,K : std_logic_vector(2 downto 0);
    signal intermediate_signals : std_logic_vector(3 downto 0);
    signal out1,out2 : std_logic_vector(1 downto 0);
    signal n1in1,n1in2 : std_logic_vector(1 downto 0);
    component NEQ_3_n2
             port (in1 : in std_logic_vector (1 downto 0);
                   in2 : in std_logic_vector (1 downto 0);
            output : out std_logic_vector (1 downto 0));
    end component;

    for all: NEQ_3_n2 use entity work.NEQ_3_n2(structure);

    component NEQ_3_n1
             port (in1 : in std_logic_vector (0 downto 0);
                   in2 : in std_logic_vector (0 downto 0);
            output : out std_logic_vector (1 downto 0));
    end component;

    for all: NEQ_3_n1 use entity work.NEQ_3_n1(structure);

begin
    J(2) <= not (in1(0) or in1(2) or in1(4) or in1(6));
    K(2) <= not (in1(1) or in1(3) or in1(5));
    J(1) <= not (in1(1) or in1(2) or in1(5) or in1(6));
    K(1) <= not (in1(0) or in1(3) or in1(4));
    J(0) <= not (in1(3) or in1(4) or in1(5) or in1(6));
    K(0) <= not (in1(0) or in1(1) or in1(2));

    n1in1 <= J(2) & J(0);
    n1in2 <= K(2) & K(0);
    N1: NEQ_3_n2 port map (n1in1, n1in2, out1);

    N2: NEQ_3_n1 port map (J(1 downto 1), K(1 downto 1), out2);

    intermediate_signals <= out1(1) & out2(1) & out1(0) & out2(0);
    N3: NEQ_3_n2 port map (intermediate_signals(3 downto 2),
intermediate_signals(1 downto 0), output);
end;
```

Figure 6.25 : The 1/7 TSC checker

plemented. Since 7 is not a power of 2, three instances of dual-rail checkers are used, configured as in Figure 2.18. This is clearly reflected in the code of Figure 6.25.

An interesting property of the particular translator descriptions is that they are purely behavioural and make no assumption whatsoever about how an RTL synthesis tool will actually implement them. In fact, a typical tool will take advantage of common terms in equations (6.11) to perform hardware sharing. For example, refer back to Figure 2.17, and consider translator outputs J_1 and K_2. Observe that both include the term x_4+x_5 in their respective equations. A typical tool will notice this, and will share the corresponding logic gate appropriately. Note that this does not harm the TSC property of the checker. Indeed,

for any (J_j ,K_j) pair, the equations producing J_j and K_j cannot have common x_i inputs. Any logic sharing will therefore be between the expressions for J_j and a K_k, or J_j and a J_k, with $k \neq j$. This means that there can be no logic sharing circumstances under which a single fault in a gate may result in a bit flip in both J_j and K_j. Thus the fault secure property is preserved.

Using the cells described in this subsection, the control path self-checking schemes can be implemented. The approach taken in this work, is to output the controller self-checking block to a separate file, as a separate VHDL `entity` (during the synthesis post processing step mentioned at the beginning of this subsection). Clearly, this entity can be totally constructed using conventional parity checkers, LFSRs, 1-hot checkers, and dual-rail checkers as applicable per situation. Then the normal MOODS output file, already supplemented by the data path self-checking techniques of chapter 5, is further augmented with an instantiation of the control path checker, as a component within the overall RTL VHDL netlist, fed by the control signals. An additional 1-pair dual-rail checker is further used to compact the responses from the datapath and the controller self-checking schemes, and to produce the overall system health indication to the 2-bit output port already introduced in chapter 5. This way, the final HLS output may as applicable per situation be based on the following files :

- the usual RTL netlist
- the control path checker
- the normal MOODS cell library (§3.2.7)
- the fault-secure comparator library (§5.3.3.3)
- the dual-rail checker library (§5.3.3.3)
- the parity checker library
- the LFSR library
- the 1/n checker library

For example, a design with both full datapath self-checking and any parity-based controller self-checking will depend on a total of 7 files (all of the above except the 1/n checker library).

Finally, note that as well as enabling the implementation of the self-checking schemes in the context of this work, the development of a self-checking infrastructure environment in

the form of synthesisable VHDL components has research value as such (refer for example to [47]).

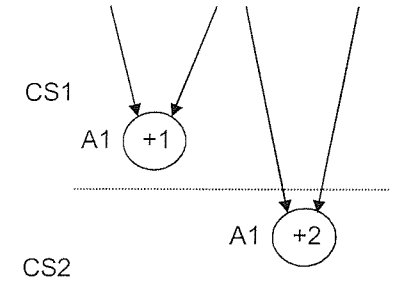
6.4.3 Facilitating Intrinsically Secure states

Although as already said the controller self-checking problem is not relevant to the synthesis tasks, it would be desirable to implement a mechanism within the synthesis process to direct the system towards creating more Intrinsically Secure states than it normally would. Such a mechanism would clearly allow experimentation with the concept of IS states, and evaluate their usefulness in practice.

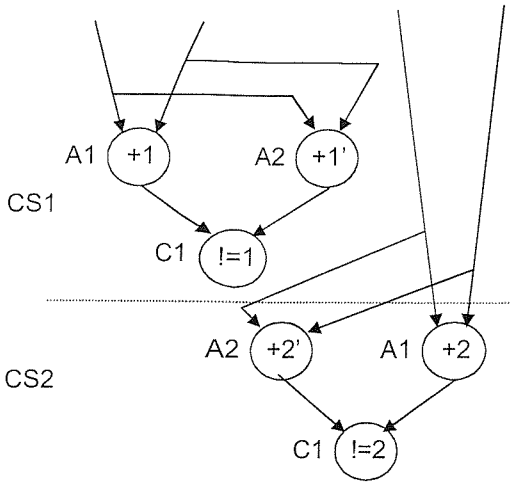
Refer to the DFG of Figure 6.26. In 6.26a, two control states are shown; one operation is scheduled at each one of them. The usual notations (§3.1.1) are used to signify that they are assigned to the same functional module. Figure 3.26b depicts a typical situation after duplication self-checking insertion and subsequent optimisation. The design is well optimised, with comparisons chained within the same CSs as the functional and redundant computations, but none of the control states is Intrinsically Secure. This author's design experience suggests that most designs tend to end up in such situations if the combination of simulated annealing and tailored heuristics explained in chapter 5 is applied.

Figure 3.26c depicts an alternative situation. Operation +1 has been moved one CS earlier, and this has allowed +2 to move up to the same CS as operations +1′ and !=1. This last move would not have been possible if +1 had not moved, since +1 and +2 are assigned to the same functional unit A1. This situation is particularly desirable for the experimental purposes of this chapter, since both CS2 and CS3 of Figure 6.26c are in fact Intrinsically Secure. The emergence of IS states can therefore be promoted by a synthesis heuristic that would move the design from situations such as that of Figure 6.26b to situations such as Figure 6.26c.

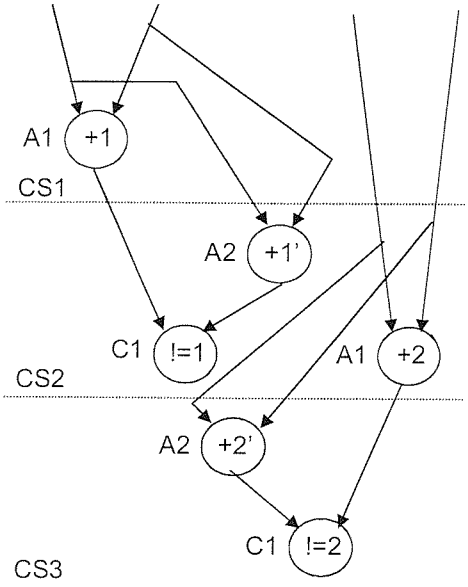
At this point, recall the set of transformations available within the standard MOODS suite (§3.2.3). Focus especially on the "merge fork and successor" TF8, and on the "unshare single instruction from control state" TF21 (Table 3.1) transformations. Notice that the application of TF21 on operation +1′ of CS1 in Figure 6.26b will create a dedicated control step for the operation. Data dependency between +1′ and !=1 will then necessarily



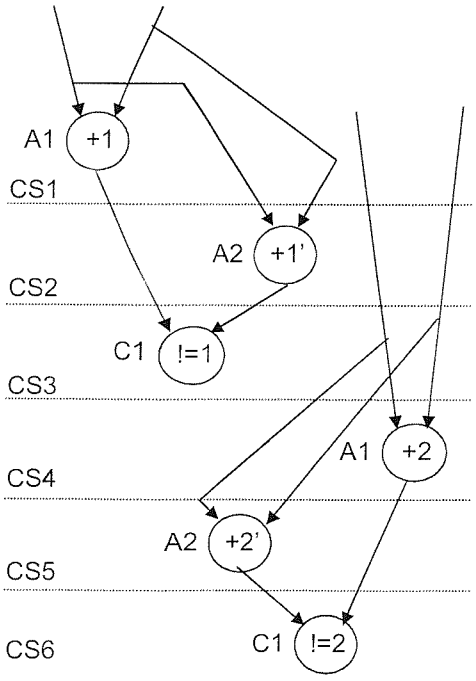
(a) Initial situation



(b) Typical self-checking situation



(c) Desirable situation, with 2 IS states



(d) Intermediate situation

Figure 6.26. Facilitating Intrinsically Secure states

create another CS, this time dedicated to comparison !=1. If the same transformation is applied to +2′, then two new CSs will similarly be created, this time for +2′ and !=2. This intermediate situation is depicted in Figure 6.26d. If control steps CS3 and CS4 are now merged using transformation TF8, then components C1 and A1 will both be active during the new merged state; this means that operation +1, *also* allocated to A1 will no more be allowed to move to the same control step as comparison !=1, since A1 will be occupied during that particular control step. Hence, this new state will always be Intrinsically Secure according to Definition 6.1, since the *functional* (+1) and *checking* (!=1) parts of the self-checking scheme composed of +1,+1′,!=1, will always be scheduled for different control steps. Subsequent optimisation using for example one of the heuristics of §3.2.5.2 will typically lead to the desirable situation of Figure 6.26c. Notice the combination of transformations that allowed the move : first all redundant operations were extracted from the shared control states using TF21, then control states where comparisons were scheduled, were merged with their successors using TF8. Of course, the underlying assumption throughout this explanation is that the "blocking" unit A1 cannot be unshared; it would therefore be sensible not to apply IS state creation within the simulated annealing block, but rather after it, and before the tailored heuristics.

Based on the above, the on-line test synthesis approach presented in §5.3.3.2 can be slightly amended to produce more Intrinsically Secure states, as follows :

- Step 1 : apply simulated annealing as in §5.3.3.2
- Step 2 : traverse all control steps, identify those that have all three parts (functional, redundant, and comparison operations) of self-checking schemes scheduled at them, and apply TF21 to the redundant operation
- Step 3 : repeat Step 2 until no more such CSs can be found
- Step 4 : apply TF8 to all control steps where fault-secure comparison operations have been scheduled
- Step 5 : repeat Step 4 until no more such CSs can be found
- Step 6 : apply tailored heuristics as in §5.3.3.2

Relatively short C++ functions implementing Steps 2 and 4 were provided to the MOODS system, together with suitable MOODS commands to enable the designer to use them through the MOODS interactive command prompt (Appendix A). Steps 3 and 5 are not automated, so it is left to the designer to make sure no more suitable CSs can be identified, or even decide to move to the tailored heuristics optimisation step prematurely.

Clearly, the procedure described here is not efficient as far as synthesis is concerned. Indeed, it would be better and faster for the design space exploration process to reach Figure 6.26c automatically, without having to go through 6.26b, also relying on user interaction to pass through 6.26d. However, bear in mind that the goal of this subsection is not to give an efficient "high-level synthesis for Intrinsically Secure states" approach; rather, it is to *somehow* facilitate the emergence of said states, for the sole purpose of experimental evaluation and comparisons, as will be made clear in the following §6.4.4. In that sense, the above rudimentary step-wise approach serves its purpose adequately.

As a concluding remark, it is to be noted that for the implementation purposes of this chapter, Intrinsically Secure states are considered according to the updated Definition 6.1′, with the threshold value t=7 as §6.1.3.2 suggests.

6.4.4 Experimental results

This subsection presents the experimental results obtained on the lines of the detailed analysis of §6.1, §6.2, §6.3, and the particular MOODS implementation details of this current section. In the following tables, a "Version 1" realisation refers to the design obtained by the usual synthesis process of chapter 5, given the user constraints, and additionally utilising the self-checking cells described in §6.4.2 to provide controller self-checking. "Version 2" signifies that the heuristic procedure of §6.4.3 has further been applied, thus in principle leading to more IS states. Results are given both for ASIC and FPGA implementations, for as many controller self-checking schemes as applicable per situation. As in chapter 5, dedicated technology library files were provided to the system for each different target technology. The behavioural synthesis RTL output was fed to the Mentor Graphics LeonardoSpectrum [132] RTL synthesis tool (version 2002e.16) in the case of ASIC implementation. When FPGA technology was targeted, the Synplicity Synplify [124] tool (version Pro 6.2) was used instead, while the design was further implemented using Xilinx Design Manager version 3.1i [125]. In both cases, the tables present results reported from the low level tools (Spectrum and Design Manager respectively); hence, they correspond to the most realistic area and delay estimations that can be obtained.

Tables 6.2 – 6.23 summarise all experimental results. The first and second columns in all tables denote the self-checking strategy applied to the data path and the controller respectively. Where applicable, the second columns also mention the number of IS states identified. The next three columns give the size and performance statistics of the particular implementations, in terms of logic gates used or FPGA slices occupied (depending on technology) as a merit of the design size, and number of clock cycles and maximum achievable frequency in MHz as a merit of the design performance. Finally, the last two columns provide the area overhead and speed penalty associated with including self-checking to the considered designs. In designs where both datapath and controller self-checking have been applied, the area overhead percentage reported accounts for both. This simply reflects the fact that, since the data path generally occupies most of the chip area, it is unlikely that a designer would want controller self-checking solely, but he or she would rather opt for a combined solution.

Tables 6.2 and 6.3 show the results obtained for a Version 1 and a Version 2 (respectively) implementations of the Tseng design, both cases with the same synthesis priorities and targeting the same ASIC technology. The first row in both tables shows the original design. All overhead percentages in the tables are always given with respect to this untestable version. A design with a self-checking data path is given immediately afterwards, followed by combinations of both datapath and controller self-checking, the latter alternatively taking all six forms described in this chapter (or as many as applicable in any given design). Table 6.2 highlights CTRL_6 as the cheapest of the six techniques among the Version 1 implementations, with a 58.4% overall hardware overhead. Also notice that 3 out of 5 states in the design are identified as IS, meaning that the majority of bidirectional controller faults will be detected, together with the unidirectional ones, providing almost complete confidence even in the most hostile environment. Finally, the degradation in the maximum frequency found in chapter 5 is naturally encountered here as well. Table 6.3 shows that CTRL_6 is also the cheapest approach among the Version 2 type designs. The IS states identified are 3 again, while compared to Table 6.2 the implementation is clearly more expensive, but a side effect of the heuristic of §6.4.3 is that higher frequency can now be achieved (~42MHz compared to ~14MHz). This is a result of breaking chains of functional – redundant – comparison operations originally scheduled for the same CS. The Table 6.3 results would therefore be preferable in a high frequency requirement scenario.

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	1768	4	54.6	N/A	N/A
inversion	-	2783	5	14.9	57.4	25.0
inversion	CTRL_1	2842	5	14.3	60.7	25.0
inversion	CTRL_2	2864	5	14.3	62.0	25.0
inversion	CTRL_3, 3 IS	2833	5	14.3	60.2	25.0
inversion	CTRL_4, 3 IS	2857	5	14.3	61.6	25.0
inversion	CTRL_5	2808	5	14.3	58.8	25.0
inversion	CTRL_6, 3 IS	2801	5	14.3	58.4	25.0

Table 6.2 : Tseng Benchmark Version 1 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area high, delay low, moderate clock period value

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	1768	4	54.6	N/A	N/A
inversion	-	2915	5	42.3	64.9	25.0
inversion	CTRL_1	3131	5	41.9	77.1	25.0
inversion	CTRL_2	3153	5	41.9	78.3	25.0
inversion	CTRL_3, 3 IS	3125	5	41.9	76.8	25.0
inversion	CTRL_4, 3 IS	3148	5	41.9	78.1	25.0
inversion	CTRL_5	3097	5	41.8	75.2	25.0
inversion	CTRL_6, 3 IS	3091	5	41.9	74.8	25.0

Table 6.3 : Tseng Benchmark Version 2 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area high, delay low, moderate clock period value

Tables 6.4 and 6.5 present corresponding results for the Diffeq benchmark. CTRL_6 is again highlighted as the most economical solution in both, while again Version 2 experiences a marginally higher area overhead, in conjunction with two additional IS states. Interestingly, the IS-state facilitating heuristic combined with the standard MOODS tailored heuristic gives rise to a Version 2 design that is comparatively faster than its Version 1 counterpart (saving 3 states in the critical path). Given the modest additional overhead, the CTRL_6 design of Table 6.5 is likely to be the preferred choice for this benchmark, especially if speed is a particularly critical concern.

Notice that both designs given so far have had controller self-checking versions utilising

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	3679	16	39.6	N/A	N/A
inversion	-	6075	25	39.6	65.1	56.3
inversion	CTRL_1	6400	25	37.8	74.0	56.3
inversion	CTRL_2	6424	25	37.8	74.6	56.3
inversion	CTRL_3, 5 IS	6361	25	37.8	72.9	56.3
inversion	CTRL_4, 5 IS	6377	25	37.8	73.3	56.3
inversion	CTRL_5	6246	25	37.8	69.8	56.3
inversion	CTRL_6, 5 IS	6237	25	37.7	69.5	56.3

Table 6.4 : Diffeq Benchmark Version 1 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area high, delay low, moderate clock period value

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	3679	16	39.6	N/A	N/A
inversion	-	6143	22	39.6	67.0	37.5
inversion	CTRL_1	6460	22	37.7	75.6	37.5
inversion	CTRL_2	6480	22	37.7	76.1	37.5
inversion	CTRL_3, 7 IS	6404	22	37.7	74.1	37.5
inversion	CTRL_4, 7 IS	6428	22	37.7	74.7	37.5
inversion	CTRL_5	6325	22	37.6	71.9	37.5
inversion	CTRL_6, 7 IS	6307	22	37.6	71.4	37.5

Table 6.5 : Diffeq Benchmark Version 2 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area high, delay low, moderate clock period value

the multi-process schemes CTRL_2 and CTRL_4. In fact, both designs are essentially single-process ones, but they also include short processes, solely responsible for the updating of system primary outputs. These processes are so short that indeed the hardware introduced to implement "dummy" processes (Figures 6.5 and 6.8) is more than the savings achieved through using a single parity checker; hence, in all tables so far CTRL_2 and CTRL_4 are more expensive than CTRL_1 and CTRL_3 respectively (recall the prediction of §6.2.6 that in order for hardware savings to be noticeable and significant a degree of parallelism of the order of 10 would be needed).

The case is different in the QRS benchmark (Tables 6.6 and 6.7). Here primary outputs are updated within the same process as the rest of the operation, therefore CTRL_2 and CTRL_4 are not applicable; thus the tables have two rows less. The familiar (from chapter 5) phenomenon of self-checking designs that are faster than the untestable ones can be observed here. CTRL_6 is once more the cheapest choice in both cases. Notably, Version 2 in Table 6.7 exceeds 100% in overhead when combined self-checking is applied; it is therefore expected that the Version 1 options of Table 6.6 would appear preferable.

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	7343	56	43.2	N/A	N/A
duplication	-	13278	51	23.4	80.8	-8.9
duplication	CTRL_1	13748	51	23.4	87.2	-8.9
duplication	CTRL_3, 11 IS	13648	51	23.4	85.9	-8.9
duplication	CTRL_5	13442	51	23.4	83.1	-8.9
duplication	CTRL_6, 11 IS	13417	51	23.4	82.7	-8.9

Table 6.6 : QRS Benchmark Version 1 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area low, delay high, strict clock period value

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	7343	56	43.2	N/A	N/A
duplication	-	14624	50	25.6	99.2	-10.7
duplication	CTRL_1	15288	50	25.0	108.2	-10.7
duplication	CTRL_3, 18 IS	15154	50	25.0	106.4	-10.7
duplication	CTRL_5	14985	50	24.9	104.1	-10.7
duplication	CTRL_6, 18 IS	14943	50	25.0	103.5	-10.7

Table 6.7 : QRS Benchmark Version 2 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area low, delay high, strict clock period value

Table 6.8 overviews the experiments conducted for an 8-bit Viterbi decoder. All operations in this design are of low bit width; no Intrinsically Secure states are therefore considered, and the corresponding schemes CTRL_3, CTRL_4 and CTRL_6 are not applicable. The design is highly parallel (8 concurrent processes). Comparison of the CTRL_1 and CTRL_2-based solutions now verifies the hardware savings associated with moving from the former to the latter (67.8% overhead reduced to 63.5%). However, the solution based on 1/n checking is again the cheapest with 51.7%.

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	3262	5	106.9	N/A	N/A
duplication	-	4734	7	127.4	45.1	40.0
duplication	CTRL_1	5475	7	116.4	67.8	40.0
duplication	CTRL_2	5333	7	116.9	63.5	40.0
duplication	CTRL_5	4947	7	115.6	51.7	40.0

Table 6.8 :8-bit viterbi decoder synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area high, delay low, moderate clock period value

Tables 6.9 and 6.10 present corresponding results for the elliptical filter design. The Version 1 datapath-only self-checking realisation experiences an overhead of 95.7%. Naturally, one would reject this option and change the specifications in the search of a better solution on the lines of chapter 5; however, for the experimental purposes of this work it is interesting to add a controller self-checking scheme and observe if this will raise the cost to more than 100%. In fact, Table 6.9 reveals that only CTRL_2 produces a cost of exactly 100%, while all other techniques remain below that line, with CTRL_6 once again the least expensive. Version 2 in this case offers both a particularly expensive, and slower design; hence, the 4 additional IS states it produces are unlikely to appear tempting.

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	3697	9	35.7	N/A	N/A
duplication	-	7236	12	21.4	95.7	33.3
duplication	CTRL_1	7374	12	21.3	99.5	33.3
duplication	CTRL_2	7393	12	21.3	100.0	33.3
duplication	CTRL_3, 6 IS	7328	12	21.4	98.2	33.3
duplication	CTRL_4, 6 IS	7349	12	21.4	98.8	33.3
duplication	CTRL_5	7292	12	21.4	97.2	33.3
duplication	CTRL_6, 6 IS	7283	12	21.2	97.0	33.3

Table 6.9 : Ellip Benchmark Version 1 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area high, delay high, moderate clock period value

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	3697	9	35.7	N/A	N/A
duplication	-	7897	14	26.6	113.6	55.6
duplication	CTRL_1	8247	14	24.0	123.1	55.6
duplication	CTRL_2	8258	14	24.0	123.4	55.6
duplication	CTRL_3, 10 IS	8166	14	24.1	120.9	55.6
duplication	CTRL_4, 10 IS	8187	14	24.1	121.4	55.6
duplication	CTRL_5	8154	14	24.1	120.6	55.6
duplication	CTRL_6, 10 IS	8140	14	24.1	120.2	55.6

Table 6.10 : Ellip Benchmark Version 2 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area high, delay high, moderate clock period value

Tables 6.11 and 6.12 show the results obtained for the GCD benchmark design. Both versions were considered; however, none of them included any IS states. Clearly, the heuristic of §6.4.3 failed to create any such states. Interestingly, however, it gave rise to a marginally cheaper design. Further, the design is strictly single-process. This limits the choices of controller self-checking techniques to only CTRL_1 and CTRL_5, with the latter appearing cheaper in both versions. It can be observed that the CTRL_5 choice of the Version 2 design is the cheapest overall, and notably achieves a maximum frequency value of 45MHz, that is, even higher than the untestable design itself. Version 1 might be preferable if the shorter (by a single state) critical path it offers is of any significance in the

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	1022	9	42.0	N/A	N/A
duplication	-	1471	8	40.3	43.9	-11.1
duplication	CTRL_1	1559	8	40.3	52.5	-11.1
duplication	CTRL_5	1502	8	43.7	47.0	-11.1

Table 6.11 :GCD Benchmark Version 1 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area high, delay low, moderate clock period value

data path testing	control path testing	area (gates)	speed (cycles)	maximum frequency (MHz)	hardware overhead (gates %)	speed penalty (cycles %)
-	-	1022	9	42.0	N/A	N/A
duplication	-	1455	9	45.1	42.4	0.0
duplication	CTRL_1	1563	9	45.1	52.9	0.0
duplication	CTRL_5	1499	9	45.0	46.7	0.0

Table 6.12 :GCD Benchmark Version 2 synthesis results (Target Technology Alcatel CMOS 0.35 VLSI), synthesis priorities : area high, delay low, moderate clock period value

context of the considered project.

An interesting observation of the experiments so far, is that in all of them, *solutions based on 1-hot checking are cheaper than their parity checking counterparts*. Of course, this does not invalidate the fact that parity is indeed in principle the cheapest among error detecting codes; what makes the above parity schemes comparatively expensive is the application of the LFSRs in the checkers (Figure 6.9), to provide the self-testing property. In other words, it appears that parity checking is not the best solution for the particular problem, unless strict adherence to self-checking theory could be abandoned. Before endorsing this rather premature conclusion, it is instructive to investigate the effect of target technology.

In the following Tables 6.13 – 6.23, the above experiments are effectively repeated, this time targeting Xilinx Virtex FPGA devices [106]. As the experiments of chapter 5 have also illustrated, designs targeting this technology are less straightforward than the respective ASIC targeting ones, and are particularly hard to assess at any design stage other than the final implementation, since the amount of FPGA resources utilised greatly depends on the low level synthesis tools and the packing algorithms they employ.

Tables 6.13 and 6.14 present both versions of the Tseng benchmark targeting the XCV1000 Xilinx FPGA component. Comparing with Tables 6.2 and 6.3, one can notice that the designer requirements provided to MOODS are the same as in the VLSI targeting experiments. This is actually true for all experiments hereafter. The results, however, are often different. In the particular case of Table 6.13, first of all notice that, in contrast to Table 6.2, no IS states are identified for the considered benchmark. Therefore CTRL_3, CTRL_4 and CTRL_6 are meaningless, and CTRL_5 is the cheapest solution. The maxi-

mum frequency is hugely degraded; this is true for Version 2 (Table 6.14) as well, once more in contrast with its VLSI counterpart (Table 6.3).

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	127	4	43	N/A	N/A
inversion	-	193	5	4	52.0	25.0
inversion	CTRL_1	198	5	4	55.9	25.0
inversion	CTRL_2	199	5	4	56.7	25.0
inversion	CTRL_5	193	5	4	52.0	25.0

Table 6.13 : Tseng Benchmark Version 1 synthesis results (Target Technology Xilinx XCV1000 FPGA), synthesis priorities : area high, delay low, moderate clock period value

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	127	4	43	N/A	N/A
inversion	-	222	5	4	74.8	25.0
inversion	CTRL_1	237	5	4	86.6	25.0
inversion	CTRL_2	236	5	4	85.8	25.0
inversion	CTRL_3, 3 IS	228	5	4	79.5	25.0
inversion	CTRL_4, 3 IS	230	5	4	81.1	25.0
inversion	CTRL_5	226	5	4	78.0	25.0
inversion	CTRL_6, 3 IS	230	5	4	81.1	25.0

Table 6.14 : Tseng Benchmark Version 2 synthesis results (Target Technology Xilinx XCV1000 FPGA), synthesis priorities : area high, delay low, moderate clock period value

Tables 6.15 and 6.16 are devoted to the Diffeq benchmark. Two facts are particularly noticeable in these tables. Firstly, for the first time parity-based solutions are cheaper than 1/n-based ones (CTRL_4 in Table 6.15 and CTRL_3 in 6.16). Secondly, Version 2 here not only imposes a higher overhead, but also fails to achieve its main goal, since it creates one Intrinsically Secure state *less* than Version 1. This is not very surprising, since the heuristic of §6.4.3 was based on a simple observation and did not offer any comprehensive analysis or sophisticated synthesis procedure; experiments up to now have shown that in principle it directs designs towards more IS states, but failures are possible. In contrast, in

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	260	18	9	N/A	N/A
inversion	-	439	23	6	68.8	27.8
inversion	CTRL_1	457	23	6	75.8	27.8
inversion	CTRL_2	456	23	6	75.4	27.8
inversion	CTRL_3, 8 IS	455	23	6	75.0	27.8
inversion	CTRL_4, 8 IS	453	23	5	74.2	27.8
inversion	CTRL_5	460	23	6	76.9	27.8
inversion	CTRL_6, 8 IS	454	23	5	74.6	27.8

Table 6.15 : Diffeq Benchmark Version 1 synthesis results (Target Technology Xilinx XCV800 FPGA), synthesis priorities : area high, delay low, moderate clock period value

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	260	18	9	N/A	N/A
inversion	-	450	23	8	73.1	27.8
inversion	CTRL_1	466	23	8	79.2	27.8
inversion	CTRL_2	466	23	7	79.2	27.8
inversion	CTRL_3, 7 IS	462	23	7	77.7	27.8
inversion	CTRL_4, 7 IS	464	23	8	78.5	27.8
inversion	CTRL_5	469	23	8	80.4	27.8
inversion	CTRL_6, 7 IS	467	23	7	79.6	27.8

Table 6.16 : Diffeq Benchmark Version 2 synthesis results (Target Technology Xilinx XCV800 FPGA), synthesis priorities : area high, delay low, moderate clock period value

the QRS benchmark experiments presented in the following Tables 6.17 and 6.18, it can be noticed that Version 2 produces a total of 24 IS, accounting for more than a third of all states in the design. CTRL_3 is the cheapest option in both versions; Version 1 generally occupies less slices, but Version 2 is faster, primarily because of allowing somewhat higher frequencies, but also because of a slightly shorter critical path (1 CS). It is worthwhile to compare Tables 6.17 and 6.18 with the VLSI-targeting equivalents 6.6 and 6.7. Apart from the natural difference in frequency values, further differences in the number of clock cycles, the overhead percentages and the most economical controller self-checking technique stress the effect of target technology and the importance of providing for both ASIC and FPGA solutions within high-level synthesis, to accommodate a wider range of designer needs.

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	591	69	17.4	N/A	N/A
duplication	-	837	67	1.9	41.6	-2.9
duplication	CTRL_1	944	67	2.0	59.7	-2.9
duplication	CTRL_3, 19 IS	933	67	2.0	57.9	-2.9
duplication	CTRL_5	956	67	1.9	61.8	-2.9
duplication	CTRL_6, 19 IS	944	67	2.2	59.7	-2.9

Table 6.17 : QRS Benchmark Version 1 synthesis results (Target Technology Xilinx XCV1000 FPGA), synthesis priorities : area low, delay high, strict clock period value

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	591	69	17.4	N/A	N/A
duplication	-	905	66	3.8	53.1	-4.3
duplication	CTRL_1	994	66	3.2	68.2	-4.3
duplication	CTRL_3, 24 IS	983	66	3.3	66.3	-4.3
duplication	CTRL_5	1001	66	3.8	69.4	-4.3
duplication	CTRL_6, 24 IS	992	66	3.8	67.9	-4.3

Table 6.18 : QRS Benchmark Version 2 synthesis results (Target Technology Xilinx XCV1000 FPGA), synthesis priorities : area low, delay high, strict clock period value

Table 6.19 is dedicated to the 8-bit FPGA-based Viterbi decoder. The overhead percentages are always well above 100%, with a significant delay penalty as well. These overheads are in agreement with the previous observation on Table 5.19, that the particular benchmark is not suitably accommodated by duplication testing. What can be kept out of this experiment though is that the CTRL_2 scheme provides the least expensive solution for the first time in this experimentation.

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	174	4	38	N/A	N/A
duplication	-	447	7	31	156.9	75.0
duplication	CTRL_1	517	7	29	197.1	75.0
duplication	CTRL_2	508	7	31	192.0	75.0
duplication	CTRL_5	541	7	30	210.9	75.0

Table 6.19 :8-bit viterbi decoder synthesis results (Target Technology Xilinx XCV1000 FPGA), synthesis priorities : area high, delay low, moderate clock period value

The Elliptical filter design also experiences very high overheads in this technology (Tables 6.20 and 6.21). Probably the most interesting point in these results is the disagreement between Versions 1 and 2 regarding the most efficient and economical realisation (CTRL_6

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	229	10	5.0	N/A	N/A
duplication	-	492	17	4.2	114.8	70.0
duplication	CTRL_1	525	17	3.9	129.3	70.0
duplication	CTRL_2	528	17	4.1	130.6	70.0
duplication	CTRL_3, 10 IS	523	17	4.3	128.4	70.0
duplication	CTRL_4, 10 IS	523	17	4.3	128.4	70.0
duplication	CTRL_5	530	17	4.2	131.4	70.0
duplication	CTRL_6, 10 IS	523	17	4.5	128.4	70.0

Table 6.20 : Ellip Benchmark Version 1 synthesis results (Target Technology Xilinx XCV1000 FPGA), synthesis priorities : area high, delay high, moderate clock period value

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	229	10	5.0	N/A	N/A
duplication	-	582	18	2.3	154.1	80.0
duplication	CTRL_1	605	18	2.3	164.2	80.0
duplication	CTRL_2	607	18	2.3	165.1	80.0
duplication	CTRL_3, 14 IS	595	18	2.2	160.0	80.0
duplication	CTRL_4, 14 IS	594	18	2.3	159.4	80.0
duplication	CTRL_5	615	18	2.3	168.6	80.0
duplication	CTRL_6, 14 IS	600	18	2.3	162.0	80.0

Table 6.21 : Ellip Benchmark Version 2 synthesis results (Target Technology Xilinx XCV1000 FPGA), synthesis priorities : area high, delay high, moderate clock period value

in Version 1, CTRL_4 in Version 2). Note also that the majority of states in Version 2 have been made Intrinsically Secure (14 out of 18).

Finally, Tables 6.22 and 6.23 give the FPGA results for the GCD benchmark. Version 2 (Table 6.23) succeeds in creating 3 Intrinsically Secure states (in contrast to Table 6.12), but utilising them does not save hardware (indeed, CTRL_1 and CTRL_3 give the same overhead).

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	85	10	45	N/A	N/A
duplication	-	124	9	34	45.9	-10.0
duplication	CTRL_1	131	9	33	54.1	-10.0
duplication	CTRL_5	132	9	33	55.3	-10.0

Table 6.22 :GCD Benchmark Version 1 synthesis results (Target Technology Xilinx XCV1000 FPGA), synthesis priorities : area high, delay low, moderate clock period value

data path testing	control path testing	area (slices)	speed (cycles)	maximum frequency (MHz)	hardware overhead (slices %)	speed penalty (cycles %)
-	-	85	10	45	N/A	N/A
duplication	-	140	11	35	64.7	10.0
duplication	CTRL_1	148	11	35	74.1	10.0
duplication	CTRL_3, 3 IS	148	11	35	74.1	10.0
duplication	CTRL_5	153	11	33	80.0	10.0
duplication	CTRL_6, 3 IS	154	11	33	81.2	10.0

Table 6.23 :GCD Benchmark Version 2 synthesis results (Target Technology Xilinx XCV1000 FPGA), synthesis priorities : area high, delay low, moderate clock period value

The FPGA-targeting experiments have shown that in such technology, no definite control path self-checking technique can be favoured a priori; it is important to conduct a number of experiments and choose the most appropriate for any given case. Of course, it can be argued that the number of occupied slices does not make a real difference in the price of the design, as long as it fits into the target FPGA part. If a designer adopts such an approach, then the preferable designs are probably different from the less resource-occupying ones highlighted in the tables above. For example, if maximum fault detection

capabilities in an extremely hostile environment is required, then CTRL_6 in Table 6.21 may be preferable over CTRL_4, since it will detect all unidirectional faults due to 1-hot checking, and in addition all bidirectional faults affecting any of the 14 (out of a total of 18) states – in practice, the vast majority of bidirectional faults.

6.4.5 Discussion

As mentioned in the discussion of the datapath self-checking experiments (§5.3.3.6), numerical comparisons with previously presented controller / datapath self-checking solutions are not always reliable, mainly due to differences in target technology. The situation is even more complicated in the controller self-checking problem of this chapter, because only two previous publications [23, 134] acknowledge the need to provide a dedicated self-checking scheme for the control path. Even then, [23] mentions parity checking, but does not elaborate on how to achieve the totally self-checking goal and does not quantify the proportional contribution of the control path self-checking resources to the overall hardware overhead. [134] proposes the expectably expensive solution of full hardware duplication and once more does not report on the relative overhead due to the controller checking hardware. Other publications referring to FSM self-checking by means of a variety of EDCs [38, 133, 37, 22] are not relevant, because the FSMs they target do not control a sequential datapath (i.e. the right-hand side part of the architecture of Figure 6.1 is entirely missing). Therefore the architectures they address are different from the one considered here. The conclusion is that there is no published data that the results of this work can be compared against; in fact, the work in this chapter is the first to comprehensively address all aspects of the control path self-checking problem in a controller / datapath hardware implementation. In spite of the absence of material for comparison, some comments evaluating the results of §6.4.4 are provided in the following.

With respect to the overall cheapest solution, the results herein have shown that it greatly depends on target technology. There is a definite trend in favour of 1/n checking using Khakbaz's checker of [29] when VLSI technology is targeted. In contrast, there is no clear winner when FPGA parts are used alternatively. In fact, each one of the six self-checking solutions implemented in this chapter was found to be the cheapest option in at least one of the FPGA targeting experiments. Therefore experimentation is needed before a choice

is made in these cases. Such experimentation is clearly facilitated by the material of both chapter 5 and this present chapter.

It is also possible to evaluate the two ideas that gave rise to the parity-based variations in §6.2, namely the single checker in multi-process designs idea, and the Intrinsically Secure states concept. As for the former, it was verified that it can benefit designs with a high degree of parallelism. For example, an approximate 4% of hardware savings (with respect to using multiple checkers) was experienced for the 8-process design tried in table 6.8. As regards IS states, comparisons of respective techniques (i.e. CTRL_1 vs CTRL_3, CTRL_2 vs CTRL_4, and CTRL_5 vs CTRL_6) show that the associated hardware savings exist, but are very modest, on average around 1%. The conclusion from this is that utilising *existing* Intrinsically Secure states in a given design is a valid option, leading to some little hardware savings and offering increased protection against multiple fault scenarios in particularly hostile environments. Notably, this is very much in line with the prediction given in §6.2.6. Further, Version 2 type implementations were on most tables more expensive than their Version 1 counterparts. One reason for that is the mandatory introduction of additional registers, along the lines of Figure 6.3. On the other hand, it was also found that often Version 2 designs can run at higher frequencies (the most illustrative example is the comparison between tables 6.2 and 6.3). The conclusion is that modifying optimised designs to create additional Intrinsically Secure states is not advisable due to significant extra hardware, unless the hostility of the operating environment is a major concern and the extra cost can thus be justified and / or high frequency operation is desired.

Notably, it is possible to implement controller self-checking as described herein in a tool other than MOODS. For this purpose, it is enough to perform an analysis of the control path model of the synthesis tool at hand (similar to the analysis of §6.4.1) and amend appropriately if any problems are identified (i.e. if any controller faults may corrupt control signals of even multiplicity and / or create bidirectional errors). This is typically done by suitably replicating selected pieces of logic, as [37] and [22] have widely covered.

6.5 Summary

Overall, the discussion of this chapter establishes that the control path self-checking problem is in fact much more complicated than in the considerably simpler situations ad-

dressed in the past, has a variety of possible solutions, while interaction with the synthesis system provides opportunities for existing data path self-checking construct reuse, for enhanced system operation reliability. The key elements that define the contribution of this thesis to the field of control path self-checking are the following three :

- both parity and 1-out-of-n self-checking solutions are considered and compared, under alternative technologies
- the option of using a single parity checker in highly parallel designs is provided
- increased security against very hostile environments is achieved, through the definition, identification and exploitation of Intrinsically Secure states

As a by-product of the development phase of this work, a comprehensive library of synthesisable VHDL descriptions of parity, 1/n, and dual-rail checkers is produced.

Together with the datapath self-checking solutions of chapter 5, this chapter implements *combined* controller and datapath self-checking design, in a unified, integral, highly automated and designer-friendly high-level synthesis environment, enabling the rapid realisation of hardware for safety-critical applications.

Chapter 7

Reliability Evaluation

A theoretical and, where needed, experimental evaluation of the robustness of the implemented datapath (chapter 5) and controller (chapter 6) self-checking schemes is given here. In the datapath case, the totally self-checking (TSC) property (Definition 2.3) is guaranteed under the single fault condition stated in Hypothesis 2.1, *if* all code words appear at the inputs of the duplication schemes. The validity of the hypothesis is therefore arguably strongly dependent not only on the structure of the system, but also on the input data it is fed with. It is therefore interesting to consider the robustness of the datapath scheme in cases when the set of functional inputs is restricted, potentially resulting in faults remaining undetected and leading to the accumulation of multiple faults. In contrast, the controller self-checking scheme receives inputs that are totally predictable at design time, and in principle independent of the data the system receives. Further, the implementations of chapter 6 take the TSC property into full account (§6.2.7, §6.3.1.2). A theoretical evaluation therefore fully covers the issue.

Section 7.1 deals with the reliability of datapath self-checking, initially by expressing theoretical concerns, and subsequently by setting up a fault simulation environment and evaluating the scheme through experiments. Section 7.2 addresses the control path self-checking properties, effectively by formally summarising the error detection properties of the alternative schemes presented in chapter 6. Section 7.3 concludes the chapter.

7.1 Datapath self-checking

In this section, an estimation of the robustness of the datapath self-checking scheme is given. As already mentioned in the beginning of chapter 2, the stuck-at fault model is as-

sumed throughout this thesis. Of course, this model is not literally valid in the on-line testing context. Indeed, any wire physically stuck at a particular value would naturally be detected during the off-line production test. Transient faults are more relevant to this work. In order to model them, at any given point of time a faulty signal is considered to behave *as if* it was stuck-at a logic value. After a period of time equal to the defined duration of the fault, the signal is allowed to behave properly and assume the value of the wire driving it. This way, "temporary" stuck-at faults are used to model the transient faulty *behaviour* of a faulty circuit element rather than its actual state, that is, the *effect* rather than the physical *cause* of a fault.

Further, not all stuck-at faults of a system are considered. Rather, only those at the inputs and outputs of the datapath RTL modules are of interest. This idea is explained by referring to Figure 7.1, where a datapath module, its duplicate and the associated comparator are shown. Three faults f_1, f_2 and f_3 are illustrated in Module 1, each one signifying that a given wire within the module is stuck at a particular value. Let us assume that there exist electrical connections between the stuck wires and Module 1 output bits, in particular between f_1 and output bit x_l, f_2 and bit x_i, and f_3 and both bits x_j and x_k. Define also the following four conceptual faults at the outputs of Module 1.

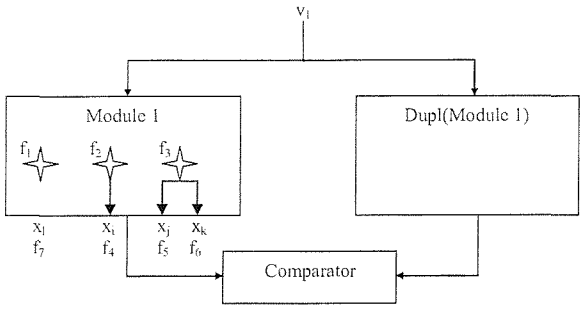


Figure 7.1 : The duplication checking scheme

f_4 : x_i stuck-at-0

f_5 : x_j stuck-at-1

f_6 : x_k stuck-at-1

f_7 : x_l stuck-at-0

Assume that f_4, f_5, f_6 and f_7 are not physically present in the system.

Further consider a random primary input vector v_1 feeding Module 1. In the scenario depicted by Figure 7.1, fault f_1 is not sensitized by v_1, and therefore remains internal to the module (*latent* fault event). In the same scenario, fault f_2 is propagated to the module output, corrupting bit x_i. Finally, f_3 is propagated to the output through both possible paths,

thus corrupting two different module output bits, x_j and x_k. Without loss of generality, let us assume that under the presence of the corresponding faults f_2 and f_3, and when fed by the particular input, x_i assumes the D (1/0) value, while x_j and x_k become $\overline{D}$ (0/1). Then the behaviour of the system under fault f_2 is clearly equivalent to the behaviour that the system would experience if the above defined fault f_4 was present. It follows that in this case a single fault in the RTL module primary output (f_4) fully represents an internal module fault (f_2). Moreover, the behaviour under f_3 is equivalent to the hypothetical behaviour under the superposition of faults f_5 and f_6. It can therefore be stated that a certain class of internal faults (f_3 being a member of this class) can be fully represented by the suitable superposition of multiple faults at the module outputs (in this example f_5 and f_6). Furthermore, observe that *either* f_5 or f_6 would *alone* cause the comparator to detect faulty operation, exactly as f_3 would do. In that sense, f_3 can be considered "loosely" equivalent to either single fault f_5 or f_6 *alone*. This does *not* mean to say that e.g. f_3 and f_5 are generally equivalent; however, the nature of the considered problem is such that here their primarily interesting effects (i.e. triggering the fault detection mechanism) are equivalent, although the two faults clearly lead to overall different situations. Indeed, the output of Module 1 under the presence of f_3 is different from its output under the presence of f_5, assuming that in both cases it is fed by v_1. Still, the information that both output values are erroneous and therefore detectable means that from the perspective of this thesis the "interesting" effect of internal faults that corrupt *multiple* output values can be represented by *single* primary output faults.

As regards fault f_1, if no available Module 1 input can sensitize it, then it remains internal to the module forever, and it does not corrupt the system operation. Such faults are of no interest and not considered in this work. Alternatively, suppose that there is at least one available input vector $v_2 \neq v_1$ that sensitizes f_1, such that f_1 manifests itself at the module output, naturally by corrupting output bit x_l to which it is electrically connected. Further, without loss of generality assume that at the instance depicted in Figure 7.1, with Module 1 fed by v_1, x_l correctly assumes the logic 0 value. Therefore, at the instance of Figure 7.1, the behaviour of the system under fault f_1 at the instance of Figure 7.1 can be considered to be equivalent to the hypothetical behaviour under fault f_7 defined above. This is a valid statement, since both faults are latent at the particular moment. It has therefore been established that even latent internal faults can be modelled by equivalent latent single faults at the primary outputs of the RTL modules.

Moreover, depending on the module functionality and inputs feeding it, stuck-at faults at the module *inputs* can provide convenient means to model multiple *output* faults. For example, consider an adder module and a corrupted $\overline{D}$ (0/1) value at bit position i of the first adder operand. If the respective bit of the second operand is a fault-free 1, then all output bits to the left of i, and up to the first fault-free 0 will be inverted. In this case, *multiple output* faults can be modelled by a *single* stuck-at fault at the *input*. It is therefore useful to include all input stuck-at faults in the set of considered faults as well, unless there is a clear and data-independent 1 to 1 equivalence between an input and an output fault, suggested by the particular module functionality.

The above discussion has established that single stuck-at faults at the inputs and outputs of RTL modules satisfactorily model the system faulty behaviour under Hypothesis 2.1 (single internal fault), *in the context of the considered problem*. A particular advantage of this approach is that it is fully consistent with the high level design philosophy, since it makes no assumption whatsoever about the gate-level structural implementations of RTL modules, but is only concerned with their behaviour. This thesis is not further concerned with the general, recently surfaced idea of fault representativeness at the RT level. The relevant literature [140, 141, 142] can be consulted for statistical analyses and discussions of this still open issue.

7.1.1 Theoretical concerns

Recall once more Hypothesis 2.1, repeated in the following Hypothesis 7.1 for convenience :

Hypothesis 7.1: Faults occur one at a time, and the time distance between the occurrences of two consecutive faults is long enough for all the available input code words to be applied to the circuit.

Let us also recall the totally self-checking property established through definitions 2.1 – 2.5. In short, Module 1 and Dupl(Module 1) of Figure 7.1 will need to be such that under the presence of an internal fault, the following two properties should be satisfied :

a) fault secure property : for every available input word, the comparator input will either be fault-free or a non-code word (i.e. it cannot be an *incorrect code word*)

b) self-testing property : at least one of the available input words sensitizes the internal fault, i.e. produces a non-code output

As for the comparator of Figure 7.1, it additionally needs to exclusively map input code words to output code words and vice versa (code disjoint property).

Given that duplication and, where applicable (§5.2.2), inversion testing are fault secure by nature (§2.2.2.1, §5.2.2), if Hypothesis 7.1 is accepted for a scheme like the one of Figure 7.1, one would sensibly state that, in a fault-free scenario, Module 1 and its duplicate will produce all possible code outputs, thus feeding the comparator with all possible code words. Assuming a comparator based on a suitably structured (fault secure and code-disjoint) dual-rail checker, as is the case in this work (§5.3.3.3), this leads to the conclusion that the datapath self-checking scheme is totally self-checking, and therefore detects all single faults in any of the functional, duplicate or comparator modules.

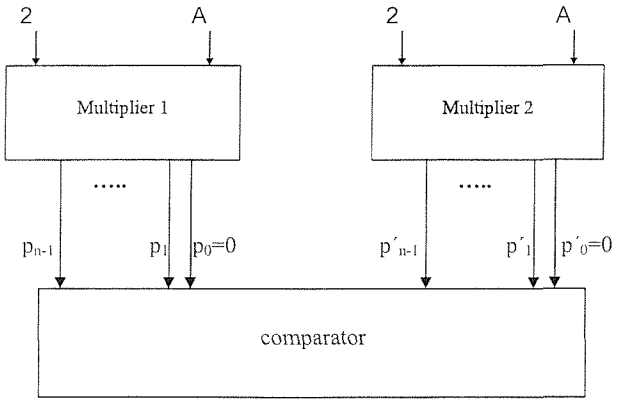


Figure 7.2 : Multiplication by 2

While the discussion in the above paragraph is valid, it silently assumes that the module inputs are random, that all possible inputs are available, and that they have equal probability to appear. This is a sensible assumption when the self-checking scheme is considered in isolation; let us, however, not forget that in the context of a complex overall system, operations are embedded deep into a design, being fed by the outputs of other operations, the other operations themselves fed by further previous levels of operations etc. This relationship is illustratively depicted in the DFG representation of a circuit (§3.1.1). The effect of this, is that, while primary inputs can in principle be considered random, the randomness and availability of the inputs of operations "lower" in the DFG are not guaranteed; in fact, these inputs greatly depend not only on the primary inputs, but also on the actual functionality and on the presence of constants in the data flow graph. At times it is obvious that *not all* possible inputs are available. Such a characteristic situation is shown in Figure 7.2. The functional module (multiplier 1) performs the operation 2×A, where A is an (n/2)-bit number. Multiplier 2

duplicates the operation, and both outputs are fed to the comparator. Clearly, multiplication by 2 never produces an odd number. Consequently, the LSBs p_0 and p'_0 of both outputs will always be 0 under fault free operation. Therefore, the comparator will always miss all input code words for which $p_0=p'_0=1$, that is, half of all possible code words. Further, the dual-rail checker within the comparator is never fed by the rows of a suitable $4\times(2\times n)$ matrix (Lemma 2.3 and [58]), and therefore the self-testing property is not achieved. In practice, this means that any fault in the checker that is equivalent e.g. to the LSB of the left hand side operand to be stuck-at-0 cannot be detected and will remain in the design for ever. Now, if an additional fault in the functional Multiplier 1 causes p_0 to be stuck-at-1, then the corrupted value will not be detected, it will be led to the rest of the system and thus hinder the overall system operation. The example is analogous to the one described in §2.2.1.1 referring to Figure 2.11.

The discussion has established that there can be cases within a DFG for which Hypothesis 7.1 is not enough to guarantee the TSC property, resulting in the possibility that faults remain undetected. In order for this to have disastrous effects on the system functionality, a subsequent fault in the system must corrupt *selected* modules at *selected* times. To understand this, refer back to the example of Figure 7.2, and remember that typically such a self-checking scheme will be one of a few tens of such schemes in the overall system. Consider the above mentioned fault scenario, wherein the LSB of the comparator left hand side operand is stuck-at-0 and therefore undetectable. In order for the next fault to have disastrous effects, it must hit the *particular* scheme (among tens of others), in a *particular* way (causing p_0 to be stuck-at-1, and not effecting any other bit in any other way). A fault in a different scheme or with a slightly different effect is more likely to be detected rather than cause a fault escape. Intuitively, given the typical complexity of the considered systems, featuring a few thousands of possible RTL faults, it can be argued that the probability of a disastrous fault effect is rather insignificant. Of course, this has to be backed by experimental data, as done in the following §7.1.2.

7.1.2 Experimental evaluation

Recall the Transparent Fault Injection and Simulation technique of §4.2.1 (also [118, 49]). Clearly, using the gate models with fault injection capabilities it proposes, fault simulation at gate-level netlists can conveniently be conducted. In order for the technique to be appli-

cable at the RT level according to the model of §7.1, RTL component models with fault injection capabilities need to be developed. Such components will effectively define an RTL cell library with fault injection capabilities, as an extension to the standard MOODS cell library (§3.2.7), thus setting up a fault simulation environment to supplement on-line test synthesis. As §4.2.1 established, this environment can easily utilize a commercial digital simulation tool such as the very popular ModelSim tool [115].

7.1.2.1 Transparent Fault Injection and Simulation at the RTL

A straightforward RTL extension of the transparent fault injection and simulation technique is given here, through a "pseudo"-VHDL example of a generic RT-level N-bit adder with fault injection capabilities (Figure 7.3). The model makes use of the fault_inject package provided in Figure 4.1. As can be seen in Figure 7.3, appropriate `_mask` vector variables are defined for all module input and output ports. In fact, two such vectors are defined for each port, the first corresponding to stuck-at-0 type faults and the second to stuck-at-1s (point #1 in the figure, at the declaration part of the VHDL process `nn`). Just like in the gate-level case, a unique, suitably-named local fault variable is created for every modelled RTL fault when simulation starts (point #2). Appropriate values are assigned to the mask vectors in every simulation instance (point #3), depending on which fault is simulated at the given instance. An element of the *stuck-at 0 (1) mask vector* for signal x is assigned a 0 (1) value if the corresponding fault is simulated. Subsequently the mask vector is ANDed (ORed) with signal x, in order to produce the effective value that is going to contribute to the simulation output, also taking into account the fault-free module functionality (point #4). This clearly defines a "mutant" N-bit adder, equivalent to the mutant gates concept encountered in §4.2. In line with the transparent nature of the gate-level technique of §4.2.1, ANDing and ORing here are *conceptual*, as are the mask variables, fault pointers and fault model records. They do *not* involve the introduction of any physical hardware gates; hence, the (non-synthesisable) fault simulated cell model is "structurally" equivalent to the synthesisable model, effectively meaning that no extra fault lines need to be included in the design for fault simulation purposes. When no fault is simulated, the two models are behaviourally equivalent as well. Indeed, it can be verified that the model of Figure 7.3 computes a proper unsigned addition (at point #4) in the fault-free case (i.e., when all stuck-at-0 mask vectors carry the all-1s value, and all stuck-at-1 masks bear all 0s). The situation is clearly analogous to its gate-level counterpart.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.fault_inject.all;
entity UADD_1 is
  generic (n: positive := 1);
  port (in1, in2 : in std_logic_vector (n-1 downto 0);
        sum : out std_logic_vector (n downto 0));
end UADD_1;
architecture inject_fault of UADD_1 is
begin
nn : process is
  variable in1_sa0, in1_sa1, in2_sa0, in2_sa1 : fault_ptr_array (n-1 downto 0) := (others => null);
  -- #1
  variable sum_sa0, sum_sa1 : fault_ptr_array (n downto 0) := (others => null);
  variable in1_sa0_mask, in2_sa0_mask : std_logic_vector (n-1 downto 0) := (others => '1');
  variable in1_sa1_mask, in2_sa1_mask : std_logic_vector (n-1 downto 0) := (others => '0');
  variable sum_sa0_mask : std_logic_vector (n downto 0) := (others => '1');
  variable sum_sa1_mask : std_logic_vector (n downto 0) := (others => '0');
begin
  -- #2
  if in1_sa0(0) = null then
    wait for 1 ns;
    for i in 0 to n-1 loop
      in1_sa0(i) := new fault_model'(
        new string'(inject_fault'instance_name &
            "in1(" & integer'image(i) & ")_sa0"),
            false, false, first_fault);
      first_fault := in1_sa0(i);
      -- objects in1_sa1(i), in2_sa0(i), in2_sa1(i), sum_sa0(i), sum_sa1(i)
      -- are created similarly
    end loop;
    sum_sa0(n) := new fault_model'(
      new string'(inject_fault'instance_name &
          "sum(" & integer'image(n) & ")_sa0"),
          false, false, first_fault);
    first_fault := sum_sa0(n);
    sum_sa1(n) := new fault_model'(
      new string'(inject_fault'instance_name &
          "sum(" & integer'image(n) & ")_sa1"),
          false, false, first_fault);
    first_fault := sum_sa1(n);
  end if;
-- #3
for i in 0 to n-1 loop
    if in1_sa0(i).simulating then
      in1_sa0_mask(i) := '0';
    else
      in1_sa0_mask(i) := '1';
    end if;
    if in1_sa1(i).simulating then
      in1_sa1_mask(i) := '1';
    else
      in1_sa1_mask(i) := '0';
    end if;
    -- mask elements in2_sa0_mask(i), in2_sa1_mask(i), sum_sa0_mask(i),
    -- sum_sa1_mask(i) are handled similarly
  end loop;
  if sum_sa0(n).simulating then
    sum_sa0_mask(n) := '0';
  else
    sum_sa0_mask(n) := '1';
  end if;
  if sum_sa1(n).simulating then
    sum_sa1_mask(n) := '1';
  else
    sum_sa1_mask(n) := '0';
  end if;
  -- #4
  sum <= sum_sa1_mask or
        (sum_sa0_mask and
        (std_logic_vector(unsigned("0" & (in1_sa1_mask or (in1_sa0_mask and in1))) +
                          unsigned("0" & (in2_sa1_mask or (in2_sa0_mask and in2))))));
  wait on in1, in2;
end process nn;
end architecture inject_fault;
```

Figure 7.3 : RTL N-bit unsigned adder cell with fault injection capabilities

No obvious equivalent or dominant faults can be found in this adder module, leading to a total of 2*(3*N+1) total modelled faults. However, even in RTL modules there are cases when not all input and output line faults need to be considered. An example is the generic left shifter module pseudo-VHDL template of Figure 7.4. In this module, the second input in2 corresponds to the number of bits by which input in1 will be left-shifted. Mask vectors are employed just as in the adder case, however this time the module functionality implies that the output signal bits are either hardwired to appropriate input bits, or directly connected to logic 0. There is no point in explicitly modelling faults for the former (since they are equivalent to corresponding input faults), while only stuck-at-1 faults need to be considered for the latter (since a possible stuck-at-0 would be equivalent to the fault-free operation). Hence the number of output faults in the model of Figure 7.4 is reduced. These ideas are reflected in the figure through the absence of fault pointers and masks corresponding to stuck-at-0 type faults in the outputs, and also through the reduced range in the loop taking care of output stuck-at-1 faults (point #1 in the code fragment). Other than that, the philosophy of the fault-injectable shifter module clearly follows that of the adder.

```
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all, work.fault_inject.all;
entity SLL_1 is
  generic (n: positive := 1;
           m: positive := 1);
  port (in1: in std_logic_vector (n-1 downto 0);
        in2: in std_logic_vector (m-1 downto 0);
        output: out std_logic_vector (n-1 downto 0) );
end SLL_1;
architecture inject_fault of SLL_1 is
begin
nn : process is
  variable in1_sa0, in1_sa1 : fault_ptr_array (n-1 downto 0) := (others => null);
  variable output_sa1 : fault_ptr_array (n-1 downto 0) := (others => null);
  variable in2_sa0, in2_sa1 : fault_ptr_array (m-1 downto 0) := (others => null);
  variable in1_sa0_mask : std_logic_vector (n-1 downto 0) := (others => '1');
  variable in1_sa1_mask, output_sa1_mask : std_logic_vector (n-1 downto 0) := (others => '0');
  variable in2_sa0_mask : std_logic_vector (m-1 downto 0) := (others => '1');
  variable in2_sa1_mask : std_logic_vector (m-1 downto 0) := (others => '0');
begin
  if in1_sa0(0) = null then
    wait for 1 ns;
      -- create new stuck fault records as in the adder example
  end if;
  -- #1
  for i in 0 to to_integer(unsigned(in2))-1 loop
    -- fix output_sa1_mask(i)
  end loop;
  for i in 0 to n-1 loop
    -- fix in1_sa0_mask(i) and in1_sa1_mask(i)
  end loop;
  for i in 0 to m-1 loop
    -- fix in2_sa0_mask(i) and in2_sa1_mask(i)
  end loop;
  output <= output_sa1_mask or
            std_logic_vector(shift_left(unsigned((in1 and in1_sa0_mask) or in1_sa1_mask),
                                        to_integer(unsigned(((in2 and in2_sa0_mask) or in2_sa1_mask)))));
  wait on in1, in2;
end process nn;
end architecture inject_fault;
```

Figure 7.4 : RTL generic shift left module with fault injection capabilities

Defining fault-injectable VHDL models for the rest of the standard MOODS cells (§3.2.7) proceeds exactly as the two examples above. At this point, it has to be noted that no fault

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity NEQ_3 is
    generic (n: positive := 1);
    port (in1, in2 : in std_logic_vector (n-1 downto 0);
          output: out std_logic_vector(1 downto 0));
end NEQ_3;
architecture structure of NEQ_3 is

function steps (h: positive) return integer is
-- auxiliary function : returns the number of checker arrays (levels) needed for an h-pair checker
-- see Figure 5.13
  variable i : integer :=0;
begin
  i:=0;
  loop
    exit when h/(2**i)=1;
    i:=i+1;
  end loop;
  if (n rem (2**i))>0 then
    return i+1;
  else
    return i;
  end if;
end steps;

constant levels : integer:= steps(n);

function no_of_int_sig (p: positive; i: integer) return positive is
-- auxiliary function : returns the number of output signals of the ith level of a p-pair checker
-- see Figure 5.13
  variable pairs, old : positive;
begin
  if i>levels then
    return 2;
  end if;
  pairs:=p;
  for k in 1 to i loop
    if k/=1 then
      old:=pairs;
      pairs:=pairs/2;
      if (old rem 2)=1 then
        pairs:=pairs+1;
      end if;
    end if;
  end loop;
  return 2*pairs;
end no_of_int_sig;

function index (num: positive; sum_over:integer) return integer is
-- auxiliary function : helps the calculation of the starting location of the outputs of level sum_over within
-- the intermediate_signal array (see below), in a num-pair checker
  variable sigs : integer:=0;
begin
  if num=1 then
    return 2;
  end if;
  for i in 1 to sum_over loop
    sigs := sigs + no_of_int_sig(num,i);
  end loop;
  return sigs;
end index;

constant tot : integer:= index(n,levels) + 2;

-- all signals connecting checker arrays
signal intermediate_signals : std_logic_vector(tot-1 downto 0);

component CHK_ARR
    generic (m: positive := 1);
    port (in1 : in std_logic_vector (m-1 downto 0);
          in2 : in std_logic_vector (m-1 downto 0);
          output: out std_logic_vector ((m + (m rem 2))-1 downto 0));
end component;

begin
  intermediate_signals(tot-1 downto tot-n) <= in1;
  intermediate_signals(tot-n-1 downto tot-2*n) <= in2;
  Z0: if n=1 generate -- trivial
      intermediate_signals(1 downto 0) <= intermediate_signals(3 downto 2);
    end generate Z0;
  Z1: for i in 1 to levels generate
      U1: CHK_ARR generic map (no_of_int_sig(n,i)/2)
                  port map (intermediate_signals(tot-index(n,i-1)-1 downto tot-index(n,i-1)-no_of_int_sig(n,i)/2),
                            intermediate_signals(tot-index(n,i-1)-no_of_int_sig(n,i)/2-1 downto tot-index(n,i)),
                            intermediate_signals(tot-index(n,i)-1 downto tot-index(n,i+1)));
   end generate Z1;
   output <= intermediate_signals(1 downto 0);
end;
```

Figure 7.5 : A generic N-pair dual-rail checker

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity NEQ_3 is
    generic (n: positive := 1);
    port (in1, in2 : in std_logic_vector (n-1 downto 0);
          output: out std_logic_vector(1 downto 0));
end NEQ_3;

use work.fault_inject.all;
architecture inject_fault of NEQ_3 is

-- function, constant, signal and component declarations exactly as in Figure 7.5

begin

-- generate statements exactly as in Figure 7.5

nn : process is
  variable in1_sa0, in1_sa1, in2_sa0, in2_sa1 : fault_ptr_array (n-1 downto 0) := (others => null);
  variable in1_sa0_mask, in2_sa0_mask : std_logic_vector (n-1 downto 0) := (others => '1');
  variable in1_sa1_mask, in2_sa1_mask : std_logic_vector (n-1 downto 0) := (others => '0');
begin
  if in1_sa0(0) = null then
    for i in 0 to n-1 loop
      in1_sa0(i) := new fault_model'(
        new string'(inject_fault'instance_name &
            "in1(" & integer'image(i) & ")_sa0"),
            false, false, first_fault);
      first_fault := in1_sa0(i);

-- all other fault variables in both in1 and in2 are handled similarly

    end loop;
   end if;
  for i in 0 to n-1 loop
    if in1_sa0(i).simulating then
      in1_sa0_mask(i) := '0';
    else
      in1_sa0_mask(i) := '1';
    end if;

-- similarly for the other fault variables

  end loop;
   intermediate_signals(tot-1 downto tot-n) <= in1_sa1_mask or (in1_sa0_mask and in1);
   intermediate_signals(tot-n-1 downto tot-2*n) <= in2_sa1_mask or (in2_sa0_mask and in2);
   wait on in1, in2;
 end process;

nn2: process is
  variable out_sa0, out_sa1 : fault_ptr_array (1 downto 0) := (others => null);
  variable out_sa0_mask : std_logic_vector (1 downto 0) := "11";
  variable out_sa1_mask : std_logic_vector (1 downto 0) := "00";
begin
  if out_sa0(0)=null then
   out_sa0(0) := new fault_model'(
      new string'(inject_fault'instance_name &
          "out_sa0(0)"),
          false, false, first_fault);
    first_fault := out_sa0(0);

-- the same for out_sa0(1), out_sa1(0) and out_sa1(1)

  if out_sa0(0).simulating then
    out_sa0_mask(0) := '0';
  else
    out_sa0_mask(0) := '1';
  end if;

-- the same for out_sa0(1), out_sa1(0) and out_sa1(1)

  output <= out_sa1_mask or (out_sa0_mask and intermediate_signals(1 downto 0));
  wait on intermediate_signals(1 downto 0);
end process;
end;
```

Figure 7.6 : A generic N-pair dual-rail checker with fault injection capabilities

injection is considered and naturally no fault-injectable models needed for storage elements (registers) or interconnect (multiplexer, sign extension) modules; instead, these modules are assumed free of faults. This issue is revisited later in §7.1.3 and comments on its implications provided. Further, no fault injection was considered for control cells either. The reason for this is that the controller self-checking scheme is totally self-checking (as chapter 6 established), therefore its behaviour under the presence of faults, and its fault detection capabilities are fully predictable, as mentioned throughout chapter 6, and will be summarized in §7.2.

The fault secure comparator and dual-rail checker components of §5.3.3.3 can have faults injected in their inputs and outputs by likewise defining suitable models, following exactly the same principles as in the standard MOODS models. However, recall that an enormous number of dual-rail checkers and comparators were automatically produced by suitable software in §5.3.3.3. Writing separate mutant components for each one of them would be an impractically time-consumming process. To cope with this problem, a concise generic description of an N-pair dual-rail checker was firstly configured, shown in Figure 7.5. The description uses the `CHK_ARR` cell of Figure 5.14. It further defines and utilises three auxiliary arithmetic functions; comments on the functions are provided in the figure with references to the generic dual-rail checker scheme of Figure 5.13. The description appears complicated but it fully describes Figure 5.13 for any value of N; for example, it can be verified that for N=16 it becomes equivalent to Figure 5.15. In fact, there are RTL synthesis tools that cannot synthesize the code of Figure 7.5. The reason for that is the VHDL component instantiation statement labelled U1 towards the bottom of Figure 7.5. This statement defines three slices of the long `intermediate_signals` array as the actual ports of component U1. However, the slice boundary definition includes variable `i` (the "loop" variable of the "generate" statement Z1). Using a variable in slice boundary definitions was found by this author not to be acceptable by all VHDL compilers. For this purpose, the description of Figure 7.5 is not generally synthesisable and cannot in principle be used instead of the cells produced in §5.3.3.3. However, the description was accepted by the compiler of the simulator tool [115] used herein for simulation experiments. The fact that a *single* description is used for all values of N is particularly advantageous, since it enables the development of a respective "mutant" description, as Figure 7.6 outlines. The general structure of the description of Figure 7.6 is exactly the same as that of Figure 7.5; the difference is that in Figure 7.6 the simple assignment statements that involve the inputs

and the output of the checker are replaced by suitable processes that control the injection of faults in the checker ports, using suitable mask variables, exactly as done in Figure 7.3 for the N-bit adder. The fault-free behaviour of the mutant checker of Figure 7.6 is identical to the behaviour of the original checker of Figure 7.5, which in turns behaves identically to the fully synthesizable modules of §5.3.3.3. Exactly as done in §5.3.3.3, a generic N-pair fault secure comparator with fault injection capabilities is described by simply complementing one of the dual-rail inputs in Figure 7.6.

A test bench to control the overall fault simulation campaign can now be written as the gate-level prototype of Figure 4.4 outlined. As an interesting word of note, the input / output interfaces of a gate-level and an RT-level design are identical, and so effectively the same test bench can be used for fault simulation at both levels, if so desired. In either case, a test bench written as in Figure 4.4 can be tuned to implement exhaustive, deterministic, or random injection experiments. Multiple faults can be injected as well as single ones (by simply activating more than one `.simulating` fields in the suitable fault model record). Furthermore, by activating a fault and then deactivating it at a chosen simulation time (by resetting the respective `.simulating` field), one can model transient (as opposed to permanent) faults, again simply by suitably amending the testbench. The corresponding input vectors file (`vectors.txt`) can include an exhaustive, or an incomplete but predetermined (even random) set of test vectors. Finally, the processing and presentation of obtained results can be carried out as desirable through the testbench directives. Therefore, the designer has all flexibility to tailor the simulation experiments through the test bench and input vectors, to reach the desired conclusion, as applicable per situation.

In the simulation experiments described in the following two subsections, the commercial simulator used was Model Technology ModelSim, version SE Plus 5.5e [115].

7.1.2.2 Injecting single faults

It has already been established that the duplication and (where applicable) inversion datapath self-checking schemes of chapter 5 are fault secure against single faults. Therefore, any single stuck-at fault in any of the functional, redundant or comparison modules embedded within an overall self-checking datapath is expected either to be detected or to remain latent. To verify this, the technique of §7.1.2.1 was used to conduct a number of

faults			
injected	latent	detected	escaped
100000	65912	34088	0

Table 7.1 Tseng benchmark fault simulation results (independent experiments)

fault simulation experiments on a self-checking version of the Tseng datapath, produced as explained in chapter 5. In particular, the version randomly picked for the experiment was the fourth one of Table 5.11 (the one employing duplication testing). The choice of version is, however, not important, since all versions are equally secure against RTL faults. Random faults were injected, and random inputs applied; this way, the experiment emulated the operation of a system whose operating conditions cannot possibly be known in advance. Further, since in this subsection it is only single faults that are of interest, whenever a fault remained latent it was removed, and the next one injected at a different simulation time point, after the previous removal. Therefore, this subsection addresses independent experiments.

The results, shown in Table 7.1, indeed verify the fault secure property, by demonstrating no fault escapes at all. Notice that a particularly extensive number of experiments were conducted (100000). In the particular benchmark circuit, the overall number of injectable RTL faults was much smaller (exactly 758, automatically calculated through the test bench, as a byproduct of the simulation). This means that the total 100000 experiments included several incarnations of every fault, each time under different operating conditions and different input values, thus increasing confidence in the system dependability. It is worth noticing that the majority of experiments led to latent fault events (§7.1). This can be explained by the fact that both the injected faults and the applied inputs were random. As a result, in several cases an RTL input or output was driven to logic value $x \in \{0,1\}$, while at the same time a stuck-at-x was simulated at the same signal. This clearly leaves the fault latent; statistically these situations should account for 50% of all experiments. Further recall that the Tseng benchmark includes logic operations (AND, OR) as seen in the VHDL of Appendix B. At times injected faults at the logic function operators were prevented from manifesting themselves simply by the natural masking properties of logic functions (e.g. 0 AND D = 0). These two phenomena resulted in the increased percentage of latent faults.

7.1.2.3 Injecting Multiple Faults

The effect of the accumulation of multiple latent faults in a self-checking design is experimentally addressed here. As the experiments of Table 7.1 established, in a typical design, there is a high probability that a fault hitting the design is not detected immediately, but remains latent. The scenario of the previous subsection removed such latent faults, considering them harmless transients; here, they are considered to remain permanently on the system, thus giving rise to the said multiple fault accumulation. Such accumulation can be particularly severe in situations where a self-checking scheme receives a restricted subset of all possible input words (as in Figure 7.2), or in very hostile environments, where faults occur very frequently, so that a self-checking configuration does not have the time to receive all available input words. Clearly such accumulation is expected to result in a probability of faults remaining undetected "for ever", and potentially corrupting the design primary outputs. A situation where this can happen was explained around Figure 7.2, wherein an undetectable latent fault in the checker, together with a subsequent fault in the functional module can cause a fault escape. Another typical fault escape scenario would be the one shown in Figure 7.7, where the two modules in the duplication testing scheme have their respective *n*-position bits stuck-at the same value *x*, and *in addition* the common input at that time happens to be such that the said bit should assume the logic complement ~*x* value under fault free operation. Once more, assuming that all modeled faults have equal probability to appear, and further taking into account that even small benchmark designs include anything between a few hundreds and a few thousands of such RT-level faults, one can trivially conclude that the probability that such a situation appear is very small (significantly less than 1%).

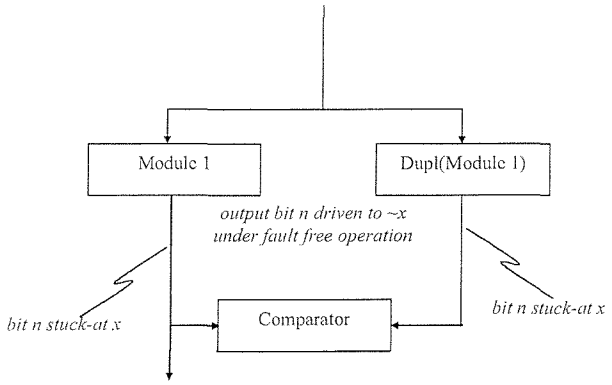


Figure 7.7 : A possible fault escape

inputs	faults injected	latent fault events	detection events	fault escapes
"sensible"	50000	23189	26811	0
random	50000	39227	10773	0

Table 7.2 Diffeq benchmark fault simulation results (multiple faults)

This is further verified by the series of fault simulation experiments performed on a self-checking version of the Diffeq benchmark. The version used was taken from Table 5.13, and it was the one on the second row, featuring a "mixed" self-checking strategy (duplication and inversion as determined by the system to be better applicable per situation), and including a total of 776 identifiable faults in the RTL datapath. This time, latent faults were not removed, but remained in the system, and further faults were subsequently injected. All accumulated faults were removed every time a detection event occurred. Two sets of experiments were conducted : in the first, random but "sensible" input vectors were applied; while in the second totally random ones were used. The significance of this is related to the functionality of the particular benchmark. Indeed, "sensible" inputs cause the system to perform a number of repetitions of its main functional loop (see Appendix B for the VHDL code), while totally random inputs are very likely to leave significant parts of the datapath idle (and therefore unable to detect any faults) for long periods of time. This way, high accumulations of faults were expected to be achieved in the design. The results are summarized in Table 7.2. The table indeed verifies that the number of latent fault situations significantly increases when totally random inputs are used. Most importantly, it is demonstrated that 100000 simulation runs, including several accumulations of faults, under a rich variety of conditions and inputs, were once more unable to produce a single failure. This experimentally verifies the prediction of §7.1.1, also mentioned in this subsection : although the TSC property cannot be guaranteed for the duplication- / inversion-based schemes of the self-checking datapath, and consequently fault escapes are theoretically possible, the probability that such an escape occurs is insignificant. Differently put, given a self-checking datapath produced as chapter 5 of this work explains, and assuming a number of latent faults in the datapath, the probability that the next fault hitting the system will either be detected or remain harmlessly latent, is overwhelmingly higher than the probability that the said fault will interact with an existing latent fault to cause a disastrous fault escape.

7.1.2.4 Common mode faults

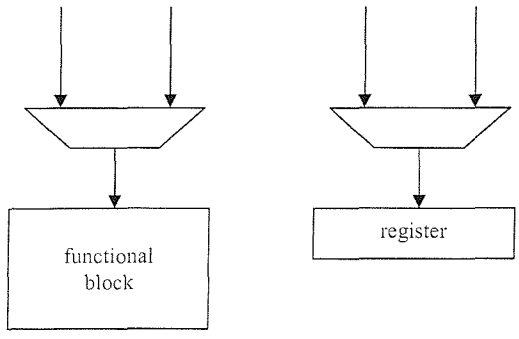
The discussion so far has assumed that all modelled faults have the same probability to appear in the design at any given moment. However, in particular VLSI technologies, minor defects in the fabrication process of standard cell masks can result in common mode

faults (§2.2.2.1). Simply put, this means that *all* cells of a particular kind (e.g. all adders) may feature some common, minor malfunction, not detectable in off-line production test, due to its insignificant initial effect. Under certain environmental conditions or over time, such defects may develop into logic faults, thus resulting in *the same* type of faulty behaviour in all cells of the said type. In the context of this discussion, referring back to Figure 7.7, this effectively means that given a latent stuck-at-x type fault at the n-th output bit of the left-hand side module, the probability that the next fault in the system will be a disastrous stuck-at-x at the n-th bit of the right-hand side module, is significantly higher than the probability that an unrelated fault will hit another part of the circuit. In a particular application, whether or not common mode faults are likely to occur is something that can be determined only in the context of the given application, especially taking into account the target technology, reliability of fabrication process, and robustness of the off-line production test.

Common mode faults are known to escape duplication testing schemes where both duplicate modules have been produced by the same mask. Therefore, a high probability of such faults is the only significant threat the datapath self-checking scheme of this work has to face. To alleviate the risk, traditionally [51] diverse duplication is applied (§2.2.2), wherein duplicate modules are behaviourally equivalent, but structurally different. Diverse duplication cannot be currently adopted within MOODS, due to all datapath modules having a single realisation within the cell library (§3.2.7). Assuming subsequent development work leading to alternative cells, however, the synthesis process of chapter 5 would be perfectly applicable to diverse duplication. In this work, inversion testing is proposed as a valid alternative, if the frequency degradation often associated with it is tolerable in the context of the particular project.

7.1.3 Faults in the interconnect and storage units

As has been obvious in the discussion so far, and explicitly mentioned in §7.1.2.1, the datapath self-checking scheme addressed here is dedicated to the functional datapath modules of designs resulting from high-level synthesis. The other parts of the datapath, namely the interconnect and storage elements, are, as a first approach, assumed fault-free. This is a sensible assumption, taking into account that the theme itself of this work is the high-level synthesis of *functional hardware* blocks. In such blocks, the chip area occupied by



(a) feeding a functional block *(b) feeding a register*

Figure 7.8 : Multiplexer configurations

functional modules is significantly higher than that occupied by storage and interconnect. Therefore, an environmental factor affecting the chip is much more likely to affect the area occupied by a functional module, rather than that occupied by a register or a multiplexer. It has to be clarified that considered hardware blocks exclude large memory blocks. If such blocks appear in a system, self-checking design principles have to be applied to them as well (typically some variation of parity checking, see for example [143, 144]), but this is out of the scope of this thesis (indeed, it would concern self-checking design considered at the *system level*).

A further look at multiplexer faults further backs the fault-free assumption. Consider Figure 7.8. It depicts the two situations when a multiplexer is needed : to feed a functional block (7.8a), or to feed a register (7.8b). Clearly, in 7.8a, the behaviour of a faulty multiplexer delivering a corrupted value to the functional module input, is equivalent to a *fault-free* multiplexer that correctly feeds a *faulty* functional block, in particular a functional block whose behaviour can be modelled by a suitable stuck-at fault in its input. Therefore, the faulty behaviour of the multiplexer is covered by the already mentioned RTL fault model of §7.1. Similarly, in Figure 7.8b, a faulty value delivered to the register by a corrupted multiplexer, can be considered equivalent to a corrupted register receiving a correct value. Of course, registers are not covered by the assumed fault model, and therefore such a fault would be disastrous.

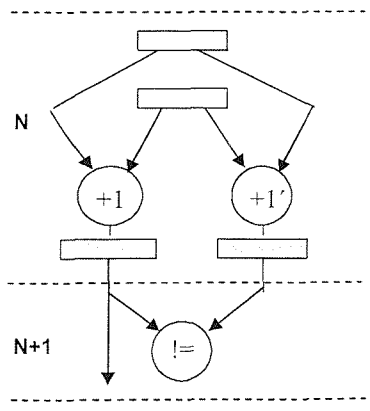


Figure 7.9 : Faulty registers equivalent to faulty functional modules

Let us now focus exclusively on registers. Careful examination of a few design data flow graphs reveals that there are classes of registers whose faults are in fact equivalent to functional module faults. Figure 7.9 shows such a DFG,

highlighting two registers in this category. The figure is notably similar to Figure 6.3; indeed, it depicts the – dominant throughout this thesis – duplication testing scheme. The comparison operation has been scheduled one control step after the functional and the redundant ones; therefore, the two highlighted registers carry the intermediate results across the boundary of CSs N and N+1. Clearly, any active fault in any of the highlighted storage units will propagate to the comparator input – it will therefore be equivalent to a suitable comparator fault, hence covered by the assumed model. However, notice that any fault in any of the non-highlighted registers feeding the functional or redundant operation will *not* be detected, since each register feeds both adders, thus producing the *same* erroneous result at the adder outputs. Interestingly, the scheme behaves very much like if under the presence of a common mode fault, thus exposing a defenceless part of the circuit.

7.2 Control path self-checking

The six alternative control path self-checking techniques presented in chapter 6 have been designed to strictly adhere to self-checking design theory. That is, they all achieve the totally self-checking goal under Hypothesis 2.1 / 7.1. In contrast to the datapath case, the hypothesis is now particularly valid. To understand this, refer back to Figure 6.1, and consider the controller as a single module, receiving the conditional signals as inputs and producing the control signals at the output. Recall (§6.4.1) that every internal fault in the MOODS-generated controller may affect a *single* control signal. Further consider that the control signals / control path checker inputs (Figure 6.15), in all realistic situations, are of the order of 100 at the very most, compared to 2^n different comparator inputs in a data path duplication self-checking scheme with, e.g., 16 being a typical value for n, yielding ~650 times more values. Finally, no situation analogous to Figure 7.2 can be conceived for the control path; that is, all control states are visited (and all control signals produced) during the usual system operation, even if some of them are visited less frequently than others. In summary, the control path checker is extremely likely to receive all of its available code words between the occurrences of two consecutive faults, because they are relatively very few, and because nothing prevents them from being produced. In combination with the single fault property, this directly supports Hypothesis 2.1 / 7.1. The conclusion, hence, is that any single fault in the controller or the control path checker will definitely be detected before the next one occurs, thus excluding latent faults and accumulations of faults resulting from them.

It is instructive to go one step further, and consider a particularly hostile environment, in which multiple faults may hit the controller at any given time, thus producing multiple faulty control signals. Although this is probably an unrealistically hostile scenario, not normally considered in self-checking literature, it is interesting to note that the proposed schemes of chapter 6 provide enhanced detection capabilities that accommodate several such situations as well. These enhanced capabilities should be familiar since there were references to them throughout chapter 6; the following comments remind them and Table 7.3 formally summarises them. The CTRL_1 technique (Figure 6.4) detects all single or odd-multiplicity errors among the control signals of any single process individually. The CTRL_2 scheme of Figure 6.5 detects all single or odd-multiplicity errors among the control signals of all processes in the design accumulatively. CTRL_3 (Figure 6.7) offers single and odd-multiplicity error detection on an individual process basis just like CTRL_1; further, it detects any combination of faults so long as at least one of them corrupts the control signal of an Intrinsically Secure state (§6.1.3.1). The CTRL_4 scheme (Figure 6.8) has the same capabilities as CTRL_2, with the addition that identifying and taking IS states into account once more provides detection of any multiplicity errors that corrupt at least one IS control signal. CTRL_5 (Figure 6.12) detects all unidirectional errors on individual processes, and so does CTRL_6 (Figure 6.14), with the addition that the latter detects even bidirectional errors if any of them corrupts the control signal of an IS state.

technique	detection capabilities
CTRL_1	single or odd-multiplicity errors per process
CTRL_2	single or odd-multiplicity errors in all the control signals
CTRL_3	any multiplicity errors if the control signal of an IS state is corrupted signle or odd-multiplicity errors per process otherwise
CTRL_4	any multiplicity errors if the control signal of an IS state is corrupted signle or odd-multiplicity errors in all the control states otherwise
CTRL_5	any multiplicity unidirectional errors per process
CTRL_6	any multiplicity unidirectional errors per process, plus bidirectional errors for which the $\overline{D}$ value is assumed by an IS-state control signal

Table 7.3 : Error-detecting properties of controller self-checking techniques

Note that in the above evaluation of CTRL_3, CTRL_4 and CTRL_6, it is assumed that the precautions of §6.1.3.2 have been respected, so that fault escapes related to Intrinsically Secure states are practically very unlikely. As a reminder, this means that IS states are considered only when the data path bus is reasonably wide, while appropriate registers

are reset to a value that has a low probability of occurrence as soon as their functional contents are not needed anymore.

7.3 Summary

To summarize, this chapter has theoretically and experimentally established the reliability properties of the self-checking schemes of this thesis. In particular :

- the datapath scheme succeeds in its primary mission, that is, it is extremely robust, detecting all realistic fault scenarios affecting datapath functional modules. An exception to this can be common mode faults. Since such faults may or may not appear depending on the dependability of the production line in individual situations, this does not compromise the success of the technique. Further, if frequent common mode faults are expected, the inversion testing technique can provide a defence against them.
- although not explicitly targeting them, the datapath scheme also detects a portion of interconnect and storage unit faults. The remaining multiplexer and register faults can cause disastrous fault escapes, but the area they occupy on the chip is small enough to demote this to a minor issue.
- the alternative control path self-checking schemes are totally self-checking by construction and therefore detect all single controller faults; in addition, they also defend against a variety of multiple-fault scenarios.

Chapter 8

Future Research and Conclusion

This last chapter comprises two short sections. Section 8.1 proposes ideas for future work, while §8.2 gives the final concluding remarks of this thesis.

8.1 Future research directions

There are two families of research themes that can expand the work of this thesis :

- algorithms for on-line test synthesis
- expanding the fault detection capabilities provided herein to implement fault *tolerance*

The motivation for the first direction is that the modified version of the general-purpose simulated annealing algorithm defined in §5.3.3.2 was shown in the experimental results to be useful for designs that include up to around 300 operations (§5.3.3.5). Indeed, it was found that the tool run-time would probably be unacceptably high for bigger designs. While 300 operations is enough to accommodate a good number of practical designs, and it is still about double the size of anything presented in the past, this author expects that dedicated research on synthesis algorithms can take good advantage of the "case for on-line test synthesis" made in this thesis and configure automatic design flows that would explore the three-dimensional design space faster than the random and general-purpose simulated annealing choice. Such algorithms would probably need to be entirely new heuristics that would take into account the nature itself of the self-checking resource insertion problem, while still not neglecting the traditional high-level synthesis criteria. It would be particularly interesting to investigate *constructive* (as opposed to transformational) high-level on-line test synthesis algorithms, motivated by the fact that previous research on constructive algorithms has produced excellent results [100].

The second proposed research direction effectively refers to implementing "high-level synthesis for fault tolerance" as an extension of high-level synthesis for on-line testability. Very much as in the on-line testing case, a comprehensive survey of fault tolerance techniques will be needed, the most suitable for inclusion in synthesis will need to be chosen, and further tool implementation / expansion details through suitable transformations, algorithms and metrics will have to be devised. A complication of fault tolerance is that choosing the most suitable technique will be likely to depend both on target technology and on the assumed fault scenario (targeting transient or permanent faults). That is, while it was possible to define generic RT-level, technology-independent solutions for the on-line testing problem that were proved robust even in very hostile environments, this author feels that this will not apply for the fault tolerance problem. The implication is that multiple techniques will probably need to be implemented within the synthesis tool and the designer will be required to make a pre-synthesis choice of technique.

8.2 Concluding remarks

The work described in this thesis has produced an *integral, on-line test synthesis system*, based on the original MOODS behavioural synthesis suite.

It is the first time on-line testability is thoroughly integrated into the core of the synthesis process in a fully automatic manner. This is particularly achieved in the datapath self-checking scheme of chapter 5, and visualised by the 3-dimensional design space used, through the definition of an arithmetic expression that quantifies on-line testability. The resulting tool offers fast, painless, technology-independent and versatile exploration of the 3-dimensional space, all inherited from traditional high-level synthesis. Complex VHDL constructs such as loops, conditionals and parallel processes are fully accommodated.

From the testability point of view, all intermediate computations are checked, thus giving a constant monitoring of the health of the system and keeping error latency low. The inversion testing scheme is defined and exploited. All this is offered at comparatively reasonable hardware overhead values.

The work of chapter 6 arms the RTL synthesis outputs with protection for the second one of its constituent parts, that is the controller. Six alternative solutions are configured, analysed, implemented and experimentally evaluated for the controller self-checking problem. The idea of reusing datapath self-checking resources for control path checking is conceived and relevant solutions configured, through the introduction of the Intrinsically Secure control states concept. A comprehensive self-checking component library is produced. Overall, control path self-checking resource insertion is formulated and implemented in a fully automated manner, as an add-on to the MOODS synthesis tool. Together with the material of chapter 5, *combined* datapath and controller self-checking design is thus implemented.

Overall, from the point of view of EDA tool development, this work explored the totally new area of including on-line testability in the design parameters and optimising for it in a 3-dimensional design space. From the point of view of self-checking design, it enabled the realisation of well-studied gate-level techniques in a much higher level of the design flow, increasing their practical significance by including them in realistically-sized designs. Therefore, from both points of view it advances the state-of-the-art and opens up opportunities for further research.

Appendix A

Modified MOODS User's Guide

This appendix briefly presents the practical steps required to implement on-line testable designs using the MOODS command prompt. The appendix assumes some familiarity with the original synthesis system operation. Its intention is to instruct the *experienced* MOODS designer on the new functionality of the modified system. Nevertheless, parts of the original MOODS are briefly repeated when needed for the sake of completeness, while references [126, 127, 128, 105, 8] can be consulted for more background information.

A.1 Setting up and interacting with the tool

Like most electronic CAD tools, MOODS organises its designs into *projects*. Therefore the first set-up task before a synthesis session can start is the definition of a new project, the inclusion and compilation of all required behavioural VHDL source files within it, and the hierarchical assembling of the compiled files within the project in a library structure. Details of how this is done can be found in [126]. Having set-up the synthesis project, the synthesis engine, being the "heart" of the whole process can be invoked using a DOS-prompt command such as the following.

```
(moods home directory)\Bin\Moods example
     -m "(project directory)\experiments.lmf"
     -w experiments
     -mult2shift
     -disable_tforms 38000
     {-use_mux MUX_2}
     {other arguments}
```

The above command assumes that a top-level design called `example` has been compiled and the project name is `experiments`. File `experiments.lmf` contains information on the directory location of the library files used in the project and is made known to the synthesis engine through argument `-m`. Argument `-w experiments` defines the directory where created files are to be written in. Argument `-mult2shift` transforms all multiplications and divisions by powers of 2 to left- or right-shifts respectively, and it is highly recommended as it leads to significant hardware savings. Argument `-disable_tforms` excludes a number of transformations from the overall MOODS set of transformations (§3.2.3). The number `38000` is interpreted as a binary bitmap, dictating which transformations will be excluded. The particular number suggests that all register sharing transformations are disabled. Excluding these transformations is highly recommended for the purposes of this work, since it was found that the said transformations are rather experimental in the current version of MOODS and using them only lengthens the simulated annealing algorithm run-time (note that register sharing transformations were not included in the presentation of §3.2.3, for the same reasons). The exact bitmap-to-transformation correspondence for the above number can be found in appendix D of [105]. Argument `-use_mux MUX_2` is recommended when the target technology is an FPGA part. It instructs MOODS to use a particular cell library multiplexer description, that subsequent RTL synthesis tools synthesize using the tristate buffers available within FPGA slices [106]. This leads to better resource usage within the FPGA. However, the argument should not be used when VLSI technology is targeted.

Other arguments exist [126], but exceed the scope of this appendix.

The first task of the system as soon as the above command is issued, is to read the initialisation file, `MOODS.ini`, and be informed about a number of design options. While several pieces of initialisation information can be passed to the tool through this file, the most important information for the purposes of this thesis is the choice of target technology. The target technology becomes known to the tool through a single declaration line in the initialisation file. A typical declaration for this purpose would look like the following.

```
XC4000XV-09 = GenericLibrary, 4000XV.mlib
```

File `4000XV.mlib` is the pre-existing system technology library, targeting Xilinx FPGA parts. This library was augmented to include characterisation information for the newly

added dual-rail checker and fault secure comparator cells (§5.3.3.3). The new technology library file is named `4000XVplus.mlib` and in order to be taken into account the line above should be substituted by

```
XC4000XV-09 = GenericLibrary, 4000XVplus.mlib
```

Furthermore, a new technology library file was written for the Alcatel CMOS VLSI 0.35 technology also used in the experiments of chapters 5 and 6. To use this technology, the following declaration is needed instead of any of the two above.

```
HYA_MTC45000 = GenericLibrary, MTC45000plus.mlib
```

The designer can thus choose his or her target technology of interest by editing file `MOODS.ini`.

A.1.1 Defining the cost function

When MOODS is invoked, the input design has been read and certain preliminary tasks have finished, it presents a command prompt and waits for the designer's instructions. Sensibly the first task is to define synthesis specifications through the cost function. The command that gets MOODS to cost function definition mode is

```
cf
```

Now the designer needs to specify his or her requirements. This is done by "adding" parameters to the (initially empty) cost function vector. For example, adding an area constraint is done by

```
aa
```

The tool asks for the priority value of the area constraint, to which the designer may respond by

```
1
```

or

```
2
```

for first or second priority respectively. The tool then again asks for the target area, to which in this thesis the answer is always

```
0
```

meaning "as cheap as possible" (§3.2.4). Of course, non-zero numbers can be given instead. The delay constraint is declared to the tool similarly, by using command

```
ad
```

instead of `aa`. Again, in this thesis the target delay is always 0. On-line testability has been configured to work similarly. Indeed, the cost function command

```
at
```

includes on-line testability in the set of constraints. The on-line testability priority in all experiments of chapters 5 and 6 has always been 1, while the target value for on-line testability has always been 100 (for 100%, §5.3.3.1).

The choice of control path self-checking scheme is done in the cost function definition as well. Six alternative independent commands have been implemented for this. Command

```
a1
```

instructs the tool to append the CTRL_1 (§6.2.1) self-checking scheme to the controller. Alternatively, `a2`, `a3`, `a4`, `a5` or `a6` can be used, to order CTRL_2 (§6.2.2), CTRL_3 (§6.2.4), CTRL_4 (§6.2.5), CTRL_5 (§6.3.2) or CTRL_6 (§6.3.3) respectively. No further information is required by the tool with respect to controller self-checking, other than choice of scheme. If none of the above six commands is issued, the tool by default assumes that controller self-checking is *not* desired.

When all of the cost function parameters have been set up, command

```
f
```

finishes the cost function definition session and returns to the main MOODS prompt.

A.1.2 Manual application of the testing transformations

After leaving cost function set-up mode and returning to the main prompt, the user can start applying transformations to the design under optimisation. These include the generic transformations of §3.2.3 or the additional testing ones of §5.3.2.1 and §5.3.3.4. The manual application of transformations proceeds as follows. Initially the "select transformation" command is given

```
st
```

The designer is presented with a list of available transformations, *including* the five testing ones added in this thesis. Selection is made by entering the appropriate number, e.g.

```
8
```

for TF8. Assuming familiarity with the original transformations, let us focus on the testing ones. Selecting TF22, TF23, TF25 or TF26 will cause MOODS to prompt for a single instruction characteristic number. In the case of TF22 (physical duplication) or TF23 (physical inversion), this will be the number of the instruction to which the designer desires to attach self-checking resources. In the case of TF26 (remove testing scheme), it will be the instruction whose testing scheme is to be removed. Finally, if TF25 (restore original test response register) is the transformation at hand, then the instruction will be the one for which the self-checking comparison output signal is desired to be unshared. Selecting TF24 (share test response register) will prompt for two instructions. The second will be the one for which the test response register is desired to be abandoned and the response directed to the respective register of the first.

If the selected transformation passes the validity tests of §5.3.2.1 and §5.3.3.4, then the tool will automatically estimate its effect and present the result on screen. Issuing the "perform" command

```
p
```

will subsequently actually perform the transformation.

The semi-automatic insertion of self-checking resources in the experiments of §5.3.2.2 was carried out using several repetitions of the above procedure for transformations TF22 and TF23.

A.1.3 Application of the automatic algorithms

Applying the automatic optimisation algorithms (simulated annealing, heuristics) of §3.2.5 proceeds exactly as in the original MOODS. Hence, the annealing initialisation command

```
ai
```

causes the tool to ask for four arithmetic values : initial temperature, ultimate temperature, temperature decrease factor and number of transformations per optimisation step. Annealing execution command

```
ao
```

sets off the simulated annealing optimisation algorithm with the parameters given by the designer in the initialisation step. If on-line testability has been given as a designer specification during the cost function set-up phase (§A.1.1), then transformations TF22 and

TF23 are included in the set of transformations and the simulated annealing algorithm takes its modified form described in §5.3.3.2.

The tailored heuristic algorithm is set off by the following command.

```
aoh
```

and takes any of the forms of Figure 3.10, depending on the relative values of designer area and delay priorities.

A.1.4 Experimenting with Intrinsically Secure states

This section explains how to apply the ideas of facilitating Intrinsically Secure (IS) states of §6.4.3 within the modified MOODS. Two new MOODS commands are implemented. The "extract IS states" command

```
ei
```

implements step 2 of §6.4.3, that is, it directs the system to traverse all control states, identify those that have all three parts (functional, redundant, and comparison operations) of self-checking schemes scheduled at them, and extract the redundant operation, by applying transformation TF21 to it.

The "merge IS states" command

```
mi
```

likewise implements step 4 of §6.4.3. Again it traverses all control states, identifies those that have a fault secure comparison operation scheduled at them, and applies transformation TF8 (merge fork and successor) to them.

The "Version 2" realisations explained and presented in §6.4.4 were produced as follows.

- Step 1 : optimise using simulated annealing (`ai`, `ao`)
- Step 2 : apply `ei` (effectively bringing self-checking schemes to the state of Figure 6.26d)
- Step 3 : repeat Step 2 until there is no self-checking scheme at the state of Figure 6.26b
- Step 4 : apply `mi` (creating configurations such as the desired state of Figure 6.26c)
- Step 5 : repeat Step 4 until there is no self-checking scheme at the state of Figure 6.26b
- Step 6 : apply tailored heuristics (`aoh`)

When all desired optimisation has finished, the "finish optimisation" command

```
fi
```

terminates the synthesis engine.

A.1.5 Deliberately separating instructions

This final subsection briefly describes how two instructions can be forcibly separated in two different control steps if the designer wishes so. The presented feature exists in the original MOODS tool, and it is used in this thesis to prevent chaining in the manual experiments of §5.3.1.

Consider two consecutive VHDL operations, for example the following two, taken directly from the examples of §5.3.1.

```
v8i := v3i + v5i;
sc1 := v8i - v5i;
```

Assume that it is desirable to forcibly prevent the chaining of the two instructions. The most explicit way to do that is by directly disallowing the synthesis engine to schedule them in the same control step, by using a VHDL `wait for` statement directly in the source code, as in the following.

```
v8i := v3i + v5i;
wait for 10ns;
sc1 := v8i - v5i;
```

Any non-zero delay value (e.g. 10ns as above) will cause the synthesis tool to *always* schedule the subtraction and all instructions below it, *at least* one control step after the addition and all instructions above it. In essence, the wait statement acts as a "barrier" preventing the control step below it from merging with any of the control steps above.

Appendix B

Benchmarks

This appendix provides the behavioural VHDL codes for five benchmark designs used in this thesis for the experimentation of chapters 5 and 6.

B.1 Tseng

The Tseng datapath was introduced in [121] and it is very often the first benchmark used for evaluation purposes in the field of behavioural synthesis. That is because it is considered to be representatives of situations often encountered in the synthesis of real designs. The VHDL code used in this thesis is as follows.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bench is
  port(v3, v5, v7, v8, v9, v11, v14, v15 : out unsigned(15
downto 0));
end bench;

architecture bench_beh of bench is
  signal  v1i, v2i, v3i, v4i, v5i, v7i, v8i, v9i, v10i,
v11i, v12i, v13i, v14i, v15i : unsigned(15 downto 0);
begin
```

```
main_proc:process

begin

  v1i <="1100011111011000";
  v2i <="0000101100111001";
  v3i <="1110101100110001";
  v4i <="0001000111101001";
  v5i <="0101011111001110";
  v7i <="1100110111111001";
  v8i <="0100111101000001";
  v9i <="0011101010010001";
  v10i <="1100100000011000";
  v11i <="1100100011111001";
  v12i <="1100101000000101";
  v13i <="1010110011010001";
  v14i <="1001111111100111";
  v15i <="1110000111111101";

  loop
  v3i <= v1i + v2i;
-- can be commented out
  wait for 2 ns;

  v5i <= v3i - v4i;
-- can be commented out
  wait for 2 ns;

  v8i <= v3i + v5i;
-- can be commented out
  wait for 2 ns;

  v14i <= unsigned(std_logic_vector(v11i) and
std_logic_vector(v8i));
  v1i <= v14i;
```

```
    v12i <= v1i;
    v7i  := v3i * 4;
-- can be commented out
    wait for 2 ns;

    v9i  <= v1i + v7i;
-- can be commented out
    wait for 2 ns;

    v15i <= unsigned(std_logic_vector(v12i) or
std_logic_vector(v9i));
    v2i <= v15i;

    v13i <= v3i;
    v11i := v10i / 2;

-- can be commented out
wait for 2 ns;
end loop;
end process;

process
  begin
    v3 <= v3i;
    v5 <= v5i;
    v7 <= v7i;
    v8 <= v8i;
    v9 <= v9i;
    v11 <= v11i;
    v14 <= v14i;
    v15 <= v15i;
  end process;

end bench_beh;
```

Signals `v1i` – `v15i` are normally primary input ports. However, it was found that many FPGAs tried did not have enough input pins to accommodate 15 16-bit primary inputs. They were therefore converted to internal signals and assigned initial values in the "uncomfortable" way shown in the code. While this is not elegant or efficient coding, it has no negative implications as regards the datapath operations, data dependencies and synthesis tasks that are of primary interest here.

An additional point to note on the behavioural code above are the `wait for 2 ns;` statements found therein. As explained in §A.1.5, these are used to control CS merging and can be removed or commented out at the designer's discretion.

B.2 Differential equation solver

The Diffeq benchmark is a simple differential equation solver, inspired from [129] and slightly modified for synthesis within MOODS. The VHDL code is the following.

```
PACKAGE diffeq_types IS
  SUBTYPE nat is integer range 0 to 65535;
END diffeq_types;

USE work.diffeq_types.all;
entity dif is
   port (Xinport: in nat;
         Yinport: in nat;
         Uinport: in nat;
         Aport  : in nat;
         DXport : in nat;
         Xoutport: out nat;
         Youtport: out nat;
         Uoutport: out nat;
         done: out bit);
end dif;

architecture diffeq of dif is
```

```
signal oldx, oldy, oldu : nat;
signal newx, newy, newu : nat;

begin

MAIN : process
  variable x_var, y_var, u_var, a_var, dx_var: nat;
  variable y1, t1, t2, t3, t4, t5, t6: nat;
  variable looping : bit:='0';
  variable i : nat;
begin
    done<='0';
    if (looping = '0') then
      x_var := Xinport;
      y_var := Yinport;
      u_var := Uinport;
      looping := '1';
    else
      x_var := newx;
      y_var := newy;
      u_var := newu;
    end if;
    a_var := Aport;
    dx_var := DXport;
    if (x_var < a_var) then
-- can be commented out
      wait for 2 ns;
      t1 := u_var * dx_var;
-- can be commented out
      wait for 2 ns;
      t2 := 3 * x_var;
-- can be commented out
      wait for 2 ns;
      t3 := 3 * y_var;
-- can be commented out
```

```
        wait for 2 ns;
        t4 := t1 * t2;
-- can be commented out
        wait for 2 ns;
        t5 := dx_var * t3;
-- can be commented out
        wait for 2 ns;
        t6 := u_var - t4;
-- can be commented out
        wait for 2 ns;
        u_var := t6 - t5;
-- can be commented out
        wait for 2 ns;
        y1 := u_var * dx_var;
-- can be commented out
        wait for 2 ns;
        y_var := y_var + y1;
-- can be commented out
        wait for 2 ns;
        x_var := x_var + dx_var;
        oldx <= x_var;
        oldy <= y_var;
        oldu <= u_var;
    else
        Xoutport <= x_var;
        Youtport <= y_var;
        Uoutport <= u_var;
        looping := '0';
        done<='1';
    end if;
-- can be commented out
    wait for 2 ns;
end process;

SYNCH: process
```

```
  begin
      newx <= oldx;
      newy <= oldy;
      newu <= oldu;
      wait for 2 ns;
  end process;

end diffeq;
```

B.3 QRS

The QRS design is a medical electronics application, also popular as a high-level synthesis benchmark since first used for benchmarking purposes, in [130].

```
PACKAGE qrs_types IS
    SUBTYPE int16 IS integer RANGE  32767 DOWNTO -32768;
--  16 bit integer

    SUBTYPE nat2  IS integer RANGE      3 DOWNTO 0;
--   2 bit unsigned integer
END qrs_types;

USE work.qrs_types.all;

ENTITY qrs IS
PORT (ecg1    : IN     int16;
      low     : IN     int16;
      high    : IN     int16;
      indx    : IN     int16;
      ftm1in  : IN     int16;
      ftm2in  : in     int16;
      ftm1out: buffer    int16;
      ftm2out: buffer    int16;
      new_data : IN     boolean;
```

```
      data_done: out     boolean;
      fl3o    : OUT    nat2;
      RRpeak  : OUT    boolean;
      RRo     : OUT    int16);
END qrs;

USE work.qrs_types.all;

ARCHITECTURE system OF qrs IS
BEGIN
  qrs_proc: PROCESS

    CONSTANT    ACTIVE          : boolean := false;
    CONSTANT    INACTIVE        : boolean := true;

    VARIABLE    ft, ecgm1, ysi     : int16;
    VARIABLE    ymax, xmax, y0, ath, ys, y0m2, zmax, y0m1 :
int16;
    VARIABLE    sth1, sth2, lxmax, lymax, lzmax : int16;
    VARIABLE    count, RR : int16;
    VARIABLE    fl3 : nat2;
    VARIABLE    fl1, fl2 : boolean;
    variable    y2, y4, y8, y16, x2, x4, x8, x16, z2, z4, z8,
z16, lx8, ly8, lz8 : int16;
    variable    ecg_dif, ecg_dif256 : int16;

  begin
    RRpeak <= Inactive;
    fl3o   <= 0;
    RRo    <= 0;
    y0m1   := 0;
    y0m2   := 0;
    ymax   := 0;
    xmax   := 0;
    zmax   := 0;
```

```
    y2:=0;
    y4:=0;
    y8:=0;
    y16:=0;
    x2:=0;
    x4:=0;
    x8:=0;
    x16:=0;
    z2:=0;
    z4:=0;
    z8:=0;
    z16:=0;
    RR      := 0;
    lymax   := 0;
    lzmax   := 0;
    lxmax   := 0;
    lx8:=0;
    ly8:=0;
    lz8:=0;
    fl3     := 0;
    fl1     := False;
    fl2     := False;
    count   := 0;

    ecgm1 := ftm1in;

    init: FOR i IN 1 TO indx LOOP      -- initialization loop

    data_done<=false;
    wait on new_data until new_data;
    ecg_dif:=ecg1-ecgm1;
    ecg_dif256:=ecg_dif/256;
    ft  := ftm1in + ecg_dif - ecg_dif256;
    ysi := ft - ftm2in;
-- can be commented out
```

```
      wait for 2 ns;
      IF (ysi > ymax) THEN
      ymax := ysi;
      y2:=ysi/2;
-- can be commented out
      wait for 2 ns;
      y4:=ysi/4;
-- can be commented out
      wait for 2 ns;
      y8:=ysi/8;
      y16:=ysi/16;
  END IF;
  IF ( ft > xmax) THEN
    xmax := ft;
    x2:=ft/2;
-- can be commented out
    wait for 2 ns;
    x4:=ft/4;
-- can be commented out
    wait for 2 ns;
    x8:=ft/8;
    x16:=ft/16;
  END IF;
  IF (ft > 0) THEN
    y0 := ft;
  else
    y0 := -ft;
  END IF;
     ath := x4;
  IF ( ath > y0) THEN
    y0 := ath;
  END IF;
  ys := y0 - y0m2;
  IF (ys > zmax) THEN
    zmax := ys;
```

```
    z2:=ys/2;
    z4:=ys/4;
-- can be commented out
    wait for 2 ns;
    z8:=ys/8;
    z16:=ys/16;
  END IF;
    ftm2out  <= ftm1in;
    ftm1out  <= ft;
    ecgm1 := ecg1;
    y0m2  := y0m1;
    y0m1  := y0;
    sth1  := y2 + y8 + y16;
    sth2  := z2 + z8 + z16;
   data_done<=true;
   wait for 2 ns;

    END LOOP init;

    regular : LOOP

        IF (ysi > sth1) THEN
         fl1   := true;
         count := 0;
        END IF;
        IF (ys > sth2) THEN
            fl2   := true;
            count := 0;
        END IF;
     IF ((fl1 = true) AND (fl2 = true) AND (RR > low)) THEN
            RRpeak <= Active;
            xmax  := x2 + x4 + x8 + lx8;
            x2:=xmax/2;
-- can be commented out
        wait for 2 ns;
```

```
            x4:=xmax/4;
-- can be commented out
        wait for 2 ns;
            x8:=xmax/8;
            x16:=xmax/16;
            ymax  := y2 + y4 + y8 + ly8;
            y2:=ymax/2;
-- can be commented out
       wait for 2 ns;
            y4:=ymax/4;
-- can be commented out
        wait for 2 ns;
            y8:=ymax/8;
            y16:=ymax/16;
            zmax  := z2 + z4 + z8 + lz8;
            z2:=zmax/2;
-- can be commented out
        wait for 2 ns;
            z4:=zmax/4;
-- can be commented out
        wait for 2 ns;
            z8:=zmax/8;
            z16:=zmax/16;
            RR    := 0;
            count := 0;
            fl1   := false;
            fl2   := false;
            fl3   := 0;
            lxmax := 0;
            lx8:=0;
            lymax := 0;
            ly8:=0;
            lzmax := 0;
            lz8:=0;
       ELSE
```

```
        RRpeak <= Inactive;
    END IF;

    IF ((fl1 = true) OR  (fl2 = true)) THEN
        count := count + 1;
    END IF;

    fl3o  <= fl3;
    RRo   <= RR;

    data_done<=false;
    wait on new_data until new_data;
    ecg_dif:=ecg1-ecgm1;
    ecg_dif256:=ecg_dif/256;

    ft  := ftm1out + ecg_dif - ecg_dif256;
    ysi := ft - ftm2out;
    IF (ysi > lymax) THEN
      lymax := ysi;
      ly8:=ysi/8;
    END IF;
    IF ( ft > lxmax) THEN
      lxmax := ft;
      lx8:=ft/8;
    END IF;
    IF (ft > 0) THEN
      y0 := ft;
    else
      y0 := -ft;
    END IF;
    ath := x4;
    IF (y0 < ath) THEN
      y0 := ath;
    END IF;
    ys := y0 - y0m2;
```

```
        IF (ys > lzmax) THEN
          lzmax := ys;
          lz8:=ys/8;
        END IF;
        IF (count = 8) THEN
            fl1   := false;
            fl2   := false;
            count := 0;
        END IF;
        IF (RR > high) THEN
            fl3  := fl3 + 1;
            RR   := 0;
            ymax := y2;
-- can be commented out
        wait for 2 ns;
            y2:=ymax/2;
            y4:=ymax/4;
            y8:=ymax/8;
            y16:=ymax/16;
            zmax := z2;
-- can be commented out
        wait for 2 ns;
            z2:=zmax/2;
            z4:=zmax/4;
            z8:=zmax/8;
            z16:=zmax/16;
        END IF;
        sth1 := y2 + y8 + y16;
        sth2 := z2 + z8 + z16;
        RR   := RR + 1;
        ecgm1 := ecg1;
        y0m2  := y0m1;
        y0m1  := y0;
        ftm2out  <= ftm1out;
        ftm1out  <= ft;
```

```
        data_done<=true;
        wait for 2 ns;

    END LOOP regular;

  END PROCESS qrs_proc;

END system;
```

Clearly it is a sizeable design. Hence it was claimed in chapter 5 that it is particularly encouraging that modified MOODS was able to cope with it.

B.4 Viterbi decoder

The 8-bit Viterbi decoder recently presented in [131] and used as a benchmark in chapters 5 and 6, is shown in the following. It can be observed that it is composed of 8 almost identical concurrent processes. There is also a 32-bit version (comprising 32 processes) used in one experiment (Table 5.41), not shown here for brevity.

```
package pack_Viterbi is
type four_bit_array is array (0 to 7) of integer range 0 to
6;
type array_of_bit_vector is array (0 to 3) of bit_vector(0
to 7);
type two_bit_integer_array is array (0 to 1) of integer
range 0 to 6;

  procedure vector_multi0(entri:in bit; wpa, wpb: in integer
range 0 to 6; path0, path1: in bit_vector(0 to 7);
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6);

  procedure vector_multi1(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
```

```
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6);

  procedure vector_multi2(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6);

  procedure vector_multi3(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6);

  procedure vector_multi4(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6);

  procedure vector_multi5(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6);

  procedure vector_multi6(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6);

  procedure vector_multi7(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6);

end pack_Viterbi;
```

```
package body pack_Viterbi is

procedure vector_multi0(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
-- moods inline
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6) is
   variable weight_vector: integer range 0 to 6;
       begin
   if (entri='0') then -- this only makes a one clock cycle
difference
  weight_vector:=wpb+1; --doing this reduced one clock cycle
  if weight_vector <= wpa then
     if (weight_vector < 2) then
        wp1:=weight_vector;
        pathx:=path1(1 to 7)&'1';
     else
        wp1:=weight_vector;
     end if;
  else
     if (wpa < 2) then
        wp1:=wpa;
        pathx:=path0(1 to 7)&'0';
     else
        wp1:=wpa;
     end if;
  end if;
else
  weight_vector:=wpa+1;
  if weight_vector <= wpb then
     if (weight_vector < 2) then
        wp1:=weight_vector;
        pathx:=path0(1 to 7)&'0';
     else
        wp1:=weight_vector;
```

```
      end if;
   else
      if (wpb < 2) then
        wp1:=wpb;
        pathx:=path1(1 to 7)&'1';
      else
        wp1:=wpb;
      end if;
   end if;
end if;

end vector_multi0;

procedure vector_multi1(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
-- moods inline
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6) is
   variable weight_vector: integer range 0 to 6;
       begin
   if (entri='0') then
  weight_vector:=wpb+1;
  if weight_vector <= wpa then
     if (weight_vector < 2) then
        wp1:=weight_vector;
        pathx:=path1(1 to 7)&'1';
     else
        wp1:=weight_vector;
     end if;
  else
     if (wpa < 2) then
        wp1:=wpa;
        pathx:=path0(1 to 7)&'0';
     else
        wp1:=wpa;
```

```
      end if;
   end if;
else
   weight_vector:=wpa+1;
   if weight_vector <= wpb then
      if (weight_vector < 2) then
         wp1:=weight_vector;
         pathx:=path0(1 to 7)&'0';
      else
         wp1:=weight_vector;
      end if;
   else
      if (wpb < 2) then
        wp1:=wpb;
        pathx:=path1(1 to 7)&'1';
      else
        wp1:=wpb;
      end if;
   end if;
end if;

end vector_multi1;

procedure vector_multi2(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
-- moods inline
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6) is
   variable weight_vector: integer range 0 to 6;
       begin
   if (entri='0') then
  weight_vector:=wpb+1;
  if weight_vector <= wpa then
     if (weight_vector < 2) then
        wp1:=weight_vector;
```

```
            pathx:=path1(1 to 7)&'1';
        else
            wp1:=weight_vector;
        end if;
    else
        if (wpa < 2) then
            wp1:=wpa;
            pathx:=path0(1 to 7)&'0';
        else
            wp1:=wpa;
        end if;
    end if;
else
    weight_vector:=wpa+1;
    if weight_vector <= wpb then
        if (weight_vector < 2) then
            wp1:=weight_vector;
            pathx:=path0(1 to 7)&'0';
        else
            wp1:=weight_vector;
        end if;
    else
        if (wpb < 2) then
          wp1:=wpb;
          pathx:=path1(1 to 7)&'1';
        else
          wp1:=wpb;
        end if;
    end if;
end if;

end vector_multi2;

procedure vector_multi3(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
```

```
-- moods inline
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6) is
   variable weight_vector: integer range 0 to 6;
       begin
   if (entri='0') then
  weight_vector:=wpb+1;
  if weight_vector <= wpa then
     if (weight_vector < 2) then
        wp1:=weight_vector;
        pathx:=path1(1 to 7)&'1';
     else
        wp1:=weight_vector;
     end if;
  else
     if (wpa < 2) then
        wp1:=wpa;
        pathx:=path0(1 to 7)&'0';
     else
        wp1:=wpa;
      end if;
  end if;
else
  weight_vector:=wpa+1;
  if weight_vector <= wpb then
     if (weight_vector < 2) then
        wp1:=weight_vector;
        pathx:=path0(1 to 7)&'0';
     else
        wp1:=weight_vector;
     end if;
  else
     if (wpb < 2) then
       wp1:=wpb;
       pathx:=path1(1 to 7)&'1';
```

```
      else
        wp1:=wpb;
      end if;
   end if;
end if;

end vector_multi3;

procedure vector_multi4(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
-- moods inline
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6) is
   variable weight_vector: integer range 0 to 6;
       begin
   if (entri='0') then
  weight_vector:=wpb+1;
  if weight_vector <= wpa then
     if (weight_vector < 2) then
        wp1:=weight_vector;
        pathx:=path1(1 to 7)&'1';
     else
        wp1:=weight_vector;
     end if;
  else
     if (wpa < 2) then
        wp1:=wpa;
        pathx:=path0(1 to 7)&'0';
     else
        wp1:=wpa;
     end if;
  end if;
else
  weight_vector:=wpa+1;
  if weight_vector <= wpb then
```

```
      if (weight_vector < 2) then
         wp1:=weight_vector;
         pathx:=path0(1 to 7)&'0';
      else
         wp1:=weight_vector;
      end if;
   else
      if (wpb < 2) then
        wp1:=wpb;
        pathx:=path1(1 to 7)&'1';
      else
        wp1:=wpb;
      end if;
   end if;
end if;

end vector_multi4;

procedure vector_multi5(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
-- moods inline
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6) is
   variable weight_vector: integer range 0 to 6;
       begin
   if (entri='0') then
  weight_vector:=wpb+1;
  if weight_vector <= wpa then
      if (weight_vector < 2) then
         wp1:=weight_vector;
         pathx:=path1(1 to 7)&'1';
      else
         wp1:=weight_vector;
      end if;
  else
```

```
      if (wpa < 2) then
         wp1:=wpa;
         pathx:=path0(1 to 7)&'0';
      else
         wp1:=wpa;
      end if;
   end if;
else
   weight_vector:=wpa+1;
   if weight_vector <= wpb then
      if (weight_vector < 2) then
         wp1:=weight_vector;
         pathx:=path0(1 to 7)&'0';
      else
         wp1:=weight_vector;
      end if;
   else
      if (wpb < 2) then
        wp1:=wpb;
        pathx:=path1(1 to 7)&'1';
      else
        wp1:=wpb;
      end if;
   end if;
end if;

end vector_multi5;

procedure vector_multi6(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
-- moods inline
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6) is
   variable weight_vector: integer range 0 to 6;
       begin
```

```
    if (entri='0') then
   weight_vector:=wpb+1;
   if weight_vector <= wpa then
      if (weight_vector < 2) then
         wp1:=weight_vector;
         pathx:=path1(1 to 7)&'1';
      else
         wp1:=weight_vector;
      end if;
   else
      if (wpa < 2) then
         wp1:=wpa;
         pathx:=path0(1 to 7)&'0';
      else
         wp1:=wpa;
      end if;
   end if;
else
   weight_vector:=wpa+1;
   if weight_vector <= wpb then
      if (weight_vector < 2) then
         wp1:=weight_vector;
         pathx:=path0(1 to 7)&'0';
      else
         wp1:=weight_vector;
      end if;
   else
      if (wpb < 2) then
        wp1:=wpb;
        pathx:=path1(1 to 7)&'1';
      else
        wp1:=wpb;
      end if;
   end if;
end if;
```

```
end vector_multi6;

procedure vector_multi7(entri:in bit; wpa,wpb:in integer
range 0 to 6; path0,path1: in bit_vector(0 to 7);
-- moods inline
   pathx : out bit_vector(0 to 7); wp1:out integer range 0
to 6) is
   variable weight_vector: integer range 0 to 6;
       begin
   if (entri='0') then
  weight_vector:=wpb+1;
  if weight_vector <= wpa then
     if (weight_vector < 2) then
        wp1:=weight_vector;
        pathx:=path1(1 to 7)&'1';
     else
        wp1:=weight_vector;
     end if;
  else
     if (wpa < 2) then
        wp1:=wpa;
        pathx:=path0(1 to 7)&'0';
     else
        wp1:=wpa;
     end if;
  end if;
else
  weight_vector:=wpa+1;
  if weight_vector <= wpb then
     if (weight_vector < 2) then
        wp1:=weight_vector;
        pathx:=path0(1 to 7)&'0';
     else
        wp1:=weight_vector;
```

```
      end if;
   else
      if (wpb < 2) then
        wp1:=wpb;
        pathx:=path1(1 to 7)&'1';
      else
        wp1:=wpb;
      end if;
   end if;
end if;

end vector_multi7;

end pack_Viterbi;

use work.pack_Viterbi.ALL;

entity ent_Viterbi is
PORT ( entry: in bit;
       exitx0, exitx1, exitx2, exitx3, exitx4, exitx5,
exitx6, exitx7: out bit_vector(0 to 7));
end ent_Viterbi;

architecture arch_Viterbi of ent_Viterbi is
signal w0:four_bit_array:=(0,3,3,3,3,3,3,3);
signal path0,path1,path2,path3,path4,path5,path6,path7:
bit_vector(0 to 7);

begin

process
begin
exitx0<=path0;
exitx1<=path1;
exitx2<=path2;
```

```
exitx3<=path3;
exitx4<=path4;
exitx5<=path5;
exitx6<=path6;
exitx7<=path7;
wait for 1 ns;
end process;

entry0:process
variable wx: integer range 0 to 6;
variable pathx: bit_vector(0 to 7);
begin
wait for 1 ns;
vector_multi0(entry,w0(0),w0(4),path0,path4,pathx,wx);
path0<=pathx;
w0(0)<=wx;
end process entry0;

entry1:process
variable wx: integer range 0 to 6;
variable pathx: bit_vector(0 to 7);
begin
wait for 1 ns;
vector_multi1(entry,w0(5),w0(1),path5,path1,pathx,wx);
path1<=pathx;
w0(1)<=wx;
end process entry1;

entry2:process
variable wx: integer range 0 to 6;
variable pathx: bit_vector(0 to 7);
begin
wait for 1 ns;
vector_multi2(entry,w0(1),w0(5),path1,path5,pathx,wx);
path2<=pathx;
```

```
w0(2)<=wx;
end process entry2;

entry3:process
variable wx: integer range 0 to 6;
variable pathx: bit_vector(0 to 7);
begin
wait for 1 ns;
vector_multi3(entry,w0(4),w0(0),path4,path0,pathx,wx);
path3<=pathx;
w0(3)<=wx;
end process entry3;

entry4:process
variable wx: integer range 0 to 6;
variable pathx: bit_vector(0 to 7);
begin
wait for 1 ns;
vector_multi4(entry,w0(2),w0(6),path2,path6,pathx,wx);
path4<=pathx;
w0(4)<=wx;
end process entry4;

entry5:process
variable wx: integer range 0 to 6;
variable pathx: bit_vector(0 to 7);
begin
wait for 1 ns;
vector_multi5(entry,w0(7),w0(3),path7,path3,pathx,wx);
path5<=pathx;
w0(5)<=wx;
end process entry5;

entry6:process
variable wx: integer range 0 to 6;
```

```
variable pathx: bit_vector(0 to 7);
begin
wait for 1 ns;
vector_multi6(entry,w0(3),w0(7),path3,path7,pathx,wx);
path6<=pathx;
w0(6)<=wx;
end process entry6;

entry7:process
variable wx: integer range 0 to 6;
variable pathx: bit_vector(0 to 7);
begin
wait for 1 ns;
vector_multi7(entry,w0(6),w0(2),path6,path2,pathx,wx);
path7<=pathx;
w0(7)<=wx;
end process entry7;

end arch_Viterbi;
```

B.5 Greater Common Divider

The greater common divider (GCD) benchmark is the last design listed here.

```
entity GCD is
 port (X, Y           : in integer range 0 to 65535;
       gcd_output     : out integer range 0 to 65535);
end GCD;

architecture behavioural of GCD is
begin
BIGLOOP: process
variable xvar, yvar :  integer range 0 to 65535;
begin
```

```
        wait for 20 ns;
        xvar := X;
        yvar := Y;
        if ((xvar = 0) or (yvar = 0)) then
            gcd_output <= 0;
            xvar := 0;
            yvar := 0;
        else
          COMP: loop
            wait for 20 ns;
            if (xvar < yvar) then
                yvar := yvar - xvar;
            else
                if (xvar > yvar) then
                    xvar := xvar - yvar;
                end if;
            end if;
            exit COMP when (xvar = yvar);
          end loop COMP;
          gcd_output <= xvar;
        end if;
        wait on x,y;
end process;

end behavioural;
```

Appendix C

List of papers

The research work in this thesis was presented and published in the official proceedings of rigorously refereed conferences through the following research papers :

- P. Oikonomakos and M. Zwolinski, "Transformation Based Insertion of On-Line Testing Resources in a High-Level Synthesis Environment", 8th IEEE International On-Line Testing Workshop (IOLTW02), Isle of Bendor, France 2002, p. 185 [poster presentation, paper available at http://www.cl.cam.ac.uk/~po230/ioltw02.pdf]
- P. Oikonomakos, M. Zwolinski and B. M. Al-Hashimi, "*Versatile High-level Synthesis of Self-checking Datapaths Using an On-line Testability Metric*", Design Automation and Test in Europe Conference and Exhibition (DATE03), Munich, Germany 2003, pp 596-601 [oral presentation, paper available at http://www.cl.cam.ac.uk/~po230/date03.pdf]
- P. Oikonomakos and M. Zwolinski, "*Foundation of Combined Datapath and Controller Self-checking Design*", 9th IEEE International On-Line Testing Symposium (IOLTS03), Kos Island, Greece 2003, pp 30-34 [oral presentation, paper available at http://www.cl.cam.ac.uk/~po230/iolts03.pdf]

The following informal presentations were also given :

- P. Oikonomakos and M. Zwolinski, *"Using High-Level Synthesis to Implement On-Line Testability"*, IEEE/IEE Real-Time Embedded Systems Workshop (RTES01), London, UK 2001
- P. Oikonomakos and M. Zwolinski, "*High-Level Synthesis for On-Line Testability*", Postgraduate Research in Electronics, Photonics, communications and software (PREP02), Nottingham, UK 2002

- P. Oikonomakos, "*Implementing On-Line Testable Designs in Behavioural Synthesis*", 5th SIGDA PhD Forum at the Design Automation Conference (DAC02), New Orleans, USA 2002
- P. Oikonomakos and M. Zwolinski, "*On-Line Testability in a Transformation-Based and Cost Function-Driven High-Level Synthesis Environment*", 2nd UK ACM SIGDA Workshop on Electronic Design Automation (UKSIGDA02), Bournemouth, UK 2002
- P. Oikonomakos and M. Zwolinski, "*Controller Self-checking in a Controller / Datapath Architecture*", 3rd UK ACM SIGDA Workshop on Electronic Design Automation (UKSIGDA03), Southampton, UK 2003

References

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

2. P.K. Lala, "Fault tolerant & Fault testable hardware design", Prentice Hall 1985.

3. G.E. Moore, "Cramming more components onto integrated circuits", Electronics, Vol. 38, No. 8, April 1965.
(available at ftp://download.intel.com/research/silicon/moorespaper.pdf)

4. International Technology Roadmap for Semiconductors (ITRS), 2003 edition.
(available at http://public.itrs.net/Files/2003ITRS/Home2003.htm)

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

6. A.K. Nieuwland, R.P. Kleihorst, "The Positive Effect on IC Yield of Embedded Fault Tolerance for SEUs", IEEE International On-line Testing Symposium, 2003, p. 75-79.

7. J.J.A. Fournier, S. Moore, H. Li, R. Mullins, G. Taylor, "Security Evaluation of Asynchronous Circuits", International Workshop on Cryptographic Hardware and Embedded Systems, 2003, p. 137-151 (LNCS 2779).

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

9. 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.

10. D.A. Anderson, G. Metze, "Design of Totally Self-Checking Check Circuits for m-out-of-n Codes", IEEE Transactions on Computers, Vol. 22, No. 3, March 1973, p. 263-269.

11. S. Tarnick, "Controllable Self-Checking Checkers for Conditional Concurrent Checking", IEEE Transactions on CAD, Vol. 14, No. 5, May 1995, p. 547-553.

12. S. Tarnick, "Embedded Parity and Two-Rail TSC Checkers with Error-Memorizing Capability", VLSI Design, Vol. 5, No. 4, 1998, p. 347-356.

13. J.E. Smith, G. Metze "Strongly Fault Secure Logic Networks", IEEE Transactions on Computers, Vol. 27, No. 6, June 1978, p. 491-499.

14. S.J. Piestrak, "Self-checking design in Eastern Europe", IEEE Design & Test of Computers, Vol. 13, No. 1, Spring 1996, p.16-25.

15. M. Nicolaidis, R.O. Duarte, S. Manich, J. Figueras, "Fault-Secure Parity Prediction Arithmetic Operators", IEEE Design & Test of Computers, Vol. 14, No. 2, April-June 1997, p. 60-71.

16. M. Nicolaidis, R.O. Duarte, "Design of Fault-Secure Parity-Prediction Booth Multipliers", Design Automation and Test in Europe, 1998, p. 7-14.

17. J. Khakbaz, E.J. McCluskey, "Self-Testing Embedded Parity Checkers", IEEE Transactions on Computers, Vol. 33, No. 8, August 1984, p. 753-756.

18. D. Nikolos, "Optimal Self-Testing Embedded Parity Checkers", IEEE Transactions on Computers, Vol. 47, No. 3, March 1998, p. 313-321.

19. S. Tarnick, "Embedded Parity and Two-Rail TSC Checkers with Error Memorizing Capability", IEEE International On-line Testing Workshop, 1995, p. 221-225.

20. S. Tarnick, "Embedded Parity and Two-Rail TSC Checkers with Error Memorizing Capability", University of Potsdam Technical Report MPI-I-94-606, 1994. (available at http://www.ift.cs.uni-potsdam.de/agfr/english/reports.html/MPI-I-94-606.ps.gz)

21. M.Y. Hsiao, A.M. Patel, D.K. Pradhan, "Store Address Generator with On-Line Fault-Detection Capability", IEEE Transactions on Computers, Vol. 26, No. 11, November 1977, p. 1144-1151.

22. C. Zeng, N. Saxena, E.J. McCluskey, "Finite State Machine Synthesis with Concurrent Error Detection", IEEE International Test Conference, 1999, p. 672-679.

23. G. Lakshminarayana, A. Raghunathan, N.K. Jha "Behavioral Synthesis of Fault Secure Controller/Datapaths Based on Aliasing Probability Analysis", IEEE Transactions on Computers, Vol. 49, No. 9, September 2000, p. 865-885.

24. A.M. Paschalis, D. Nikolos, C. Halatsis, "Efficient Modular Design of TSC Checkers for M-out-of-2M-Codes", IEEE Transactions on Computers, Vol. 37, No. 3, March 1988, p. 301-309.

25. W.F. Chang, C.W. Wu, "Low-cost Modular Totally Self-checking Checker Design for m-out-of-n Code", IEEE Transactions on Computers, Vol. 48, No. 8, August 1999, p. 815-826.

26. V.V. Dimakopoulos, G. Sourtziotis, A. Paschalis, D. Nikolos, "On TSC Checkers for m-out-of-n Codes", IEEE Transactions on Computers, Vol. 44, No. 8, August 1995, p. 1055-1059.

27. S.J. Piestrak, "Design of Self-Testing Checkers for m-out-of-n Codes Using Parallel Counters", Journal of Electronic Testing – Theory and Applications, Vol. 12, No. 1-2, February-April 1998, p. 63-68.

28. X. Kavousianos, D. Nikolos, G. Sidiropoulos, "Design of Compact and High speed, Totally Self Checking CMOS Checkers for m-out-of-n Codes", IEEE International Symposium on Defect and Fault Tolerance in VLSI System, 1997, p. 128-136.

29. J. Khakbaz, "Totally Self-Checking Checker for 1-out-of-n Code Using Two-Rail Codes", IEEE Transactions on Computers, Vol. 31, No. 7, July 1982, p. 677-681.

30. D.L. Tao, C.R.P. Hartmann, P.K. Lala, "A General Technique for Designing Totally Self-Checking Checker for 1-out-of-N Code with Minimum Gate Delay", IEEE Transactions on Computers, Vol. 41, No. 7, July 1992, p. 881-886.

31. A.M. Paschalis, C. Efstathiou, C. Halatsis, "An Efficient TSC 1-out-of-3 Code Checker", IEEE Transactions on Computers, Vol. 39, No. 3, March 1990, p. 407-411.

32. J.-C. Lo, S. Thanawastien, "On the Design of Combinational Totally Self-Checking 1-out-of-3 Code Checkers", IEEE Transactions on Computers, Vol. 39, No.3, March 1990, p. 387-393.

33. C. Metra, M. Favalli, B. Ricco, "Novel 1-out-of-n CMOS checker", IEE Electronics Letters, Vol. 30, No. 17, August 18, 1994, p. 1398--1400.

34. A.P. Stroele, S. Tarnick, "Programmable embedded self-testing checkers for all-unidirectional error-detecting codes", IEEE VLSI Test Symposium, 1999, p. 361-369.

35. W.-F. Chang, C.-W. Wu, "TSC Berger-code Checker Design for 2^{r-1}-Bit Information", Journal of Information Science and Engineering, Vol. 15, No. 3, 1999, p. 429-441.

36. M. Lobetti-Bodoni, A. Pricco, A. Benso, S. Chiusano, P. Prinetto, "An on-line BISTed SRAM IP core", IEEE International Test Conference, 1999, p. 993-1000.

37. R. Leveugle, "Automatic Modifications of High Level VHDL Descriptions for Fault Detection or Tolerance", Design Automation and Test in Europe (DATE) 2002, p. 837 - 841.

38. C. Bolchini, R. Montandon, F. Salice, D. Sciuto, "Design of VHDL-based totally self-checking finite-state machine and data-path descriptions", IEEE Transactions on VLSI, Vol. 8, No. 1, February 2000, p. 98-103.

39. C. Bolchini, R. Montandon, F. Salice, D. Sciuto, "Self-checking FSMs based on a constant distance state encoding", IEEE International Symposium on Defect and Fault Tolerance in VLSI System, 1995, p. 269-277.

40. S. Tarnick, A.P. Stroele, "Embedded Self-testing checkers for low-cost arithmetic codes", IEEE International Test Conference, 1998, p. 514-523.

41. I. Alzaher Noufal, M. Nicolaidis, "A CAD framework for generating self-checking multipliers based on residue codes", Design Automation and Test in Europe (DATE), 1999, p. 122-131.

42. M. Nicolaidis, "Self-exercising checkers for Unified Built-In Self-Test (UBIST)", IEEE Transactions on CAD, Vol. 8, No. 3, March 1989, p. 203-218.

43. X. Sun, M. Serra, "On-line and off-line testing with shared resources : a new BIST approach", IEEE Transactions on CAD, Vol. 16, No. 9, September 1997, p. 1045-1056.

44. M. Goessel, E.S. Sogomonyan, "A parity-preserving multi-input signature analyser and its application for concurrent checking and BIST", Journal of Electronic Testing – Theory and Applications, Vol. 8, No. 2, April 1996, p. 165-177.

45. E.S. Sogomonyan, M. Goessel, "A New Parity-Preserving Multi-Input Signature Analyzer", IEEE International On-line Testing Workshop, 1995, p. 211-215.

46. A. Hlawiczka, M. Gossel, E. Sogomonyan, "A Linear Code-Preserving Signature Analyser COPMISR", IEEE VLSI Test Symposium, 1997, p. 350-355.

47. C. Stroud, M. Ding, S. Seshadri, I. Kim, S. Roy, S. Wu, R. Karri, "A Parametrized VHDL Library for On-line Testing", International Test Conference, 1997, p. 479-488.

48. J. Bhasker, "A VHDL Primer", Prentice Hall 1999.

49. M. Zwolinski, "Digital System Design with VHDL", Prentice Hall 2000.

50. M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies", IEEE VLSI Test Symposium, 1999, p. 86-94.

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

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

53. S. Mitra and E.J. McCluskey, "Combinational Logic Synthesis for Diversity in Duplex Systems", IEEE International Test Conference, 2000, p. 179-188.

54. M. Saeed, D. Thulborn, J. Yeandel, S. Jones, "IFIS – An On-line Testing Methodology Using Dual-Rail Data Coding", IEEE International On-Line Testing Workshop, 1996, p. 68-71.

55. J. Yeandel, D. Thulborn, S. Jones, "An on-line testable UART implemented using IFIS", IEEE VLSI Test Symposium, 1997, p. 344-349.

56. J. Yeandel, D. Thulborn, S. Jones, "IFIS : an online test methodology", IEE Proceedings – Circuits, Devices and Systems, Vol. 145, No. 1, February 1998, p. 1-6.

57. J. Yeandel, D. Thulborn, S. Jones, "The design and implementation of an on-line testable UART", Journal of Electronic Testing – Theory and Applications, Vol. 12, No. 3, June 1998, p. 187-198.

58. D. Nikolos, "Self-Testing Embedded Two-Rail Checkers", Journal of Electronic Testing – Theory and Applications, Vol. 12, No. 1-2, February-April 1998, p. 69-79.

59. M. Seuring, M. Goessel, E. Sogomonyan, "A Structural Approach for Space Compaction for Concurrent Checking and BIST", IEEE VLSI Test Symposium, 1998, p. 354-361.

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

61. R. Narasimhan, D.J. Rosenkrantz, S.S. Ravi, "Efficient algorithms for analyzing and synthesizing fault-tolerant datapaths", IEEE International Symposium on Defect and Fault Tolerance in VLSI System, 1995, p. 81-89.

62. 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.

63. S.N. Hamilton, A. Orailoglu, A. Hertwig, "Self Recovering Controller and Datapath Codesign", Design Automation and test in Europe (DATE), 1999, 596-601.

64. R. Karri, B. Iyer, "Introspection : A Register Transfer Level Technique for Concurrent Error Detection and Diagnosis in Data Dominated Designs", ACM Trans-

actions on Design Automation of Electronic Systems, Vol. 6, No. 4, October 2001, p. 501-515.

65. A. Antola, F. Ferrandi, V. Piuri, M. Sami, "Semiconcurrent Error Detection in Data Paths", IEEE Transactions on Computers, Vol. 50, No. 5, May 2001, p. 449-465.

66. K. Wu, R. Karri, "Exploiting Idle cycles for Algorithm Level Re-Computing", Design Automation and Test in Europe (DATE) 2002, p. 842–846.

67. K. Wu, R. Karri, "Register Transfer Level Approach to Hybrid Time and Hardware Redundancy Based Fault Secure Datapath Synthesis", IEEE International Test Conference, 2003, p. 902-911.

68. K. Wu, R. Karri, "Re-computing using Ruptured Dependencies : A Low-cost, Low-latency Register Transfer Level Approach to Fault-Secure Datapaths", IEEE North Atlantic Test Workshop, 2003.

69. K. Wu, R. Karri, "Algorithm level recomputing using allocation diversity: a register transfer level approach to time redundancy-based concurrent error detection", IEEE Transactions on CAD, Vol. 21, Vol. 9, September 2002, p. 1077- 1087.

70. R. Karri, K. Wu, "Algorithm level re-computing using implementation diversity: a register transfer level concurrent error detection technique", IEEE Transactions on VLSI, Vol. 10, No. 6, December 2002, p. 864- 875.

71. K.K. Saluja, R. Sharma, C.R. Kime, "A Concurrent Testing Technique for Digital Circuits", IEEE Transactions on CAD, Vol. 7, No. 12, December 1988, p. 1250-1259.

72. J.M.V. Santos, "Concurrent Scan Monitoring and Multi-Pattern Search", IEEE International On-line Testing Workshop, 2000, p.107-111.

73. A.D. Brown, K.R. Baker, A.J.C. Williams, "On-line testing of statically and dynamically scheduled synthesized systems", IEEE Transactions on CAD, Vol. 16, No. 1, January 1997, p. 47-57.

74. K.R. Baker, M. Zwolinski, A.D. Brown, "Concurrent Testing of Latent Modules in Synthesized Systems", IEEE International On-line Testing Workshop, 1995, p. 196-200.

75. A.C. Williams, A.D. Brown, M. Zwolinski, "In-line test of synthesized systems exploiting latency analysis", IEE Proceedings : Computers and Digital Techniques, Vol. 147, No. 1, January 2000, p. 33-41.

76. H. Al-Asaad, J.P. Hayes, B.T. Murray, "Scalable test generators for high-speed datapath circuits", Journal of Electronic Testing – Theory and Applications, Vol. 12, No. 1-2, February – April 1998, p. 111-125.

77. G. Al-Hayek, C. Robach, "From Specification Validation to Hardware Testing : A Unified Method", IEEE International Test Conference, 1996, p. 885-893.

78. F. Ferrandi, G. Ferrara, G. Fornana, F. Fummi, D. Sciuto, "Testability Alternatives Exploration through Functional Testing", IEEE VLSI Test Symposium, 2000, p. 423-428.

79. R. Singh, J. Knight, "Concurrent testing in High Level Synthesis", IEEE International Symposium on High-level Synthesis, 1994, p. 96-103.

80. M.L. Flottes, D. Hammad, B. Rouzeyre, "Automatic Synthesis of BISTed Data Paths From High Level Specification", European Design & Test Conference, 1994.

81. U. Kac, G. Papa, F. Novak, J. Silc, "On-line testing of a discrete PID regulator : a case study", EUROMICRO 1997, p. 216-221.

82. U. Kac, F. Novak, C. Aktouf, C. Robach, "Combined Resource Allocation and Test Generation for On-line Test Structures", IEEE International On-line Testing Workshop, 1999, p. 211-215.

83. A.A. Ismaeel, R. Bhatnagar, R. Mathew, "Concurrent testing in high-level synthesis", Microelectronics Reliability, Vol. 40, No. 12, December 2000, p. 2095-2106.

84. A.A. Ismaeel, R. Bhatnagar, R. Mathew, "Modification of scheduled data flow graph for on-line testability", Microelectronics Reliability, Vol. 39, No. 10, October 1999, p. 1473-1484.

85. A.A. Ismaeel, R. Mathew, R. Bhatnagar, "Module allocation for on-line testing", Microelectronics Reliability, Vol. 40, No. 6, June 2000, p. 1011-1021.

86. M.A. Naal, E. Simeu, "High Level Synthesis Methodology for On-line Testability Optimization", IEEE International On-line Testing Workshop, 2000, p. 201-206.

87. F. Mayer, A.P. Stroele, "A Versatile BIST technique combining test registers and accumulators", IEEE International Conference on VLSI Design, 2000, p. 412-415.

88. A.P. Stroele, "Synthesis for arithmetic built-in self-test", IEEE VLSI Test Symposium, 2000, p. 165-170.

89. A.P. Stroele, "Synthesizing data paths with arithmetic self-test", IEEE International Symposium on Circuits and Systems, 2000, p. II-45 – II-48.

90. N. Mukherjee, J. Rajski, J. Tyszer, "Design of Testable Multipliers for Fixed-width Data Paths", IEEE Transactions on Computers, Vol. 46, No. 7, July 1997, p. 795-810.

91. D. Gizopoulos, A. Paschalis, Y. Zorian, "An Effective BIST Scheme for Datapaths", IEEE International Test Conference, 1996, p.76-85.

92. R. Karri, N. Mukherjee, "Versatile BIST : An Integrated Approach to On-line / Off-line BIST", IEEE International Test Conference, 1998, p. 910-917.

93. I. Bayraktaroglu, A. Orailoglu, "Low-cost on-line test for digital filters", IEEE VLSI Test Symposium, 1999, p. 446-451.

94. I. Bayraktaroglu, A. Orailoglu, "Unifying Methodologies for High Fault Coverage Concurrent and Off-line Test of Digital Filters", IEEE International Symposium on Circuits and Systems, v. 2, 2000, p. II-705 – II-708.

95. A. Abdelhay, E. Simeu, "Analytical Redundancy Based Approach for Concurrent Fault Detection in Linear Digital Systems", IEEE International On-line Testing Workshop, 2000, p. 112-117.

96. A. Bogliolo, M. Favalli, M. Damiani, "Enabling Testability of Fault-Tolerant Circuits by means of I_{DDQ}-checkable voters", IEEE Transactions on VLSI, Vol. 8, No. 4, August 2000, p. 415-418.

97. A. Paschalis, D. Gizopoulos, N. Gaitanis, "Concurrent Delay Testing in Totally Self-Checking Systems", Journal of Electronic Testing – Theory and Applications, Vol. 12, No. 1-2, February-April 1998, p. 55-61.

98. M. Favalli, C. Metra, "Bus Crosstalk Fault-Detection Capabilities of Error Detecting Codes for On-line Testing", IEEE Transactions on VLSI, Vol. 7, No. 3, September 1999.

99. C. Metra, M. Favalli, B. Ricco, "On-line detection of logic errors due to crosstalk, delay and transient faults", IEEE International Test Conference, 1998, p. 524-533.

100. P. Kollig, "Algorithms for Scheduling, Allocation and Binding in High Level Synthesis", PhD Thesis, Staffordshire University, 1998.

101. G. Economakos, P. Oikonomakos, I. Poulakis, I. Panagopoulos, G. Papakonstantinou, "Behavioral Synthesis with SystemC", Design Automation and Test in Europe (DATE), 2001, p. 21-25.

102. G. De Micheli, "Synthesis and Optimisation of Digital Circuits", McGraw – Hill 1994.

103. M.T.C. Lee, "High-Level Test Synthesis of Digital VLSI Circuits", Artech House 1997.

104. 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.

105. MOODS Internals, Version 1.0, July 2001.

106. Xilinx Virtex 2.5V Field Programmable Gate Arrays Product Specification DS003-1, Version 2.5.
(available at http://www.xilinx.com/partinfo/ds003.pdf)

107. J. Gracia, J.C. Baraza, D. Gil, P.J. Gil, "Comparison and Application of Different VHDL-Based Fault Injection Techniques", IEEE International Symposium on Defect and Fault Tolerance in VLSI System, 2001, p. 233-241.

108. D. Gil, J. Gracia, J.C. Baraza, P.J. Gil, "Study, comparison and application of different VHDL-based fault injection techniques for the experimental validation of a fault-tolerant system", Microelectronics Journal, Vol. 34, No. 1, January 2003, p. 41-51.

109. L. Antoni, R. Leveugle, B. Feher, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes", IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, 2000, p. 405-413.

110. M.-C. Hsueh, T.K. Tsai, R.K. Iyer, "Fault Injection Techniques and Tools", IEEE Computer, Vol. 30, No.4, April 1997, p. 75-82.

111. http://www.sertest.com/

112. P. Civera, L. Macchiarulo, M. Rebaudengo, M Sonza Reorda, M. Violante, "An FPGA-based Approach for Speeding-Up Fault Injection Campaigns on Safety-Critical Circuits", Journal of Electronic Testing – Theory and Applications, Vol. 18, No. 3, June 2002, p. 261-272.

113. L. Antoni, R. Leveugle, B. Feher, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes", IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, 2002, p. 245-253.

114. S.A. Aftabjahani, Z. Navabi, "Functional Fault Simulation of VHDL Gate Level Models", VHDL International User's Forum, 1997, p. 18-24.

115. http://www.model.com/

116. D. Shaw, D. Al-Khalili, C. Rozon, "Deriving accurate ASIC cell fault models for VITAL compliant VHDL simulation", IEEE International Symposium on Circuits and Systems, v. 5, 2001, p. V-263 – V-266.

117. T.A. Delong, B.W. Johnson, J.A. Profeta III, "A Fault Injection Technique for VHDL Behavioral-Level Models", IEEE Design & Test of Computers, Vol. 13, No. 4, Winter 1996, p. 24-33.

118. M. Zwolinski, "A Technique for Transparent Fault Injection and Simulation", Microelectronics Reliability, Vol. 41, No. 6, June 2001, p. 797-804.

119. B. Stroustup, "The C++ Programming Language", Addison Wesley, 1997.

120. J.F. Meyer, R.J. Sundstrom, "On-Line Diagnosis of Unrestricted Faults", IEEE Transactions on Computers, Vol. 24, No. 5, May 1975, p. 468-475.

121. C.-J. Tseng, D.P. Siewiorek, "Facet: A procedure for the automated synthesis of digital systems", Design Automation Conference (DAC), 1983, p. 490-496.

122. N.R. Saxena, S. Fernandez-Gomez, W.-J. Huang, S. Mitra, S.-Y. Yu, E.J. McCluskey, "Dependable Computing and Online Testing in Adaptive and Configurable Systems", IEEE Design & Test of Computers, Vol. 17, No. 1, January-March 2000, p. 29-41.

123. G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, V. Piuri, "Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard", IEEE Transactions on Computers, Vol. 52, No. 4, April 2003, p. 492-505.

124. http://www.synplicity.com/

125. http://toolbox.xilinx.com/docsan/3_1i/

126. MOODS User Guide, Version 1.2 (alpha), August 2001.

127. MOODS VHDL Reference, Version 1.2 (alpha), August 2001.

128. MOODS VHDL Style Guide, Version 1.2 (alpha), August 2001.

129. N. Dutt, C. Ramachandran, "Benchmarks for the 1992 High-level Synthesis Workshop", Technical Report #92-107, University of California Irvine, 1992.

130. P.R. Panda, C. Ramachandran, "1995 High-level Synthesis Design Repository", Technical Report #95-04, University of California Irvine, 1995.

131. J.S. Reeve, K. Amarasinghe, "A FPGA Implementation of a Parallel Viterbi Decoder for Block Cyclic and Convolution Codes", IEEE International Conference on Communications, 2004, p. 2596-2599.

132. http://www.mentor.com/leonardospectrum/

133. S. Hellebrand, H.-J. Wunderlich, A. Hertwig, "Synthesizing Fast, Online-Testable Control Units", IEEE Design & Test of Computers, Vol. 15, No. 4, October-December 1998, p. 36-41.

134. R. Karri, B. Iyer, I. Koren, "Phantom redundancy: a register transfer level technique for gracefully degradable data path synthesis", IEEE Transactions on CAD, Vol. 21, No. 8, August 2002, p. 877-888.

135. J. Carletta, M. Nourani, C.A. Papachristou, "Synthesis of Controllers for Full Testability of Integrated Datapath-Controller Pairs", Design Automation and Test in Europe (DATE), 1999, p. 278-282.

136. N.H.E. Weste, K. Eshraghian, "Principles of CMOS VLSI Design", Addison Wesley 1992.

137. P.H. Bardell, "Design Considerations for Parallel Pseudorandom Pattern Generators", Journal of Electronic Testing – Theory and Applications, Vol. 1, 1990, p. 73-87.

138. P.H. Bardell, "Primitive Polynomials of Degree 301 - 500", Journal of Electronic Testing – Theory and Applications, Vol. 3, 1992, p. 175-176.

139. N.K. Jha, S.-J. Wang, "Design and synthesis of self-checking VLSI circuits", IEEE Transactions on CAD, Vol. 12, No. 6, June 1993, p. 878-887.

140. P.A. Thaker, V.D. Agrawal, M.E. Zaghloul, "A test evaluation technique for VLSI circuits using register-transfer level fault modelling", IEEE Transactions on CAD, Vol. 22, No. 8, August 2003, p. 1104- 1113.

141. P.A. Thaker, V.D. Agrawal, M.E. Zaghloul, "Register-transfer level fault modeling and test evaluation techniques for VLSI circuits", IEEE International Test Conference, 2000, p. 940-949.

142. J. Gracia, D. Gil, L. Lemus, P. Gil, "Studying Hardware Fault Representativeness with VHDL Models", Conference on Design of Circuits and Integrated Systems, 2002.

143. http://www.iroctech.com/

144. F. Corno, P. Prinetto, M. Sonza Reorda, "Fault Tolerant and BIST Design of a FIFO cell", European Design Automation Conference, 1996, p. 233-238.