Built-In Self Test Design of an Asynchronous Block Sorter

O. A. Petlin, C. Farnsworth, S. B. Furber

Department of Computer Science, The University, Manchester, M13 9PL, UK

{oleg, cfarnsworth, sfurber}@cs.man.ac.uk

Abstract - The design of an asynchronous block sorter and issues relating to its testability are discussed in this paper. The sorter takes an input data stream and sends it to the output sorted in descending order. The testable structure of the block sorter is implemented using the built-in self test (BIST) design methodology. A novel technique for changing the operation mode of the sorting cells of the block sorter which allows them to be set to normal operation mode or BIST mode sequentially or in parallel respectively is described. In BIST mode the sorting cells are tested in parallel reducing the overall test application time of the sorter. Fault simulation results reveal 100% testability of both single stuck-at-output faults at the high-level representation of the block sorter and all stuck-at faults inside data processing blocks of its sorting cells. The total area overhead of the BIST block sorter is 15.7%.

Key words - Asynchronous circuit, VLSI, VLSI programming, *Tangram* language, stuck-at fault model, built-in self testing, random testing.

1. Introduction

Asynchronous design methodologies are a subject of growing research interest since they appear to offer benefits in low power applications, promise greater design modularity, exhibit average case performance rather than worst case performance and have no clock skew problem [Lav93, Hauck95].

Philips Research Laboratories has developed the Tangram programming language which is supported by a set of CAD tools for the design of asynchronous VLSI circuits [Berk88, Berk91, Scha93]. Tangram describes the VLSI circuit as a set of processes which communicate along channels. The Tangram program is translated into an intermediate format called a handshake circuit representation. Handshake circuits are composed of handshake components and channels on which they communicate [Berk93]. Channels communicate with each other using a 4-phase (return-to-zero) signalling protocol [Mead80]. The result of compiling the Tangram program is a silicon layout with particular performance, power consumption and silicon area properties. The transparency of the compilation process makes it possible to go back to the Tangram program level where it is easy to make improvements to the properties of the target VLSI design.

A great deal of engineering work has been carried out in the AMULET group at the University of Manchester to optimise the design flow of asynchronous circuits initially described in the Tangram language [Farn95, EdTR95]. In collaboration with Philips Research Laboratories an effective design scheme called "hybrid" design environment has been developed and incorporated within the *Cadence* design framework using *SIMIC* design verification tools [Farn95, Sim94]. As a result, during the design process which goes from the Tangram specification to a handshake circuit implementation an engineer may replace Tangram synthesised subcircuits with more efficient hand-developed design solutions.

Handshake circuits can be translated into delay-insensitive, speed-independent or bounded-delay asynchronous circuits [Berk91, Lav93, Hauck95]. In delay-insensitive circuits all delays in gates and wires are allowed to be arbitrary but finite. Gate delays in speed-independent circuits are arbitrary and finite but signal transmission along wires is instantaneous.

This assumption allows the use of the isochronic fork [Berk91], i.e., the arrival times of transitions at their destinations are equal. Both delay-insensitive and speed-independent implementations require dual-rail encoding of data where each data bit is represented by two wires: a ‘zero’ propagation wire and a ‘one’ propagation wire. Bounded-delay asynchronous circuits allow a single-rail data implementation to be used. Thus, combinational logic circuits similar to those used in synchronous designs can be used directly which offers a significant reduction in silicon area. A bounded-delay circuit uses the bounded delay model to ensure correct data processing. In this model the delays through the data paths of the circuit are known and bounded. The sender generates request signals for the receiver only when the data is stable and ready to be transmitted.

The testing of asynchronous circuits for fabrication faults is more complex than that of synchronous circuits. The major factors that complicate the testing of asynchronous circuits are [Hulg94]:

- The presence of a large number of state holding elements. This makes the generation of tests hard or even impossible.
- The difficulty of detecting hazards and races.
- The absence of global synchronization clocks. This decreases the level of controllability of internal circuit states since asynchronous circuits perform as autonomous blocks.

The most widely accepted fault model used to represent different fabrication failures in VLSI designs is the stuck-at fault model [MClus86, Russ89, Weste93]. A stuck-at fault on line *a* connects it to the power supply voltage (V_{dd}) or ground (V_{ss}) permanently. It has been observed that stuck-at-output faults in control circuits where each transition is confirmed by another cause the whole circuit to halt; this is called the self-diagnostic property of delay insensitive circuits [Dav90]. Some stuck-at-input faults can cause premature firing on the control line of the circuit. The detection of such faults requires a special testability analysis to be made [Haz92]. In this paper we consider two types of stuck-at faults: stuck-at-output faults in the control logic and gate-level stuck-at-output and stuck-at-input faults inside the data processing blocks of the test object. As was mentioned above, stuck-at-output faults on the control lines of the handshake circuit exhibit themselves by causing the circuit to deadlock when it operates normally. Thus, there is no need to develop any test techniques to detect such faults. Stuck-at faults in dual-rail encoded data paths are identified easily since they prevent any transitions on the corresponding data lines. A special testability analysis must be carried out to detect stuck-at faults on the single rail encoding data paths of the circuit under test.

The generation of tests for a system-level chip is difficult due to the high complexity of the circuit under test, especially if the circuit is implemented using design techniques which do not consider testability issues [MClus86]. Several design-for-testability techniques for asynchronous circuits have been reported. A test strategy for delay-insensitive circuits has been described [Ron93]. It was shown that this strategy allows circuits to be tested for stuck-at faults in linear time. Unfortunately, it does not address the testing issues of the circuit's data paths. A number of solutions to adapt the scan testing technique for testing micropipelines have been reported [Khoc94, Pet95]. A scan design technique to test asynchronous sequential machines was described by Wey et al. [Wey93]. These approaches reduce the test generation problem of complex sequential circuits to that of testing their combinational logic. The testable designs reported assume isolated testing of the circuit, i.e., its inputs and outputs are fully controllable and observable. Such an approach does not consider the case of the asynchronous circuit which is a part of a large asynchronous VLSI design where each asynchronous block operates autonomously.

A partial scan test technique for asynchronous circuits has been described by Marly Roncken [Ron94]. Taking the digital compact cassette (DCC) error corrector decoder as an example it was shown how the asynchronous partial scan test technique can be adapted into the high-level VLSI programming environment. The DCC error corrector decoder performs in test and normal operation modes. An asymmetric isochronic fork was used for switching the mode of operation of the circuit. It is assumed that there is a distinguished branch on which transitions are guaranteed to move faster along this fork. Unfortunately, the use of the asymmetric isochronic fork does not allow the change of mode to be implemented using delay-insensitive control logic. The reported scan solution was designed for testing the error corrector and a controller which use dual-rail data encoding.

In this paper, we present a BIST technique for asynchronous VLSI circuits with regular structures and single-rail data encoding. A case study of an asynchronous block sorter is reported. The rest of the paper is organised as follows: Section 2 describes the design of the asynchronous block sorter. The testing issues of the block sorter and the testable structure of the sorting cells are considered in Section 3. The asynchronous procedure for changing the mode of operation of the block sorter is described in Section 4. The design of the asynchronous random pattern generator used in the BIST block sorter is discussed in Section 5. Section 6 presents the simulation results and cost comparisons of the BIST version of the block sorter. Finally, Section 7 concludes the paper.

2. Design of the asynchronous block sorter

The original design of the asynchronous block sorter was developed in Philips Research Labs and generated in the hybrid design environment worked out at the University of Manchester [Berk88, Farn95, FarnTR96]. A high-level diagram of the block sorter is shown in Figure 1. It consists of a head cell, 64 sorting cells and a tail cell which are connected in a chain. All these blocks are fully asynchronous and operate autonomously. The head takes 16-bit vectors from its input and passes 17 bit vectors to the first sorting cell. An extra boolean flag is added by the head to the input data. The first 63 input vectors have zero flags and only the last 64-th input vector has a boolean flag set to one. Flag one means the end of the block. The head contains an asynchronous modulo-63 counter which counts the number of input vectors passing through it. After completing 63 handshakes the head takes the last input vector and changes its boolean flag to one. When 64 input vectors have been passed through the head to the first sorting cell the head is ready to take a new block of input vectors.

Each sorting cell of the block sorter compares the 8 most significant bits of each new input vector[1] with a value stored in a register within the cell. Afterwards, each cell passes the

1. According to the specification of the block sorter

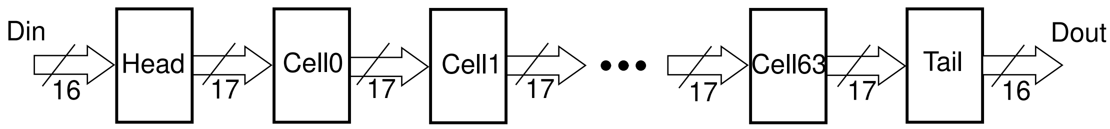

Figure 1 : *High-level design of the block sorter*

minimum vector to its output and stores the maximum one in its internal register. As a result, the set of 64 input vectors are stored in 64 sorting cells in a descending order.

The sorted block of 64 17-bit vectors are sent to the tail. The tail strips the boolean flag off its input value. The Tangram program for the asynchronous four-stage block sorter and handshake implementations of its basic components can be found elsewhere [FarnTR96].

All the sorting cells of the block sorter are identical. A block diagram of the sorting cell is shown in Figure 2. When a new block of input data is sent by the head to the sorting cells, the first 17-bit input vector of the block is passed from the *Din* inputs to the inputs of multiplexer *MX1* and register *Reg2*. Note that the first vector of each new block of data is always stored in register *Reg1* first. A new input vector is latched in register *Reg2*.

The 8 most significant bits of registers *Reg1* and *Reg2* are compared by comparator *CMP*. The design of the comparator is similar to the implementation of an asynchronous adder the *CarryOut* signal of which is used as the output of the comparator. Various designs of an

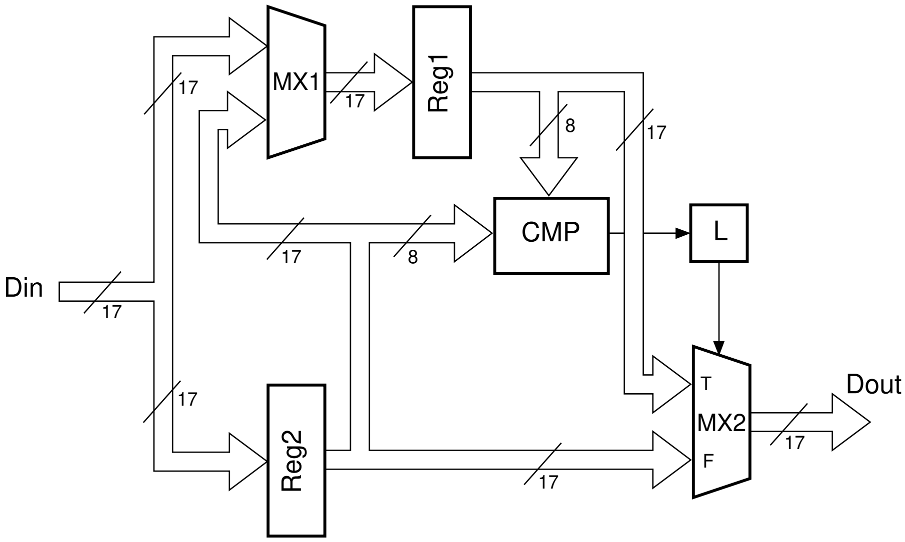

Figure 2 : *Block diagram of the sorting cell*

asynchronous adder are investigated elsewhere [Gars93, Pet96a, Pet96b]. In particular, the comparator *CMP* was designed using the hybrid implementation of its asynchronous adder which provides for the detection of all its stuck-at faults without the introduction of a specific test mode [Pet96b].

The result of the comparison is stored in latch *L* which controls multiplexer *MX2*. If the value stored in *Reg1* is greater than or equal to the current value of *Reg2* then the state of latch *L* is zero and the vector written in *Reg2* is sent through multiplexer *MX2* to the output of the cell. If the value of *Reg1* is less than the value written in *Reg2* then the state of latch *L* is changed to one. As a result, the contents of *Reg1* are passed through multiplexer *MX2* to its outputs. The contents of *Reg2* are copied into *Reg1* and the cell is ready to write a new vector into *Reg2* in order to compare it with the contents of *Reg1*. The sorting procedure described above is repeated.

3. Testing the block sorter

3.1 Fault model

To design the asynchronous block sorter for testability we assume the stuck-at fault model. In particular we consider two types of stuck-at faults:

- gate-level stuck-at output faults inside the control circuits of the block sorter;

- gate-level stuck-at faults (including stuck-at input and output faults) inside the data processing blocks of the sorter.

Stuck-at faults on the control wires of the sorter are detected during its normal operation mode since they violate the handshaking protocol causing the whole circuit to halt [Dav90, Ron94]. The detection of stuck-at faults on the data paths of the circuit requires more care to be taken since they may or may not manifest themselves by producing wrong responses on the outputs of the block sorter.

3.2 Testable implementation of the sorting cell

The testing of data paths inside sorting cells is difficult since the controllability and observability of their internal nodes are different and dependent on the position of each cell in the chain. The testability of the block sorter can be increased by making the states of the registers inside the sorting cells controllable. One of the approaches to increase the controllability of the memory elements of the circuit under test is scan testing.

In principle, the scan testing technique assumes that all the registers of the testable design can be transformed into a shift register to scan the test vectors in and scan the test results out. As can be seen from the design of the sorting cell (Figure 2) the main data processing block which must be tested is the comparator. The full scan test implementation of the sorting cell requires the use of master-slave registers instead of *Reg1* and *Reg2* which effectively doubles their sizes. These registers would be used just to scan the test vectors in since the test results are not stored in them (Figure 2). Thus, the use of scannable registers is not efficient to test the sorting cell.

The structure shown in Figure 3 uses the system data paths of the sorting cell to send test vectors to registers *Reg1* and *Reg2*. The state of latch *L* can be observed on an additional output *CmpRes*. An extra multiplexer *MX3* is required to send either test or normal 8-bit

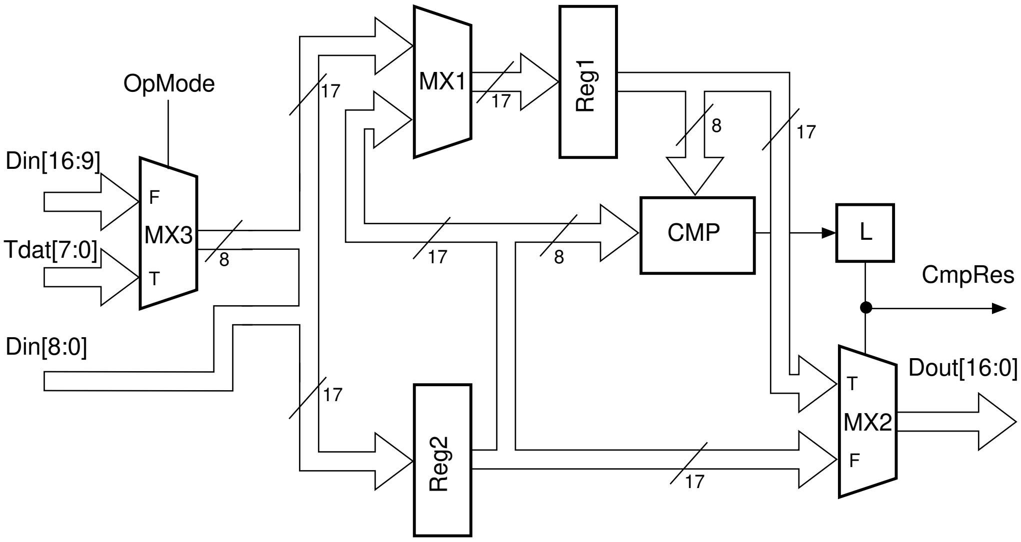

Figure 3 : *Testable sorting cell*

input vectors to the registers of the cell depending on the mode of operation. An additional boolean signal *OpMode*, which is high in test mode, controls multiplexer *MX3*.

In test mode multiplexer *MX3* passes 8-bit test vectors to the inputs of multiplexer *MX1* and register *Reg2*. The first and the second test vectors are written in the corresponding latches of registers *Reg1* and *Reg2* respectively. The data on bus *Din[8:0]* remains unchanged since it is not used to test the comparator. After the application of the pair of test vectors the test result is stored in latch *L* and can be observed on its output *CmpRes*. Thus, stuck-at faults inside the comparator can be detected during test mode.

The sorting cell is set to normal operation mode (*OpMode*=0) to test the rest of the circuit. Two blocks of tests are required to detect stuck-at faults on the data paths of the sorting cell:

- a block of 'all ones' tests;

- a block of 'running one' tests which starts with 'all zeros' test.

The first block of tests detects all stuck-at zero faults on the data paths involved in transferring the data to the outputs of the cell except the internal bus which connects the outputs of *Reg2* with the inputs of multiplexer *MX1*. The second block of tests detects the rest of the stuck-at faults which have not been identified by the previous test block. Since the test blocks must be applied to the inputs of the block sorter the size of each test block must be equal to the number of the sorting cells, i.e., 64.

3.3 Design for testability of the block sorter

Figure 4 illustrates the BIST implementation of the block sorter. The operation mode of the circuit is changed asynchronously along a two-phase *ChMode2* channel which consists of two wires: a request and an acknowledge wire. The mechanism for changing the operation mode of the block sorter will be discussed later.

In normal operation mode, the two blocks of test vectors described above are applied to inputs *Din* of the sorter. As a result, stuck-at faults on the control lines and the data paths of

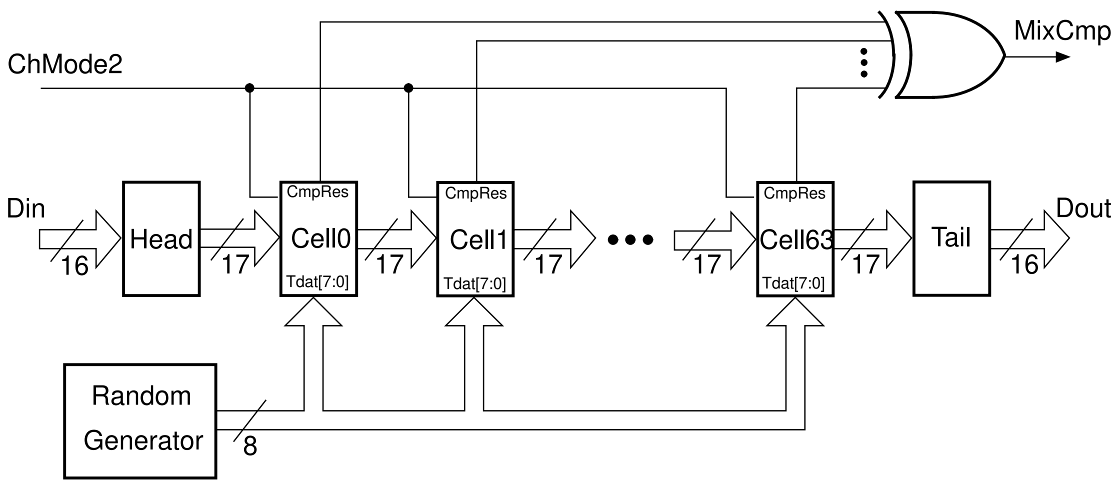

Figure 4 : *BIST design of the block sorter*

the block sorter are tested including the head and the tail of the block sorter. The next step is to test internal gate stuck-at faults of the comparator inside each sorting cell.

In test mode, test vectors are produced by the asynchronous random pattern generator (see Section 5) placed on a chip. Random tests are applied to each sorting cell in parallel since all cells are identical. The test results are mixed by a 64-input XOR gate and observed on its output *MixCmp*. In the presence of a single stuck-at fault inside the comparator of any sorting cell the output *MixCmp* will be changed to one during the test. Output *MixCmp* remains zero all the time during the test for the fault free block sorter.

4. Procedure for changing the operation mode

As was mentioned earlier a handshake circuit is a network of handshaking components which communicates via its channels. Each channel consists of a set of at least two wires (a request and an acknowledge wire) along which a four-phase communication protocol is performed. If the component performs a particular data processing operation an additional wire or a set of wires can be included to pass the data in and out. A transmission of signals along a channel can be made only after the receiver has confirmed the receipt of the previous transmission.

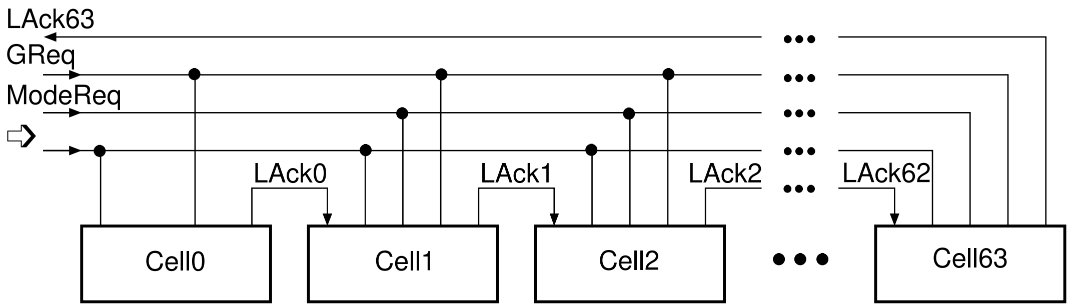

Figure 5 : *Diagram which shows the changing of the operation mode of the sorting cells*

States of asynchronous circuits are difficult to control since they operate autonomously. All the cells of the asynchronous block sorter are fully autonomous blocks which can be in any state during their operation. In order to set the sorter to test or normal operation mode one must be sure that the circuit cells are empty, i.e., the sorter has processed the current block of input data and ready to take a new one. The procedure for changing the operation mode of the circuit must be asynchronous since the cells of the sorter enter their empty states asynchronously and at different times.

4.1 Sorting cell implementation for changing the operation mode

A fragment of the chain of the 64 sorting cells is shown in Figure 5. The sorting cells are activated along the activation channel marked with ⇨. Suppose the block sorter operates in normal mode. In order to change the operation mode of the sorting cells to test mode the *ModeReq* input is set to high externally. As a result, the *GReq* signal goes high. When the first cell is empty its operation mode is changed to test mode and the acknowledge signal *LAck0* is set to high. Once the second cell went through the empty state its operation mode is changed to test mode generating a rising event on the *LAck1* output. The procedure for setting the rest of the sorting cells to test modes repeats sequentially till the last cell. Once all the cells have been set to test mode the *LAck63* signal goes high. A a result, a falling event is produced on the *GReq* input of the block sorter. The acknowledge signals Ack_i (i=0, 1, ... , 63) of all the sorting cells are reset sequentially starting from the first cell. When the *LAck63* signal has been reset the sorting cells can be tested.

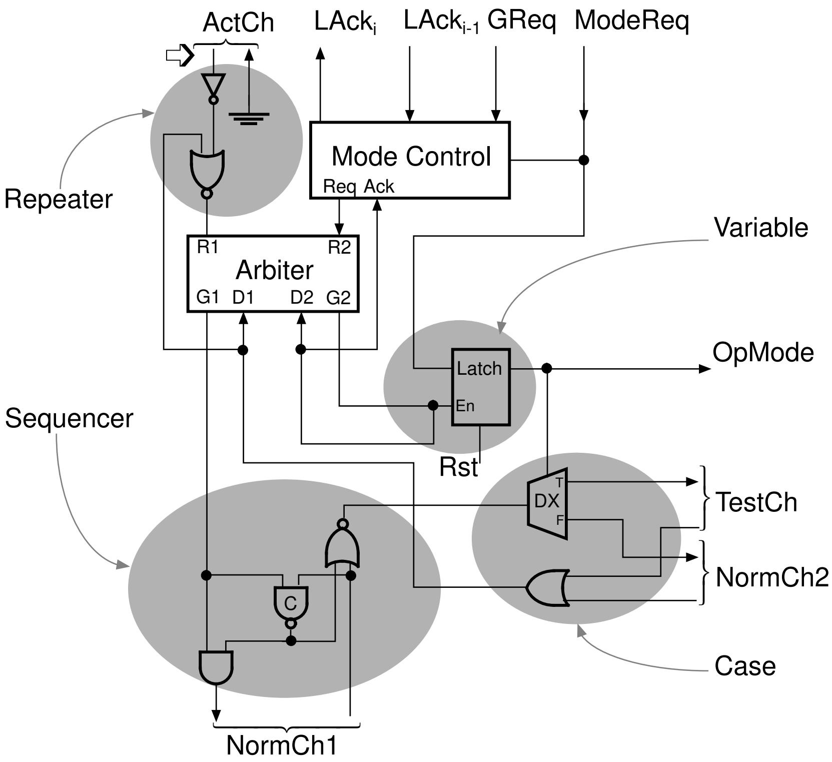

Figure 6 : *Fragment of the handshake implementation of the testable sorting cell*

The operation mode of the block sorter is changed to normal operation mode by resetting its *ModeReq* input. Thus, the *GReq* signal goes high. Since the sorting cells are tested in parallel they pass their empty states at nearly the same time. As a result, the operation mode of the sorting cells is changed in parallel. Once the *LAck63* output of the last cell has been set to high the *GReq* signal is reset. When *LAck63* is set to low the block sorter can operate according to its specification.

An arbiter can be used to identify the situation when the sorting cell is empty. Different designs of arbiter circuits can be found elsewhere [Chan73]. The arbiter takes a request for changing the operation mode of the sorting cell and waits until the cell goes to the empty state. When it happens the arbiter serves the request and changes the operation mode of the cell. Figure 6 shows a fragment of the handshake implementation of the sorting cell. This sorting cell contains an arbiter which is placed after the repeater. The left communication channel of the arbiter serves requests when the cell is empty and ready to process a new data item. The right channel is used for changing the operation mode of the circuit.

Initially, the Boolean signal *ModeReq* is low and the storage element called *variable* is reset. The operation mode signal *OpMode* is low and the case element is switched to normal mode where its passive port is connected to its active port *NormCh2*. The sorting cell is activated along its activation channel (➪).

The *ModeReq* signal is set to high in order to change the operation mode of the cell to test mode. As a result, a rising event is generated on input *GReq* of the *mode control* circuit. When the (i-1)-th sorting cell (i=1, 2, ... , 63) has been set to test mode ($LAck_{i-1}$ goes high) the request output *Req* of the mode control circuit is set to one and the arbiter of the i-th cell waits until a handshake is completed on its left channel to activate its right channel. Once the i-th cell went through its empty state (the *D1* input of the arbiter has been reset) the *G2* output of the arbiter goes high, setting the output of the variable to high (*OpMode*=1). The case element is switched connecting its passive port to its active port *TestCh*. A rising event is produced on the acknowledge input *Ack* of the mode control and the $LAck_i$ signal goes high.

If the $LAck_{i-1}$ input is reset the *Req* output of the mode control is set to low, resetting the *G2* output of the arbiter. As a result, a handshake is completed along the right channel of the arbiter and the latch of the variable is closed. A rising request event produced on the request output of the active port of the repeater is transmitted by the arbiter to its *G1* output, activating the passive channel of the *sequencer*.

The block sorter is set to normal operation mode when the boolean signal *ModeReq* is set to low. As a result, the mode control circuit passes a rising event generated on the *GReq* input of the i-th cell directly to its *Req* output. When a handshake is completed along the left channel of the arbiter (the cell is empty) the *G2* output of the arbiter is set to high, resetting the output of the variable. Thus, the case element is switched connecting its passive port to active port *NormCh2*. Once a handshake has been completed along the right channel of the arbiter a rising event produced on the request output of the passive port of the repeater is transmitted to the *G1* output of the arbiter and the cell can perform in normal operation mode.

4.2 Mode control

Figure 7a illustrates an implementation of the mode control circuit of the *i*-th sorting cell (i=2, 3, ... , n-2; n=63). The *ModeReq* input of this circuit is used to control the C-element *C1* which acts a symmetric (*ModeReq*=1) or asymmetric C-element (*ModeReq*=0). A CMOS implementation of *C1* is shown in Figure 8.

Initially, all the C-elements of the mode control are reset. In test mode the *ModeReq* is set to high and *C1* operates as a symmetric C-element. The *GReq* input is set to high. Once the $LAck_{i-1}$ input of the C-element *C2* has been set to high its state is changed to one, setting the

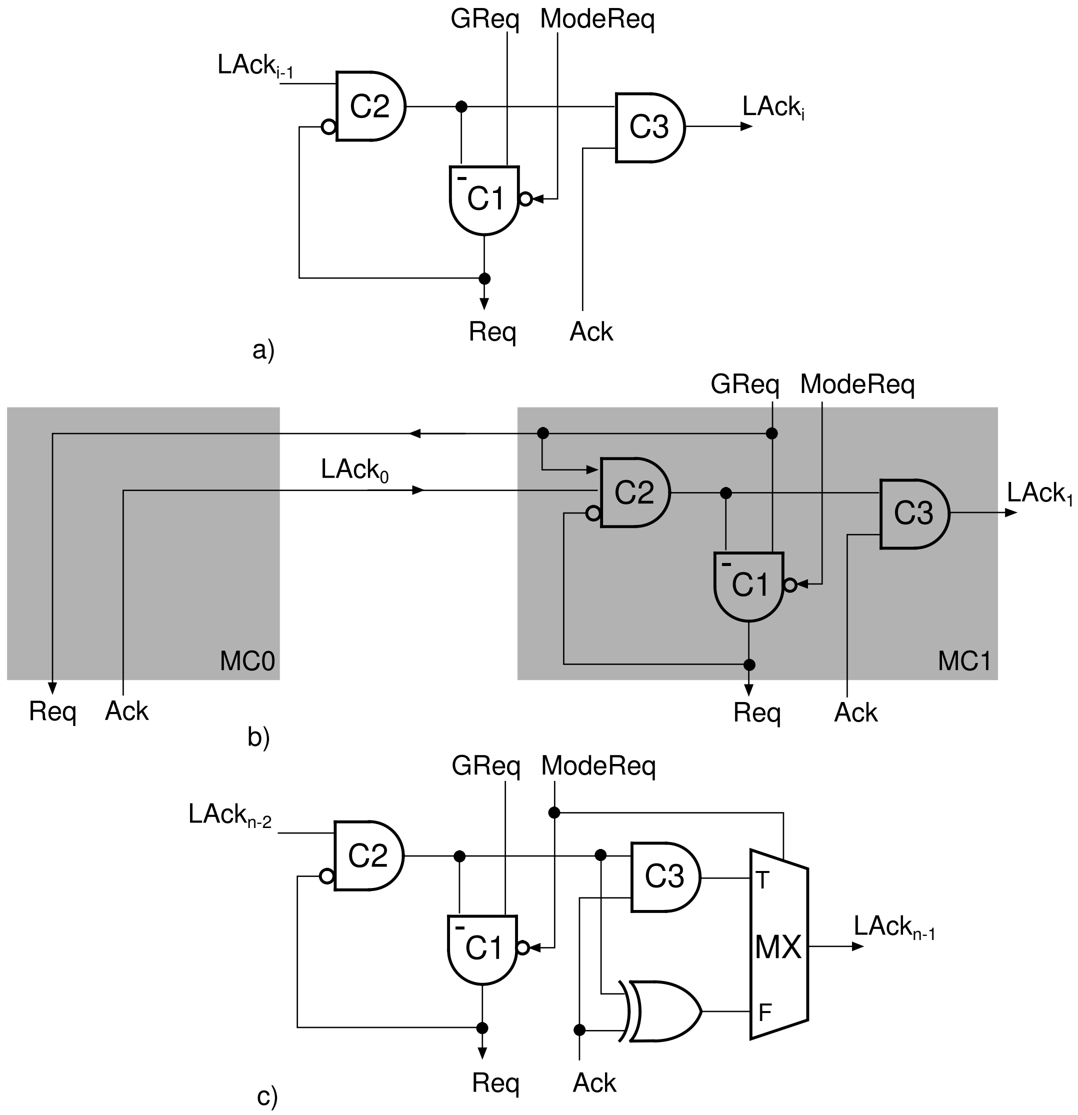

Figure 7 : *Mode control circuit implementations for a) the i-th (i=2,3, ... , n-2); a) the first and second; c) the last sorting cells*

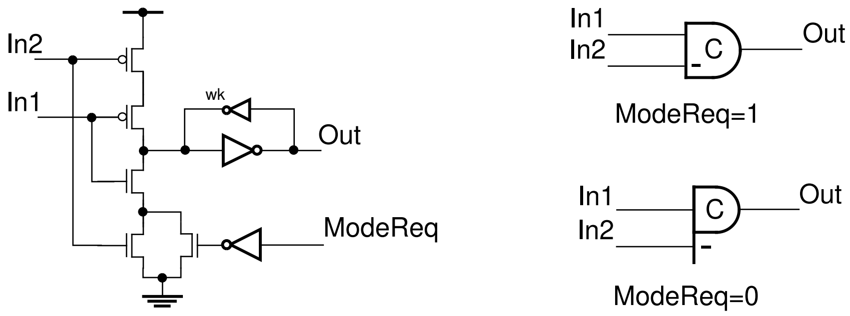

Figure 8 : *CMOS implementation of the C-element C1*

output of *C1* to high. When *Ack*=1 the output of C-element *C3* is set to one ($LAck_i$=1). The *GReq* signal is reset. When the $LAck_{i-1}$ input has been set to low the output of *C2* is reset. As a result, *Req* goes low. Once acknowledged (the *Ack* input is low) the output of *C3* is reset.

In normal operation mode (*ModeReq*=0) *C1* acts as an asymmetric C-element and the outputs of *C2* and *C3* are held at zero permanently. Thus, *C1* repeats signals from its *GReq* input to its *Reg* output. The output of *C3* is low because the output of *C2* is kept at zero when the cell is set to normal operation mode. The state of *C2* cannot be changed to one since the $LAck_{i-1}$ input is held at zero by the previous cell.

Figure 7b shows implementations of the mode control circuits *MC0* and *MC1* of the first and second sorting cells respectively. Circuit *MC0* passes a request signal from its *GReq* input directly to its *Req* output whereas acknowledge events from the *Ack* input are transformed to the *LAck0* output. The implementation of *MC1* is similar to the mode control circuit illustrated in Figure 7a except its C-element *C2* which has three inputs to ensure that its state is zero during the changing of the operation mode of the cell to normal mode. For instance, let us assume that *Req*=*LAck0*=1. When *GReq* goes low the *Req* signal is reset, priming *C2*. The two-input C-element *C2* shown in Figure 7a will be set to state one if the *LAck0* is still high. Thus, the third input of *C2* connected to the *GReq* input (see Figure 7b) prevents the state of *C2* from changing to one.

The mode control circuit of the last sorting cell (see Figure 7c) is similar to the one illustrated in Figure 7a. A multiplexer is added to the original design in order to pass the *Ack* signal through *C3* in test mode (*ModeReq*=1) and the *XOR* gate in normal operation mode (*ModeReq*=0). Since the output of *C2* is held at zero permanently when the operation mode of the block sorter is changed to normal mode the XOR gate acts as a repeater of acknowledge signals generated on its *Ack* input.

Testability of the mode control circuits

A fault analysis of the mode control circuits of the sorting cells connected in a chain (see Figure 5) carried out using *SIMIC* fault simulator revealed that their single stuck-at faults cause the block sorter to halt [Sim94].

For example, stuck-at faults on inputs and outputs of the C-elements (see Figure 7), the XOR gate and the multiplexer (in Figure 7c) violate the handshake protocol during the changing of operation mode of the block sorter.

A stuck-at-1 fault on the *ModeReq* input prevents the corresponding cell from setting to normal operation mode since the output of the *Req* output cannot be changed (the output of *C2* is held at zero). As a result, the block sorter will halt in normal operation mode.

A stuck-at-0 fault on the *ModeReq* input prevents a rising event generated on the $LAck_{i-1}$ input from propagating through the corresponding C-element *C2* since the *Req* output is set to high first by a high signal applied to the *GReq* input. Thus, the communication protocol is broken and the circuit halts.

A stuck-at-1 fault on the control input of the multiplexer of the circuit shown in Figure 7c can be easily identified by preventing any activity on its output in normal operation mode. A stuck-at-0 fault on the *ModeReq* input of this multiplexer can be detected in test mode. In this mode both inputs of the XOR gate are activated. As a result, two rising and two falling event are transmitted to the output of the multiplexer which violates the communication protocol and causes the whole circuit to deadlock.

4.3 Converting the two-phase mode control signal into a four-phase signal

The Boolean *ModeReq* is a two-phase signal which is high in test mode and low in normal operation mode. Since the whole circuit operates using four-phase signalling a mechanism for converting the two-phase signal into a four-phase one is required (see Figure 9).

In the initial state the toggle element is reset and the inputs of the circuit are low. A rising event on the *ModeReq* input is transmitted to the *GReq* output which is a request for changing the operation mode of the circuit. A rising acknowledge event on the *LAck63* input changes the dotted output of the toggle element to one. As a result, the output of the XOR gate goes low and the *GReq* signal is returned to zero. When the acknowledge signal *LAck63* has returned to zero the toggle element steers a rising event to its output *ModeAck*. Thus, the two-phase signalling protocol is completed along channel *ChMode2* after the completion of the four-phase signalling protocol along channel *ChMode4*.

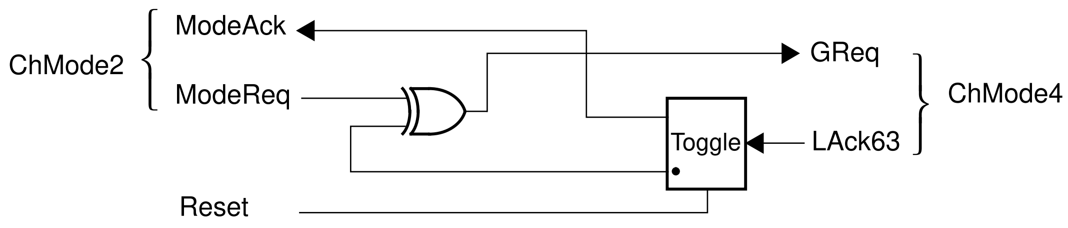

Figure 9 : *Mechanism for converting a two-phase signalling along channel ChMode2 into a four-phase signalling along channel ChMode4*

5. Implementation of the random pattern generator

In the BIST design of the block sorter random pattern test vectors applied to the inputs of the sorting cells (see Figure 4) are produced by the request-driven random pattern generator the implementation of which is illustrated in Figure 10. The core element of the generator is a synchronous linear feedback shift register (LFSR) of the maximum length [MClus86, Russ89].

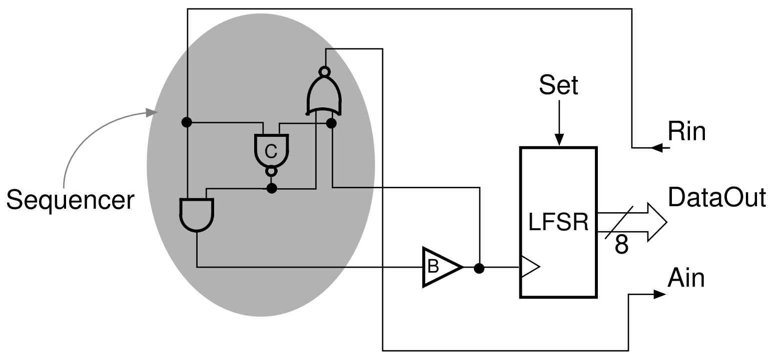

Figure 10 : *A request-driven pseudo-random pattern generator based on the LFSR*

Initially, the LFSR is set to its non-zero seed value. When the request signal *Rin* is set to high the corresponding output of the sequencer goes high (see Figure 10). This signal is buffered to ensure the required drive strength of signals applied to the clock input of the LFSR. As a result, the LFSR goes to its next state changing its outputs. When the negated output of the C-element has been reset the output of the buffer *B* goes low, setting the acknowledge output *Ain* of the generator to high. Once the *Rin* input has been reset the sequencer sets its *Ain* output to low. Thus, a handshake between the generator and the environment is completed.

The random pattern generator used in the BIST block sorter is based on the 16-bit LFSR described by the following derivation polynomial:

$$f(X) = 1 + X^3 + X^4 + X^5 + X^{16}.$$

The period of such an LFSR is equal to $2^{16} - 1$ handshakes between the generator and the environment. Only 8 outputs of the LFSR are used to stimulate the test inputs of the sorting cells.

6. Simulation results and cost comparisons

The original and BIST versions of the block sorter were designed using *Cadence* CAD tools and simulated using *SIMIC* design verification tools developed by Genashor Corporation [Sim94, Ashki94]. All the logic elements of the both designs of the block sorter were implemented on the base of the AMULET low power cell library [EdTR95].

An exact fault simulation analysis of the comparator of the sorting cell (see Figure 3) was carried out with the help of the *SIMIC* fault simulator. As a result, three hard-to-detect faults have been found (r=3) which have the minimal random pattern detection probability (p_d=1.95×10^{-3}). In order to calculate the upper bound of the random pattern test length (L) the following formula can be used [Savir84, Wag87]:

$$L \geq \frac{\ln(q_t/r)}{\ln(1-p_d)}, \tag{1}$$

where q_t is the *a priori* defined escape probability threshold of the random test procedure which is the probability that at least one stuck-at fault from the set of single stuck-at faults will not be detected.

The calculation results showed that L=1741 and L=2920 for q_t=0.1 and q_t=0.99 respectively. The fault simulation results demonstrated that the application of 1000 pairs of 8-bit random test vectors is enough to detect all stuck-at faults inside the comparators of the sorting cells.

The simulation of the BIST block sorter (see Table 1) reveals that the application time of a pair of random test vectors to each sorting cell is equal to 92ns. The area overhead of the BIST design is 15.7% compared with the original version of the block sorter. In normal operation mode, the average dynamic power consumption of the BIST block sorter is equal to 28.4nJ per pair of tests which is slightly more than that of the original design.

The maximum number of extra pins required to implement the block sorter with BIST features is 5:

- 2 pins for the signals *ModeReq* and *ModeAck* of the two-phase channel *ChMode2*;
- 3 pins for the channel along which the test results are observed, i.e., the *MixCmp* output plus a request and an acknowledge signals.

The number of extra pins for the implementation of the result observation channel can be reduced by sharing some of the system channels using select blocks and multiplexers. As a result, the number of extra pins can be reduced to 2.

Table 1 : *Simulation results and cost comparisons*

Block Sorter	NT[a] k	T1[b] ns	PD[c] %	T2[d] ns	No. pairs of tests	SA[e] mm^2	AO[f] %	NEP[g]		PC[h] nJ	
								Max	Min	NM	TM
Original	109.1	66.7	n/a	n/a	n/a	14.50	-	-	-	26.5	-
BIST	126.5	69.7	4	92	1000	16.77	15.7	5	2	28.4	30.0

a. NT is the number of transistors
b. T1 is the minimum application time of one input vector in normal operation mode
c. PD is the performance degradation
d. T2 is the minimum application time of a pair of test vectors in test mode
e. SA is the silicon area
f. AO is the area overhead
g. NEP is the number of extra pins
h. PC is the average power consumption per test in normal mode (NM) and test mode (TM)

7. Conclusions

It has been demonstrated how the BIST technique can be applied to design testable asynchronous circuits with regular structures. A case study of an asynchronous block sorter has been presented. The sorter has been designed using handshake components and post-optimized to achieve a minimal silicon area after its compilation. The structural regularity of the block sorter makes it possible to test all of its sorting cells in parallel sharing a common source of random test vectors. This increases the random test performance of the circuit. A novel technique for changing the operation mode of the BIST block sorter has been presented. The operation mode of the sorting cells of the block sorter is changed asynchronously with the help of an arbiter introduced into the design of each sorting cell to detect its empty state.

Single stuck-at output faults on the control paths and all single stuck-at faults on the data paths of the sorter have been considered. The block sorter is tested for faults on its control lines during normal operation since these faults cause the whole circuit to halt. Stuck-at faults on the control wires, which are not used in test mode, and on the data paths along which the data is transferred are detectable in normal mode by applying two blocks of test vectors. The BIST mode of the block sorter has been designed to test stuck-at faults inside the comparators of the sorting cells and to reduce the test application time.

The BIST version of the block sorter demonstrates relatively small area overhead and low average power consumption per test and requires a few extra pins. Fault simulation results reveal 100% testability of both single stuck-at output faults at the high-level representation of the block sorter and all stuck-at faults inside data processing blocks of its sorting cells.

References

[Ashki94] A. Ashkinazy, D. Edwards, C. Farnsworth, G. Gendel, S. Shikand "Tools for validating asynchronous digital circuits," Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async94), Nov. 1994, pp. 12-21.

[Berk88] C. H. Kees van Berkel, C. Niessen, M. Rem, R. W. J. J. Sages, "VLSI programming and silicon compilation," Proc. ICCD'88, Rye Brook, New York, 1988, pp. 150-166.

[Berk91] Kees van Berkel, J. Kessels, M. Roncken, R. Saeijs, F. Schalij, "The VLSI programming language Tangram and its translation into handshake circuits," Proc. European DAC, 1991, pp. 384-389.

[Berk93] Kees van Berkel, "Handshake circuits. An asynchronous architecture for VLSI programming," Int. Series on Parallel Computation 5, Cambridge University Press, 1993.

[Birt95] G. Birtwistle, A. Davis (Eds), "Asynchronous digital circuit design," Springer, 1995.

[Chan73] T. J. Chaney, C. E. Molnar, "Anomalous behavior of synchronizer and arbiter circuits", IEEE Transactions on Computers, C-22(4), April, 1973, pp. 421-422.

[Dav90] I. David, R. Ginosar, M. Yoeli, "Self-timed is self-diagnostic," TR-UT-84112, Department of Computer Science, University of Utah, Salt Lake City, UT, USA, 1990.

[EdTR95] D. Edwards, C. Farnsworth, "Exploitation of asynchronous circuit technologies", Tech. Report EXACT/MU/Nov/C4, Nov. 1995.

[Farn95] C. Farnsworth, D. A. Edwards, Jianwei Liu, S. S. Sikand, "A hybrid asynchronous system design environment", Proc. 2nd Working Conf. on Asynchronous Design Methodologies, South Bank University, May 30-31, 1995, pp. 91-98.

[FarnTR96] C. Farnsworth, “Tangram optimizations”, Technical Report, Department of Computer Science, University of Manchester, Manchester, UK, to appear in 1996.

[Gars93] J. D. Garside, “A CMOS VLSI implementation of an asynchronous ALU,” IFIP WG 10.5 Working Conference on Asynchronous Design Methodologies, Editors S. Furber, D. Edwards, Manchester, 1993.

[Hauck95] S. Hauck, “Asynchronous design methodologies: An overview,” Proc. IEEE, Vol. 83, No. 1, Jan. 1995, pp. 69-93.

[Haz92] P. Hazewindus, “Testing delay-insensitive circuits,” Ph.D. thesis, Caltech-CS-TR-92-14, California Institute of Technology, 1992.

[Hulg94] H. Hulgaard, S. M. Burns, G. Borriello, “Testing asynchronous circuits: A survey,” TR-FR-35, Department of Computer Science, University of Washington, Seattle, WA, USA, 1994.

[Kess94] J. L. W. Kessels, “Calculational derivation of a counter with bounded response time and bounded power dissipation,” Distributed Computing, 8(3), 1994.

[Khoc94] A. Khoche, E. Brunvand, “Testing micropipelines,” Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94), Utah, Nov. 1994, pp. 239-246.

[Lav93] L. Lavagno, A. Sangiovanni-Vincentelli, “Algorithms for synthesis and testing of asynchronous circuits,” Kluwer Academic Publishers, 1993.

[MClus86] E. J. McCluskey, “Logic design principles: with emphasis on testable semicustom circuits,” Prentice/Hall International Inc., 1986.

[Mead80] C. Mead, L. Conway, “Introduction to VLSI systems”, Addison-Wesley Publishing Company, 1980.

[Pet95] O.A.Petlin, S.B.Furber, “Scan testing of micropipelines,” Proc. 13th IEEE VLSI Test Symposium, Princeton, New Jersey, USA, May 1995, pp. 296-301.

[Pet96a] O. A. Petlin, “Design for testability of asynchronous VLSI circuits,” Ph.D. Thesis, University of Manchester, submitted in Jan., 1996.

[Pet96b] O. A. Petlin, C. Farnsworth, S. B. Furber, “Design for testability of an asynchronous adder”, to appear in IEE Colloquium on Design and Test of Asynchronous Systems, London, UK, 28 Feb., 1996.

[Ron93] M. Roncken, R. Saeijs, “Linear test times for delay-insensitive circuits: a compilation strategy,” IFIP WG 10.5 Working Conference on Asynchronous Design Methodologies, Editors S. Furber, D. Edwards, Manchester, 1993, pp. 13-27.

[Ron94] M. Roncken, “Partial scan test for asynchronous circuits illustrated on a DCC error corrector,” Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async94), Nov. 1994, pp. 247-256.

[Russ89] G. Russell, I. L. Sayers, “Advanced simulation and test methodologies for VLSI design,” Van Nostrand Reinhold (International), 1989.

[Savir84] J. Savir, P. H. Bardell, “On random pattern test length”, IEEE Trans. on Computers, vol. C-33(6), June, 1984, pp. 467-474.

[Scha93] F. Schalij, “Tangram manual,” Technical report UR 008/93, Philips Research Laboratories Eindhoven, P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands, 1993.

[Sim94] “SIMIC: design verification tool,” User's Guide, Genashor Corporation, N.J., 1994.

[Wag87] K. D. Wagner, C. K. Chin, E. J. McCluskey, “Pseudorandom testing”, IEEE Trans. on Computers, C-36(3), March, 1987, pp. 332-343.

[Weste93] N. H. E. Weste, K. Eshraghian, “Principles of CMOS VLSI design: A systems perspective,” Addison-Wesley Publishing Co., 1993.

[Wey93] Chin-Long Wey, Ming-Der Shieh, D. Fisher, “ASCLScan: a scan design for asynchronous sequential logic circuits,” Proc. IEEE Int. Conf. on Computer-Aided Design, 1993, pp. 159-162.