Master Thesis

Design and Functional Test of Asynchronous Processor

Candidate
Andrea Floridia

Advisors
Ernesto Sanchez
Matteo Sonza Reorda

Master in Computer Engineering

Department of Control and Computer Engineering
Politecnico di Torino
Italy, Turin

October 2017

Acknowledgments

This work would not be possible without the support of many people. Foremost among them are my advisors, Ernesto Sanchez and Matteo Sonza Reorda, whom I thank for their relentless encouragement, for the opportunities they gave me, their patience, and for engendered my interest in digital systems and computer architecture, teaching me everything I wanted to know.

To professor Luciano Lavagno and Nikos Andrikos, for the precious advices about desynchronization.

To my friends in Catania, "gli amici del Porconti". Altough we haven't met each other in these two years as we were used to, they always cheer me up whenever we encounter.

To my friends and roommates here in Turin, Dario, Gabriele, Carmelo, Marika, Giuseppe, Danilo, Robert, Luca, Leonel and Pooya who have made my staying here so rewarding. In particular to my roommates, they have been a of source of sanity in an endeavor that has occasionally been far from being sane.

Last but not least, my whole family, who have been enormously supportive even at 1500 kilometers away and always believed in me, even when no one else did.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

Nowadays electronics systems have countless applications. In particular, their application to safety-related field (e.g. automotive) is increasing at a fast pace. Industries constantly demand from electronic manufacturers for faster, less expensive, less power-consuming, and more reliable devices. As a matter of example, microprocessor-based systems are employed in cars for a great variety of applications, ranging from infotainment to engine and vehicle dynamics control, including safety-related systems such as airbag and braking control. The use of microprocessor systems in safety- and mission-critical applications, calls for total system dependability[4].

1.1 Central Processing Unit and VLSI

Central Processing Units (CPUs) are electronic circuits that perform arithmetic, logic, and input/output operations according to the instructions stored in binary format in the so called instruction memory. This memory is connected to the CPU, so that it fetches instructions to be executed. Normally a different memory is used for data to be processed, called data memory. CPUs are the core of each microprocessor or processor based system. The aforementioned memory-CPU architecture is the most widely used today, and it is called Harvard.

CPUs (or processors) are basically synchronous sequential circuits, whose basic element is the transistor. All the activities among the different blocks within the circuit are synchronized by an unique periodic signal, the clock. It is generated by an electronic oscillator, which produces a periodic wave. They have been designed since the early 1960s. In 1965 Gordon Moore, co-founder of Intel, predicted that the number of transistors within an integrated circuit(IC) would doubled every two year. That prediction turned to be true and as soon as the technology improved, it was possible to reduce the size of each transistor, thus the computational power (and frequencies) rapidly increases. From the 1970s it started the so called Very-Large-Scale Integration (VLSI) era, during which thanks to such improvements it was possible to build processors with thousands of transistors and high clock frequencies. More transistors means also complex structures to be fabricated, and in order to manage such complexity designers switched to Electronic Design Automation(EDA) software tools to automate their deigns.

To date, transistors size is in the order of nanometers, and the industrial processes to produce such small devices are rather complex. Furthermore, the physic behind the working principle of transistor of such size is quite complex and is becoming difficult to predict the behavior of the transistors at that scale. Likewise, routing the clock signal to thousands of memory element is not anymore a straightforward task. The main problem comes from the fact that with these conditions is not granted that the clock signals arrives to all blocks of the circuit at the same time, creating synchronization problems. It is mandatory to consider all these uncertainty (even the operating conditions may affect some parameters) during the design phase, that is before the circuit is actually fabricated.

As this problems emerge, several solutions or workarounds have been proposed in order to mitigate such effects from the technology process viewpoint(statistical analysis, topology optimizations, patterning techniques just to cite some of them) and from the design approach point of view. Common practice is to work with pessimistic delays and parameters in order to guarantee that the final system will work correctly, even in presence of variations. As a result, the performance of the modern digital circuits tend to be worse than they actually could be.

1.2 Asynchronous sequential digital circuits

It emerges that design of synchronous circuits is thoroughly accepted since it is fully supported by different CAD tools. In addition, all the computations take place at the clock edges, and it is reasonably easier to think in terms of clock ticks whenever a designer has to design from scratch a digital circuit. However, it is also clear that, despite this easiness in the initial steps, the next steps that lead from the initial design to the final product are becoming quite complex.

In order to overcome the issues that could arise during the whole designs, in the past decades there has been a growing interest in a design technique called asynchronous design. In asynchronous circuits, there is no clock signal, and the state of the circuit changes as soon as the inputs change. Since asynchronous circuits do not have to wait for a clock pulse to begin processing inputs, they can be faster than synchronous circuits, and their speed is theoretically limited only by the propagation delays of the logic gates.

As highlighted by the authors of [19], asynchronous design is not new: some of the earliest processors used clock-less techniques. These techniques have been used in a number of leading commercial processors (Iliac, Iliac II, Atlas, MU-5) and graphics systems (LDS-1). Designers migrated to synchronous design since asynchrony involves difficulties in designing the synchronization mechanism between different computational blocks[5]. Hence, EDA tools do not completely support asynchronous design[5] (essential for designing large-scale circuits).

Nonetheless, asynchronous circuits count with non-negligible benefits such as [34]:

- Circuit speed adapts to changing temperature and voltage conditions rather than being locked at the speed mandated by worst-case assumptions.

- Less severe electromagnetic interference (EMI).
- Lower power consumption because no transistor ever transitions unless it is performing useful computation.
- Better modularity and composability.
- Far fewer assumptions about the manufacturing process are required.
- High performance while gracefully handling variable input and output rates [18].
- Asynchronous approaches have also been explored to handle extreme environments, such as for space missions, where temperatures can vary widely [25].

1.3 Testing of digital circuits

In industry, once the final product is manufactured, it is common practice to verify whether the device is capable of delivering the intended functionalities. For instance, a digital camera should be able to memorize correctly the frames and should be able to acquire as many frame per seconds as specified in the product specifications. The very same approach applies to the vast majority of the manufactured products, also in electronic devices.

Testing of electronic devices is a vast engineering field, and in the last decades is becoming more and more important due to the wide spread applications of electronics products. In particular, testing digital electronic circuit is becoming extremely challenging due to the high number of components (i.e. transistors) composing a single device.

The primary goal of testing a digital circuit is identifying the products that do not behave according to the specifications (i.e. faulty). Defects (or faults) that cause a misbehavior of the product can have different sources:

- Design faults (incomplete or inconsistent specifications, design rules violations).
- Manufacturing faults (interconnection errors, wrong component mounted).
- Manufacturign defects, not directly related to human errors but to the manufacturing process.
- Physical malfunctions (corrosion, electromigration phenomena).

The last three bullets are called physical defects. They can be classified depending on their duration:

- Permanent.
- Intermittent (defect that appear at intervals).
- Transient (defect that appears for a short period of time only)

When devising a proper test strategy for a given product, the engineers avoid to deal directly with the physical defects that may affect the device. Instead, logical faults are often used, which are an abstract model. The way a logical fault models a physical defect (or a set of defects) is called fault model. The most diffused fault model is called single-line stuck-at fault.

Normally, a fault that may affect a device is detected by observing the misbehavior produced by the fault itself. During the test phase, the device to be tested is called Unit Under Test (UUT). The role of a test engineer is to devise a proper set of stimuli (test vectors) to be fed to the circuit, gather the response of the UUT and compare against the good circuit response (faulty-free circuit). The quality of the final product depends on the quality of the test. The metric often used to measure the product quality is the Defect Level[1].

Test vectors are generated and evaluated before the product is actually manufactured. Test vectors must be chosen such that the total test time is minimized. The longer the test, the higher the costs. These vectors are normally generated and evaluated using software tools. The vectors are generated using an Automatic Test Pattern Generator (ATPG), while the effectiveness of such vectors is evaluated using a fault simulator. A fault simulator simulates the behavior of the circuit in presence of faults. Given a set of test vectors and a fault model, the effectiveness of the test vector with respect the fault model is called fault coverage[2].

Once the test vectors are generated (during the test generation phase), they are applied to the UUT during the so-called test application phase. During such phase, the output vectors are gathered and compared with the expected output. The devices to be tested are hundreds (or even thousands) per day. In order to maximize the number of devices tested this phases is normally performed by special machines, the Automatic Test Equipments (ATEs). Automatically, it fed the UUT with the test stimuli, samples the outputs and identify whether the device is faulty or not. Due to their complexity, accuracy and speed the cost of an ATE is quite onerous. It is quite common that it represent the major component in the total cost of the test. Depending on the device to tested, different ATEs are required (ASICs, Memories, Processors, MEMS, Wafers, Boards).

With synchronous sequential circuits, additional circuitry is included in the design in order to improve the test process. The added features make easier to develop and apply manufacturing tests to the designed hardware. These features are called Design For Testing techniques (DFT). Their introduction is made easier thanks to the global synchronization mechanism of the synchronous circuit.

One of the reasons for DFT techniques is that generating test vectors for stuck-at faults for large sequential circuits is practically unfeasible. The reverse is true for combinational circuits. Hence, the idea is to modify the circuits in such a way that during the test phase they turn into pseudo-combinational circuits (by directly

[1]Percentage of faulty devices that pass the test (i.e.,that are erroneously labeled as good). It is measured in parts per million (ppm).

[2]Percentage of detected fault over the total number of possible faults.

controlling the clock signal). This strategy of testing is called scan test. Another popular DFT technique is Built-In Self-Test (BIST). The idea is to move part of the ATE functionalities within the UUT. The test patterns are generated internally and the outputs compared against the good ones. All these operations are driven by a BIST controller. In any case, thanks to DFT is also possible to reduce the cost of test phase, since they normally require a low-cost ATE. Of course, the main drawback is that part of the area is devoted to such circuitry, which cannot be used for any other purposes except for testing the device.

In general, there could be three types of testing:

- Manufacturing Test, aims at guaranteeing the correct behavior of produced devices. It is applied to all manufactured circuits, so has to be minimally costly.
- Incoming Test, Performed by the buyer (e.g., a system company) of a circuit or system to make sure that it works correctly, before it is used. This test relies on limited information (if any) about circuit internal structure.
- On-line test, aims at testing the device when deployed in-field (i.e. during the operational life). It is often performed without removing the device from its operational environment.

1.4 Functional Test

While DFT techniques are essential for achieving high stuck-at fault coverage (e.g. scan test), they are not so effective for other fault models that are becoming more important with new technologies. Moreover, DFT are expensive in terms of area devoted to such circuitry(which are embedded within the final product) and most of them can be used only for manufacturing test.

Modern circuits working frequencies are extremely high, hence it is also important to execute an at-speed test, to detect the faults affecting the timing of the circuit. Executing at-speed test could be extremely expensive, since the external equipment needed for feeding inputs and sampling the outputs of the UUT must be very precise. For these reasons and many others, in the last decades there has been a growing interest in functional test.

Functional test can be defined as a test performed acting on the functional inputs and observing the functional outputs only, without resorting to any kind of Design for Testing mechanism. The test is then developed on the basis of the functional information about the UUT only. It aims at testing the intended functionalities. Currently functional test is very often a step in the test of integrated circuits, electronic boards and electronic systems. Mainly because it covers defects not covered by other kind of test.

Functional test is used for:

- Manufacturing test

- Incoming inspection
- On-line test

Key issues related to functional test are:

- Generate effective test without knowing sufficient implementation details.
- Reducing test time.
- Automation of test generation.

Increasingly often, additional test operations need to be applied during the product's mission life, such as periodic on-line testing and concurrent error detection[4].

Hence the development of suitable functional test strategies for the different types of microprocessor-based systems (both synchronous and asynchronous) is becoming relevant part of the manufacturing process.

1.5 Asynchronous circuits testing

Testing of asynchronous circuits provides both challenges and opportunities which are distinct from the testing of clocked circuits. There has been a body of recent research, for various fault models, test structures, pattern generation, and commercial application. The absence of a global clock means that an asynchronous design cannot be slowed down simply by using a lower clock rate. In addition, many asynchronous datapaths use level-sensitive latches instead of edge-triggered flip-flops assumed by most synchronous test approaches[20]. An advantage of some asynchronous circuits, however, is that they exhibit the useful property of self-checking, entering a deadlock state when subjected to certain stuck-at faults[20].

The authors of [20] propose an overview of the works that have been carried out in the last two decades on the topic. They are mainly related on DFT techniques, which have the drawback of high cost in terms of area.

As already described in previous sections, asynchronous circuits have extremely appealing properties that make them suitable for a number of interesting application (especially reliability-related applications), outperforming their synchronous counterpart. However, designers are discouraged to use them due to the incompatibility with modern EDA tools and the lack of a suitable test strategies (especially for on-line test).

To the best of our knowledge, the work developed during this thesis is the first attempt to develop functional test methodologies targeting on-line test for asynchronous processors. The goal was also to identify a suitable design technique that guarantees the highest degree of compatibility with modern EDA tool, without the necessity of designing from scratch the software tools. This step is needed in order to develop efficient test strategies in short periods of time and to be accepted by the industry.

The next chapters will provide the reader with the essential background needed for fully understand the topics presented. The proposed solutions will be also detailed, along with the experimental results and further future steps to refine the methodology.

Chapter 2

Background

2.1 Synchronous sequential digital circuits

Electronic digital circuits can be distinguished between combinational logic circuit and sequential logic circuit. The former are circuits in which the output signals depend on the current values of input signals only. The latter, the outputs depend both on current and past input values.

In practice, sequential digital circuits are combinational logic circuits with memory elements. The basic memory element is called D latches, as depicted in figure 2.1. They came in different configurations, each of them implementing a specific functionality.

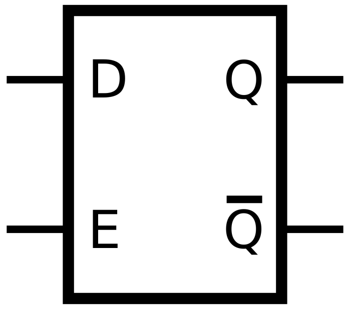

Figure 2.1: Whenever E is set to 1 the latch is said to be transparent and the input D is reflected at the output Q and $\bar{Q}$. When E is set to 0, the latch is opaque and the last value of D is mantained at the output.

The most used memory element is called master-slave edge-triggered D flip-flop (figure 2.2). It is created by connecting two D latches in series, and inverting the enable input to one of them. It is called master–slave because the second latch in the series only changes in response to a change in the first (master) latch[35]. Such connections enable the new data to be stored in the memory element only at the positive edge of the clock signal.

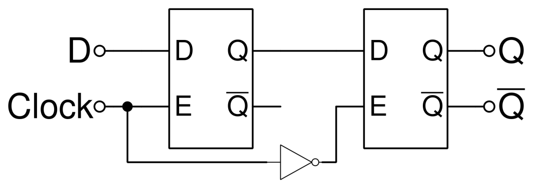

Figure 2.2: Master-slave edge-triggered flip-flop. Note that $\bar{Q}$ signals are left dangling.

In most of the circuits, the same clock signal shared among all the memory elements within the circuit. So changes to the logic signals throughout the circuit all begin at the same time, at regular intervals synchronized by the clock. The changes in signal require a certain amount of time to propagate through the combinational logic gates of the circuit. Engineers usually refer to this as propagation delay. The period of the clock signal is made long enough so the output of all the logic gates have time to settle to stable values before the next clock pulse. As long as this condition is met, synchronous circuits will operate correctly, so they are easy to design. However it is worth to be noticed that the maximum possible clock rate is determined by the logic path with the longest propagation delay, called the critical path. As a consequence, they can be slow and the circuit will spent most of the time being in an idle state, waiting for the next clock edge. Another drawback is that, in large circuits, the widely distributed clock signal takes a lot of power, and must run whether the circuit is receiving inputs or not[34].

2.2 Asynchronous circuits design

Asynchronous systems are typically organized as a set of components which communicate and synchronize using handshaking channels. These channels are defined by two key parameters: communication protocol and data encoding. Building on these fundamentals, complex asynchronous systems can be modularly constructed[19].

2.2.1 Protocol and data encoding

The basic structure of an asynchronous communication channel, between a sender and receiver, is shown in figure 2.3. Ignoring data transmission for now,the channel is typically implemented by two wires: req and ack. Two common handshaking protocols are used to define a single communication transaction: 1) a four-phase protocol (return-to-zero [RZ]), and 2) a two-phase protocol (non-return-to-zero [NRZ], also known as transition signaling). In a four-phase protocol, req and ack signals are initially at zero. The sender initiates a transaction by asserting req, and the receiver responds by asserting ack, in the active or evaluate phase. The two signals are then deasserted, in turn, in the return-to-zero or reset phase. In contrast, in a two-phase protocol, there is no return-to-zero phase: a single toggle on req indicates a request, followed by a toggle on ack to indicate an acknowledge.

Both two-phase and four-phase protocols are widely used, with interesting tradeoffs between them. A four-phase protocol has the benefit of returning interfaces to

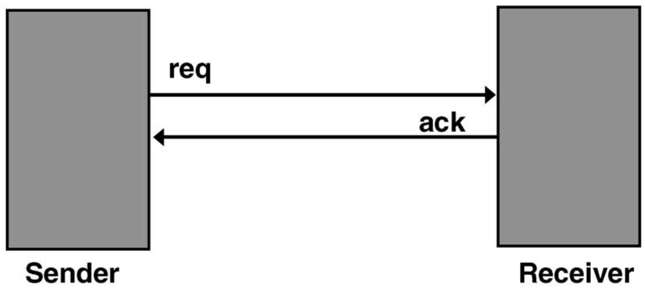

Figure 2.3: An asynchronous channel.[19]

a unique state, i.e., all-zero, which typically simplifies hardware design. However, the protocol requires two complete round-trip channel communications per transaction, which can result in lower throughput. A two-phase protocol may involve more complex hardware design, but only requires one round-trip communication per transaction, which can provide higher throughput.

Once a communication protocol for a channel has been defined, data communication is typically needed. The data itself typically replaces the single req wire in the above example. There are two common data encoding schemes: 1) delay-insensitive (DI) codes, and 2) single-rail bundled data. Figure 2.4a illustrates delay-insensitive encoding on a simple two-bit example. A common approach, dual-rail encoding, is used for each bit, X and Y, which are each encoded using two rails or wires (X1/X0 for X, Y1/Y0 for Y). Assuming a four-phase protocol, all wires are initially zero, and each bit is encoded as 00, representing a NULL or spacer token (i.e., no valid data). A one-hot encoding scheme is used: the transmission of a 1 (0) value on X involves asserting wire X1 (X0) high, and similarly for the transmission on bit Y. Once the receiver has obtained a complete valid codeword, it asserts ack. The reset phase then occurs, where data and ack are deasserted in turn. A completion detector (CD) is used by the receiver to identify when a valid codeword has been received. Dual-rail encoding is widely used [28], [36], [25], and is one simple instance of a delay-insensitive code. In particular, note that, regardless of the transmission time and relative skew of the distinct bits, the receiver can unambiguously identify when every bit is valid, by checking for the arrival of a legal codeword (01 or 10) on each bit. As a result, this approach provides great resilience to physical and operating variability[19].

Figure 2.4b shows an alternative encoding approach, single-rail bundled data. A standard synchronous-style data channel is used, i.e. with binary encoding. One extra req wire is added, serving as a “bundling signal” or local strobe, which must arrive at the receiver after all data bits are stable and valid. Interestingly, the bundled data scheme allows arbitrary glitches on the data channel, as long as data becomes stable and valid before the req signal is transmitted. Typically, data must remain valid from before the req is transmitted to after an ack is received. Therefore, the scheme facilitates the use of synchronous-style computation blocks. It also provides good coding efficiency, with only one extra req wire added to the datapath. However, unlike DI codes, a one-sided timing constraint must be enforced: the req

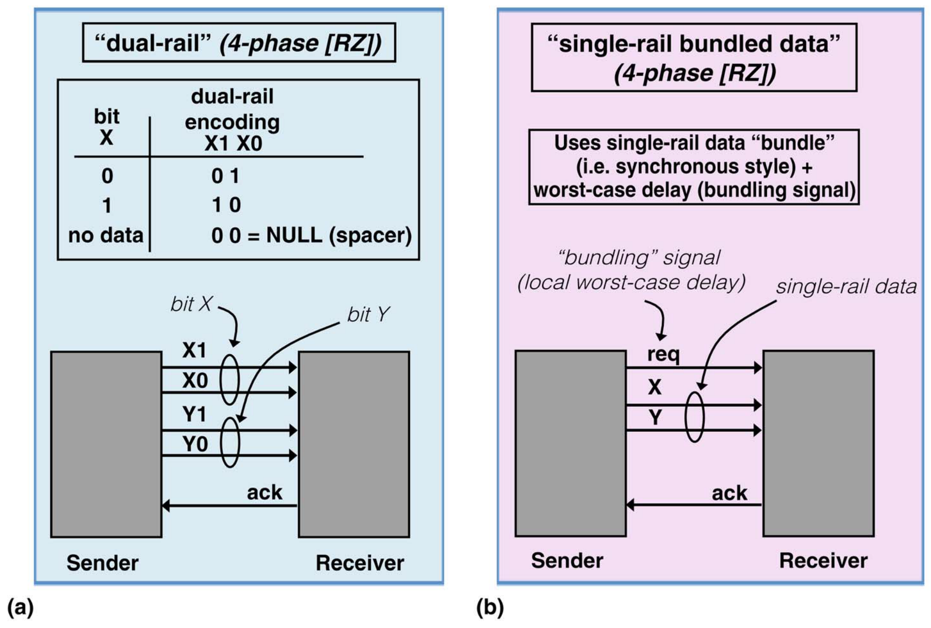
"dual-rail" (4-phase [RZ])
bit X
dual-rail encoding X1 X0
0 | 0 1
1 | 1 0
no data | 0 0 = NULL (spacer)
bit X
bit Y
X1
X0
Y1
Y0
ack
Sender
Receiver
(a)
"single-rail bundled data" (4-phase [RZ])
Uses single-rail data "bundle" (i.e. synchronous style) + worst-case delay (bundling signal)
"bundling" signal (local worst-case delay)
single-rail data
req
X
Y
ack
Sender
Receiver
(b)

Figure 2.4: Asynchronous data encoding schemes. (a) Dual-rail encoding;(b) single-rail bundled data.[19]

delay must always be longer than worst-case data transmission. To support this constraint, a small matched delay is added, either an inverter chain or carefully replicated portion of the critical path. Unlike in a clocked system, though, this is a localized constraint: stages can be highly unbalanced, each with its own distinct matched delay.Moreover, the timing margins can be made fairly tight because some parameters (e.g., process, voltage, temperature) tend to be locally more uniform[19].

In the literature there are several design techniques, most of them analyzed in [19]. However, for the developing this thesis a method called desynchronization [5] has been used. The method has been designed such that it does not require any detailed knowledge of asynchronous circuits, and can be easily automated. Most important, it has been conceived in order to be fully compliant with modern EDA tools. The following pages will be devoted to analyzing the most important parts of the design flow.

2.2.2 Introduction to Desynchronization

Desynchronization incorporates asynchrony in a conventional EDA flow, without changing the "synchronous mentality" or requiring new tools. Both aspects are quite advantageous from several standpoints. First, the notion of operation cycle lives in the subconscious of most circuit designers. Finite state machines, pipelined microprocessors, multicycle arithmetic operations, etc., are typically studied with the underlying idea of operation cycle, which is inherently assumed to be defined by a clock. As an example, one can think about the traditional lecture on computer architecture explaining the DLX pipeline. One immediately imagines the students looking at the classical timing diagram showing the overlapped IF–ID–EX–MEM–WB stages, synchronized at the level of a cycle. It would be very difficult to persuade the lecturer to explain the same ideas without that notion. Secondly, most EDA tools, from logic synthesis to verification, assume a cycle-based paradigm for computation (between clock edges) and memorization (at clock edges), which is very useful to separate functionality (Boolean logic) from performance (timing of longest and shortest paths).

Operation cycles are useful for reasoning and designing. On the other hand, an underlying asynchronous implementation is extremely valuable for the reasons described above. Both these apparently conflicting requirements can be reconciled by using the concept of desynchronization. The essential idea is to start from a synchronous synthesized (or manually designed) circuit and replace directly the global clock network with a set of local handshaking circuits. The circuit is then implemented with standard tools, using the flows originally developed for synchronous circuits. The only modification is the clock tree generation algorithm. With this approach, a design methodology is provided that can be picked up almost instantaneously and without risk by an experienced team[5].

The theory of Petri nets [15] is used by the authors of [5] to provide a formal proof of correctness, in particular the flow equivalence proves that a desynchronized circuit mimics its synchronous counterpart. The asynchronous controllers used in the desynchronization flow are designed starting from Petri nets as well. They are essential in order to implement the asynchronous communication channel, and serve as local clock generators for the blocks composing the system.

2.2.3 Desynchronization flow

The following flow has been successfully used for designing and producing several asynchronous devices [7], [17], [1]. The steps composing the design flow for an arbitrary synchronous circuit are [1]:

1. **Desynchronization region creation**: The synchronous circuit is partitioned into desynchronization regions, each one independently controlled by a an asynchronous controller. The total number of regions determines the performance of the resulting circuit; more regions result in more area overhead, while fewer regions reduce the robustness with respect to local variability and increase electro-magnetic emission. The algorithm is based on the cells connections, initially ignoring all sequential elements of the circuit and grouping together to the combinational logic cells that have a direct connection with each other. Then each sequential element is assigned to the group of the combinational logic cell that drives it. With this step, the regions of the circuit that contribute to the same function are grouped together.

2. **Flip-Flop substitution:** The desynchronization methodology requires a latch-based design to ensure that no data overwriting occurs, as it decouples the satisfaction of setup and hold constraints (by means of matched delay elements and request and acknowledge signals). Thus, any flip-flop in the original design have to be substituted by pairs of latches of equivalent behavior. The best option would be the use dual-clock flip-flops, with separate enable inputs

for the master and slave latches. As technology libraries do not usually contain such cells, separate latches are used introducing some additional area and power overhead. Basically, since any D flip-flop is conceptually composed by a pair of master-slave latches, this steps explicit reveals the internal structure of a D flip-flop.

3. **Control network insertion and matched delay insertion**: Each latch of each desynchronization region is connected to the controller that generates the clock for that region. In particular, the enable signal of each latch is connected to the control output signal of the corresponding controller. The controllers must also be interconnected each other, so that the original data flow of the circuit is respected. Signal connections between desynchronization regions are analyzed, and according to each region dependencies the corresponding controllers are connected each other. The setup constraints are satisfied by delaying the request signal directed to the target controller. Each matched delay must be greater than or equal to the delay of the critical path of the corresponding combinational block (within the desynchronization region). Each matched delay serves as a completion detector for the corresponding combinational block.

Figure 2.5 shows the circuit before and after the desynchronization process.

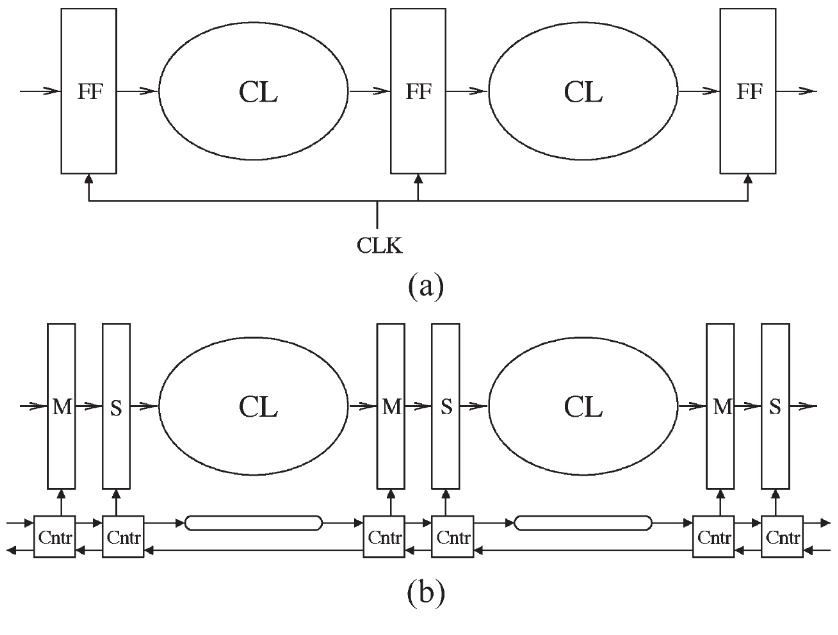

Figure 2.5: (a) Synchronous circuit. (b) Desynchronized circuit.[5]

2.2.4 Desynchronization controller

The controllers used in the desynchronization flow of [5] implements the semi-decoupled four-phase handshake protocol proposed by Furber and Day [8]. Each controller has the following ports: Ri, Ai, Ro, Ao and a control output port. The internal structure of such controllers is depicted in figure 2.6.

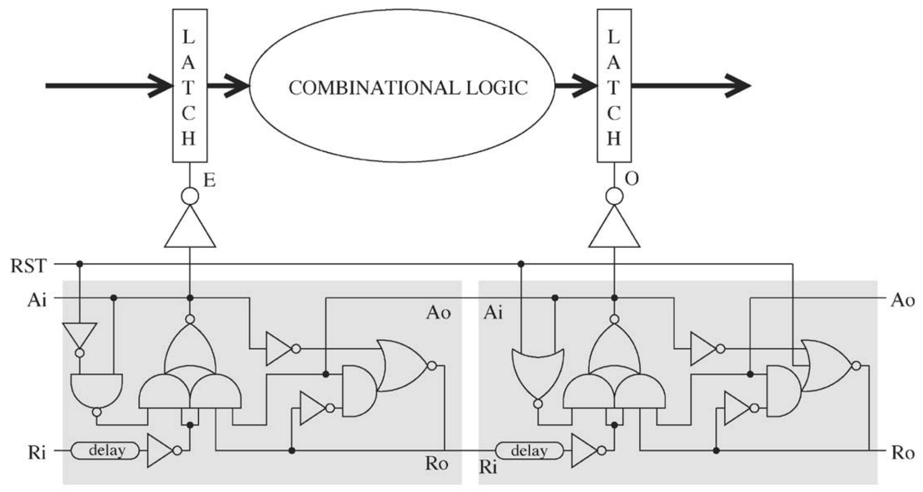

Figure 2.6: In this example, matched delays are embedded with each controller and the combinational logic has been moved between the master and slave latches.[5]

It is worth noting that, in case of multi-inputs/outputs blocks, the req/ack signals of the protocols must be implemented as a conjunction of those coming from the predecessor and successor controllers, by using C-elements. It is a small digital block, widely used in the design of asynchronous circuits. It has been introduced for the first time by Muller [14]. The truth table for a two-input C-element is shown in table 2.1.

x_1	x_2	y_n
0	0	0
0	1	$y_{(n-1)}$
1	0	$y_{(n-1)}$
1	1	1

Table 2.1: Truth table for a two-input C-element.

An example of usage of C-muller is shown in figure 2.8. In [1] has been introduced the delay element of figure 3.5.

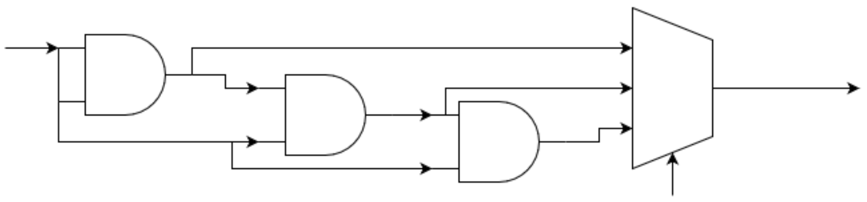

Figure 2.7: Asymmetric delay element.

It is an asymmetric (slow rise, fast fall) delay element. Ideally the logic value 1 is propagated with a delay proportional to the length of the chain, whereas the logic 0 is propagated faster through only a couples of logic gates. Such structure, with the multiplexer driving the output, is used to cope with process variability. As a matter of facts, the multiplexer controls the actual length of the chain to ease post-fabrication circuit tuning (speed-binning).

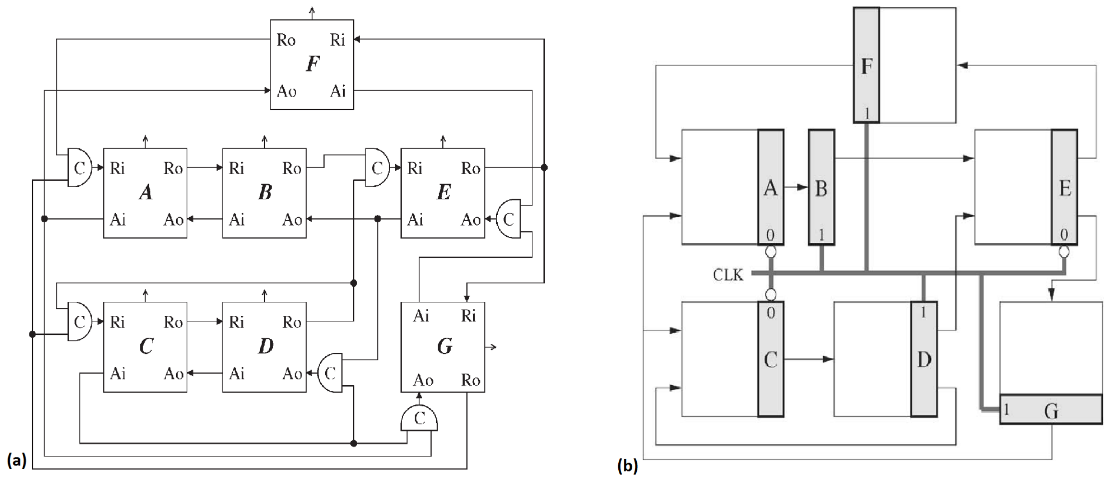

Figure 2.8: (a) Controllers connected each other. (b) The original synchronous netlist.[5]

2.2.5 Testing Desynchronized circuits

Interestingly according to [1], the conventional verification and testing techniques (e.g. DFT) can be applied also to desynchronized circuits, without any particular modification. This is reasonable, since the data flowing from one register to another are the very same of the synchronous one. Desynchronization preserve the structure of the combinational logics blocks and connections surrounding the sequential elements.

As described in details in [5], the datapath can be tested by using scan path insertion with synchronous tools. A clock can be distributed to every register and used only in test mode. An example of such technique can be found in the fabrication of the first prototypes of the ASPIDA project. The microprocessor supports multiplexed clocking, that is the chip can be operated in the fully synchronous mode, or in the desynchronized mode. Clock multiplexing is implemented at the leaf level (at every single latch). In the synchronous mode (used essentially for scan testing), the Master and Slave latches are driven by two non-overlapping clocks from two global clock trees. In the desynchronized mode, the signals that open and close the latches are generated by asynchronous handshaking controllers.

The main drawback is that, with respect to the synchronous version, the area increases due to the multiplexer at every latch.

2.3 Functional test of processors

As already shown in the Chapter 1, functional test is a crucial step in the test of modern Integrated Circuits, being used in different test phases (Manufacturing Test, Incoming Test, On-line Test). An interesting application of functional test is the functional test of processors or processor-based systems.

It requires:

- Providing the processor: 1) a program to be executed and 2) some data to work on.
- Observe the results.

Normally these operations can be achieved with a low cost Automatic Test Equipment (ATE). Several alternatives exists when addressing the test of processors, in particular Scan Test and Built-in Self-Test (BIST).

Scan Test is a mature and well known approach, widely used for manufacturing test. However it has some drawbacks:

- Cannot be used if the netlist is not available.
- Could give low defect coverage (it mainly targets stuck-at faults, only) if the module under test works at high frequency, or with advanced technologies.
- May involve over-testing.
- Could be very long.
- Cannot be used for both incoming inspection and on-line test.

BIST is a very popular method for testing memories, whose basic idea is to move part of the ATE functionalities within the circuit. Embedding this functionalities allow to run the test at-speed, and to used it for the on-line test. However it requires extensive modification of the hardware and adopting a given protocol for test activation and result extraction.

2.3.1 SBST of processor-based systems

A popular functional test technique for processor-based systems is called Software-Based Self-Test (SBST)[24]. The key idea of SBST is to exploit on-chip programmable resources to run normal programs that test the processor itself. The processor generates and applies functional-test patterns using its native instruction set,virtually eliminating the need for additional test-specific hardware. Test-specific hardware such as scan chains might exist in the chip, but SBST normally does not use this hardware. Also, the test is applied at the processor's actual operating frequency. A typical flow for an SBST application in a microprocessor chip comprises the following three steps :

1. Test code and test data are downloaded into processor memory (i.e., either the on-chip cache or the system memory). A low-cost external tester can perform test code and data loading via a memory load interface running at low speed.
2. The processor executes test programs at its own actual speed. These test programs act as test patterns by applying the appropriate native instructions to excite faults.
3. Test responses are uploaded into the tester memory for external evaluation.

Figure 2.9 depicts these steps. As it happens with classical testing strategies, in order to detect any defect, the test program should be able to excite the fault location and then propagate the effect of the fault through the circuit up to a primary output. As a matter of example, consider again the example in figure 2.9. A stuck-at fault is assumed to present in execution unit of the processor. In order to excite the fault, the test engineer have to find the right combination of instructions and data to be processed in order to create a difference with respect the fault-free circuit. The error must be then propagated so that is become visible (internal register are not accessible normally). Hence, suitable instructions that move the content of the registers to the memory are needed.

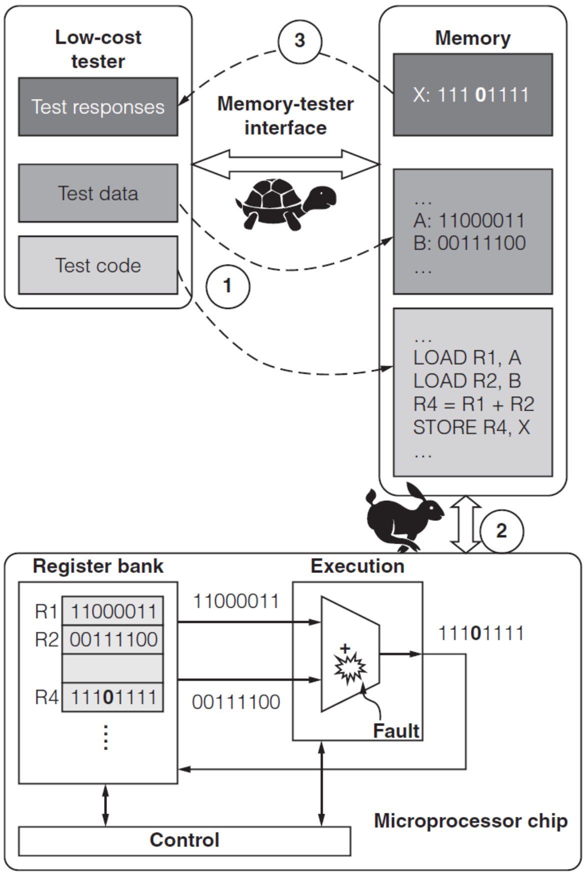

Figure 2.9: SBST flow: 1) Download test code and data, 2) Execute test program and store the results, 3) Upload the results.[24]

Summarizing, key aspects of SBST are:

- Code and Data memory should be available when uploading the test program.
- The test procedure starting method.
- Test result storage and download.

2.3.2 SBST test program upload and result collection

In oder to apply SBST, the processor core should be able to fetch the instructions composing the program from suitable or reserved memory locations. There are two main alternatives for uploading the test program:

- **RAM core:** the test program is uploaded directly in the RAM. It is uploaded every time that its execution is required (RAM is volatile memory). Due to this characteristics, it is suitable for final tests at the end of manufacturing flow.

- **Flash core:** the test program is uploaded only once. When required, they can run at any time. This solution is normally adopted when implementing on-line test of processor-based systems.

If none of the previous storage elements are available, and the processor under test supports only the on-chip cache, the self-test program must be developed so that no cache misses occur during SBST execution. This is called cache-resident testing [3].

Uploading the test program in one of the flash cores available (if any) can be done with the standard interfaces. However, when uploading the test program in the RAM cores several methods can be used:

- **Reusing SoC functionalities:** I/O ports, DMA interfaces, CAN interfaces.

- **Exploiting Dual-port memory:** not always available.

- **Standardized test access mechanism:** IEEE 1500, IEEE 1149.1, SPI, I2C. Normally already available in the SoC, since they are widely used for other purposes.

Along with the execution of the test program, results should be store in a suitable location accessible from the outside. Besides some reserved memory locations, compression circuitries (e.g. MISR) can be used (assuming they are available and accessible from the outside).

2.3.3 SBST for in-field test

Concurrent on-line test approaches are commonly based on redundancy techniques. For noncritical,low-cost embedded applications in which redundancy overhead is cost prohibitive and detection latency requirements are relaxed, an on-line periodic test via SBST provides an adequate reliability level at a low cost. However a key constraint is that the test must be non-concurrent, that is they do not interfere with the microprocessor normal behavior.

Thus the critical aspect of SBST for in-filed test is when the test should be executed. There are basically two alternatives: test at startup and test during normal operations. At startup, the test is executed at the reset (or power-on). Normally there are not particular constraints for this type of test. Indeed, being executed as first task has the advantage that the previous context has not to be preserved and then restored (other tasks are not running yet). Since is the only task, it has almost complete access to system features. If the test is executed during the operations, it means that the current application program is interrupted, and the test is executed. The period of execution should be short enough such that it does not affect the behavior of the system, but long enough to detect possible faults. In [23], the authors studied how the program execution time affect the system performance and the effects of fault detection latency. It has been proposed a strategy based on grouping self-test programs in a single system process. The process is then executed whenever the operating system detect idle periods, in this case the fault detection latency depends on test-program execution time. Alternatively, the test program can be executed at regular time intervals with the aid of programmable timers. The fault latency depends on both the test period and test program execution time. In case of operating systems with hard-real time constraints, the execution time of the test programs is definitely critical than in the normal case. In [10] it has been presented a scheduling algorithm devised for operating system with such characteristics. The scheduling algorithm maximized the effective self-test utilization without affecting the system's real-time requirements.

2.3.4 SBST advantages

There are several advantages when dealing with functional test via SBST. First of all, when targeting Manufacturing Test, a low-cost ATE can be used. The ATE task is simply to upload the test program, monitor its execution, and download the results. Thus high frequencies are not needed for these tasks. The test program is executed by the processor at its operating frequency, hence it is possible to excite and detect many defect that affect the timing of the device. The SBST approach leverages the internal hardware resources of the processor, thus only a limited amount of additional hardware is needed. Finally, test programs can easily modified, therefore the test methodology is extremely flexible.

2.4 DLX ISA overview

The DLX is a popular ISA introduced for the first time in [11]. The DLX is a simple load-store architecture, implemented in pure RISC style. It means that functionalities that are not used frequently are not implemented directly in the DLX, since they are often considered less critical in terms of performances. Normally 32 32-bit General-Purpose Registers (GPRs) are available for the integer operations. They can be loaded with a byte, an halfword (16 bits) and a full word (32 bits). The word is 32-bit long. Along with these 32 GPRs, there are 32 single-precision Floating-Point Registers (spFPRs). There are also 16 double-precision Floating-Point Registers (dpFPRs) which are shared with the spFPRs. Normally register 0 of GPRs is hardwired to zero and cannot be modified, while register 31 is used to store the return address from a subroutine. In addition, three other registers are available:

- **Program Counter:** or PC, it stores the address of the instruction to be retrieved from memory (it is 32-bit long).

- **Interrupt Address Register:** it is made of 32 bits and it stores the return address of the interrupted program, whenever a TRAP instruction is executed.
- **Floating-Point Status Register:** it is used for conditional branching based on the result of Floating-Point operations. It is 1-bit long.

Memory is byte addressable, while halfword accesses are restricted to even memory addresses only. For word memory accesses, memory addresses are divisible by 4 only.

There are 92 instructions divided into 6 classes:

1. Load and store instructions;
2. Move instructions;
3. Arithmetic and logical instructions;
4. Floating-Point instructions;
5. Jump and branch instructions;
6. Jump and branch instructions;
7. Miscellaneous of special instructions;

2.4.1 Instruction Types

All DLX instructions are 32-bit long and are word-aligned in memory. There are three instruction format:

- **I-Type:** operate on data provided by a 16-bit field.
- **R-Type:** operate on data coming from one or two registers.
- **J-Type:** used for the execution of jumps, without using a register operand to specify the branch target address.

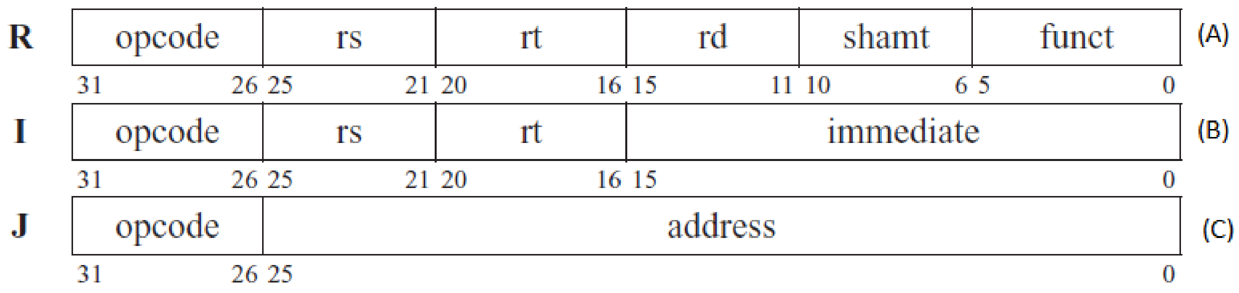

Figure 2.10: DLX instruction formats: (A) R-type, (B) I-type, (C) J-type.

I-type instructions have the format shown in figure 2.10B. This is the format of all load/store instructions, all immediate ALU instructions, conditional branch instructions, JR (jump to address contained in register operand) and JALR (jump to address contained in register operand and save the return address in register 31). Using as reference figure 2.10:

- **opcode:** operation code, specify the instruction to be executed among those avail ables.
- **rs:** source operand for ALU, base address for Load/Store, register to test for conditional branches, target for JR and JALR.
- **rt:** destination for load and ALU instructions, while it is used as source operand for store operations.
- **immediate:** used to compute the address for loads and stores, operand for ALU operations (sign-extended), sign-extension offset added to PC to compute the branch target address for a conditional branch.

R-type instructions are used for register-to-register ALU operations, write to and from special register and moves between GPRs (or FPRs). Figure 2.10A shows the instruction format:

- **opcode:** operation code, specify the instruction to be executed among those avail ables.
- **rs:** first source operand.
- **rt:** second source operand.
- **rd:** destination register.
- **shamt:** shift amount, amount of bits to be shifted.
- **funct:** specify with function the ALU should execute.

J-type instructions is used for implement absolute jumps or jump to procedures (figure 2.10C). The immediate field is a 26-bit signed offset that is added to the address of the instruction in the delay-slot to generate the target address.

The delay-slot is needed since normally the DLX ISA as been designed to be implemented with a pipelined architecture. The main goal of such architecture is to maintain an high throughput, ideally to complete an instruction at each clock cycle. The presence of branch or jump instructions in the program poses several problems to the achievement of this goal. Their outcome will be unknown until the instruction itself exits the pipeline, hence the alternatives is to stall the pipeline (to avoid unwanted instructions to enter the pipeline) or to use the a branch prediction mechanism. The simplest method (implemented also in the DLX) is to leverage the delay-slot(s). Branch instructions are statically predicted as not taken (as in the DLX), hence the next instruction is always executed. It is up to the programmer to insert meaningful instructions in that slot(s) or a NOP(s). Similar strategy is adopted also for jump instruction, even though the outcome is known (jumps are always taken). This is reasonable since even if they are always taken, the target address is not computed instantaneously. Thus, using also in this situation the delay-slot would increase the performances. In the DLX there is only one delay-slot, while in general there are as many delay-slots as the number of pipeline stages that are between the fetch of the branch instruction from the memory and when its outcome is known.

2.5 Fault Simulation

A fault simulator is a software tool that computes the behavior of a given circuit in presence of a given number of faults. The behavior is computed when the circuit is fed with a set of stimuli called test set. Therefore, the inputs needed by this kind of tool are:

- **Circuit description:** for example, a gate-level netlist.
- **Fault list:** is a list of the faults of interest.
- **Test set:** input stimuli normally derived exploiting an ATPG.

A general scheme of fault simulation environment is figure 2.11. Besides the aforementioned inputs, the simulator produces the faulty circuit behavior, the fault coverage obtained and the untested faults.

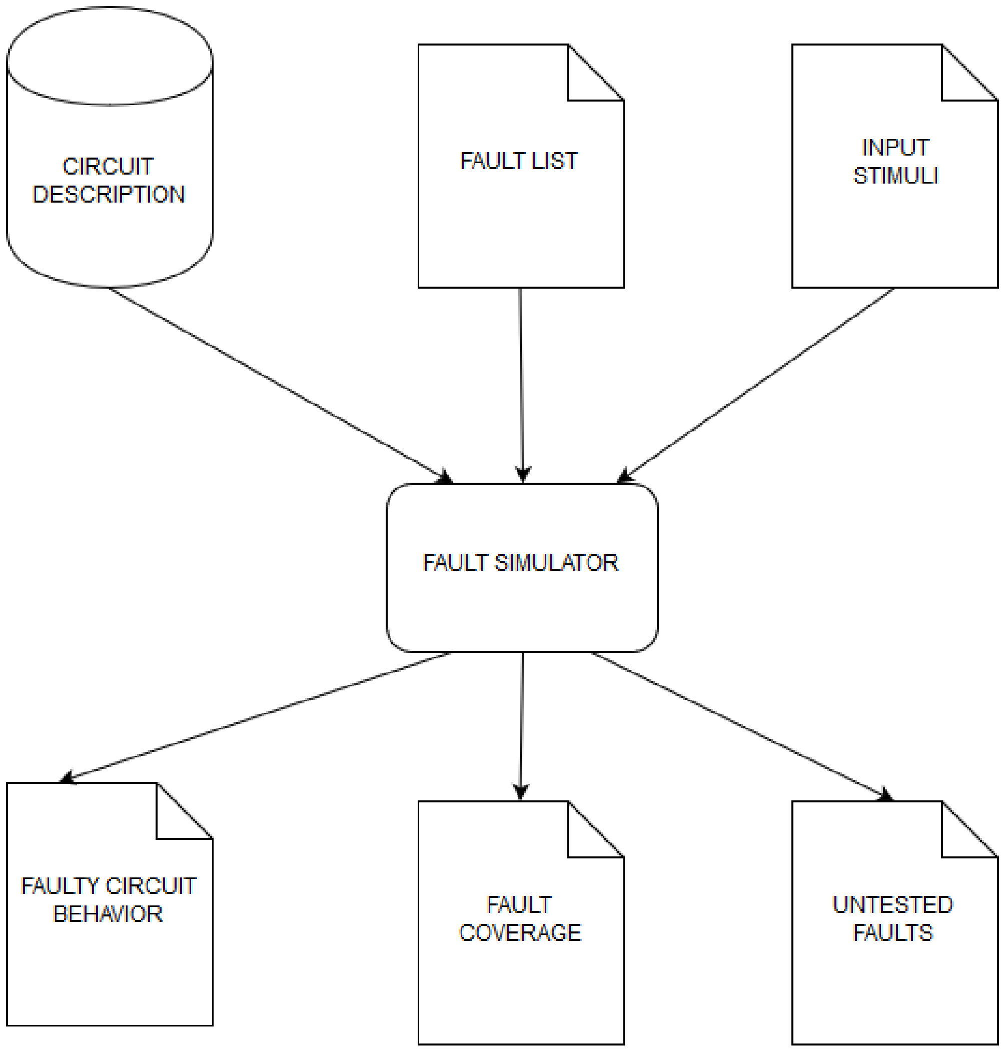

Figure 2.11: General architecture of a fault simulator.

The execution time is a key parameter for evaluating a fault simulator. Generally speaking, a full fault simulation campaign implies a number of simulations equal to the number of faults in the fault list, plus the fault-free circuit. Thus given a circuit with n gates, the simulation of one test vector has a complexity $O(n)$, while the simulation of f faults is also $O(n)$. Hence, since for each vector f faults have to be simulated, the total complexity is $O(n^2)$.

2.5.1 Logic simulation

All fault simulation algorithms are build on the top of those developed for the logic simulation (i.e. the simulation of digital circuits). These algorithms can be distinguished in:

- **compile-driven:** given the circuit function, a program is generated that implement that function. For each input vector, the program is executed from scratch. Instruction are written in the code according to the level of the gate they belong to. It is necessary in order to maintain the correct order of evaluation of the real circuit. Input stimuli can be processed in parallel (Parallel gate evaluation) to improve execution time, since a variable in the computer program can store at least a byte. The idea is to assign a vector to each bit. Despite the high efficiency, there is a limited applicability due to the fact that delays are hard to be modeled and there are huge requirements in terms of memory usage (due to the size of the generated program).

- **event-driven:** for every input vector, the circuit elements evaluated are those for which at least one input has changed. The key idea is to propagate throughout the circuit the events generated on the primary inputs. Whenever an event reaches a circuit element input, its output is evaluated again. The right order of evaluation is kept by using a priority queue to store the elements to be evaluated. The queue uses the level of each element within the circuit as key. At each clock cycle only the minimum amount of elements is evaluated (in compile-driven the whole circuit is re-evaluated). However, there is an overhead due to the priority queue management. Pseudo-code for the event-driven simulation is shown in listing 2.1.

```
while ((gate = extract()) != NULL ) {
  new_val = evaluate(gate);
  if (new_val != value[gate]) {
      value[gate] = new_val;
    for(i = 0; i < fanout[gate]; ++i)
      insert(sons[gate][i]);
  }
}
```

Listing 2.1: Event-driven fault simulation.

2.5.2 Fault simulation algorithms

The techniques for performing a fault simulation are:

- **Serial fault simulation.**
- **Fault Parallel fault simulation.**

- **Approximate fault simulation.**

Serial fault simulation performs simulation with one fault at a time. The pseudo-algorithm is shown in listing 2.2 :

```
simulate_good_circuit();
save_circuit_values();
for (each fault ) {
  force_value(fault);
  event_driven_simulation();
  analyze_outputs();
  restore_circuit_values();
}
```

Listing 2.2: Serial fault simulation.

Fault parallel fault simulation, faults are grouped in sets, each set contains as many fault as the parallelism of the hardware used for simulation. Basically, the idea is to exploit the word-oriented nature of computer operation, in order to simulate more faults simultaneously. The logic operation of a given gate is simulated using the native instructions of the host CPU (e.g OR, AND, etc..), whose normally operate on variables made of more than one bit. The number of faults that can be simulated in a single simulation pass are equal to the word length of the host CPU. Then, at each pass, serial fault simulation is adopted.

Approximate methods are an attempt to reduce the computation power required by the fault simulation. One of this method is called fault sampling. Only a subset of faults is actually simulated, and the obtained results are extrapolated to the entire set of faults. By doing so, an estimation of the achieved fault coverage is provided, but it is difficult to estimate the number of faults to be used before considering the results sufficiently corrects.

Chapter 3

Design of an asynchronous processor

Designing an asynchronous processor has been the first (necessary) step of this thesis. Unfortunately, unlike many synchronous processors, there are not so many asynchronous processors freely available. Indeed, most of the current designs are done by research groups withing industries (e.g. Philips Research Laboratories with the design of a low-power 80c51 asynchronous microcontroller [33]) or by universities (whose projects are quite dated).

Several design techniques have been carefully analyzed. Key parameters for evaluating the techniques were:

- **Compatibility with modern EDA tools:** it is essential in order to develop a functional test strategy. Developing from scratch tools for testing purposes and/or using non-conventional EDA tools it would be too time expensive.
- **Documentation availability:** in order to have a good basis for the design and reproduce the same steps in the minimum amount of time.
- **Production example:** an added value would be the existence of real chip produced (if any).

In the following the design techniques taken into consideration will be presented and analyzed. Advantages and drawbacks of each of them will be also detailed.

3.1 Tangram and VLSI programming

The term VLSI programming was used for the first time in [31]. The design of a VLSI circuit is seen as the development of a program (VLSI program). The language for this kind of programming is called Tangram [30], built on the top of Communicating Sequential Processes (CPS) [12]. The designer can specify whether a given statement has to be executed concurrently or sequentially. Mutually concurrent statements interact by synchronized communications along so-called channels. This synchronization mechanism (at the basis of CSP) is implemented with handshake signaling, the very same mechanism of an asynchronous circuit. The Tangram compiler is in charge for translating the Tangram program into a VLSI circuit. One of the successful applications of Tangram and VLSI programming is the asynchronous microcontroller 80c51 [33]. The techniques is quite attractive, however it has been discarded since is quite dated (1990s) and it is not compatible with modern design flow of Integrated Circuits. Indeed with Tangram a way of design integrated circuit was presented, whose starting point was a Tangram program (hence no HDL or classical synthesis). Furthermore, there is not a clear documentation on how to use a Tangram compiler.

3.2 MOUSETRAP design

It has been introduced by Singh and Nowick in [21]. The proposed architecture is composed of transparent latches and a single gate per latch implementing the latch controller. The simplicity is the key advantages of this architecture, in particular the asynchronous protocol used. Transition signaling is used as handshake protocol (much less overhead with respect to return-to-zero protocol) and level signaling for the latches. Although promising, this approach was also discarded in favor of desynchronization (as already discussed in Chapter 2) since in our opinion the latter would require less effort for designing and testing a processor.

3.3 Delay-Insensitive approach

Delay-insensitive design offer interesting properties, most important high tollerance to process variability. However, it requires a substantial modification of the datapath with respect the synchronous counterpart. The authors of [29] proposed an automatic design flow. However, it is based again on a non-conventional EDA tool and a custom gate library was developed. Hence also this approach was discarded.

3.4 Desynchronization and ASPIDA

Desynchronization has been described in Chapter 2 and further detail can be found in [5]. Desynchronization represents a better fit to the needs of this thesis with respect to the other design techniques. The starting point is a synchronous netlist, derived with classical design style (HDL, logic synthesis, etc.). Then the whole process modifies minimally the original synchronous datapath, adding few logics blocks that are needed for implementing the handshake communication protocol. Last but not least, because there is an exstensive documentitaion provided by the ASPIDA (ASynchronous opensource Processor Ip of the Dlx Architecture) project [7] and the artifact of such project, the ASPIDA chip. According to [7], the main objective of this project was to demonstrate the feasibility of designing and delivering asynchronous Open IPs in a portable, re-usable manner. What is more, the entire project, starting from the HDL netlist up to the post-layout netlist are freely available in [22]. The ASPIDA chip is composed of a DLX microprocessor core, two on-chip memories and a on-chip network called CHAIN [2]. The CHAIN on-chip network architecture was developed at the University of Manchester. It enables interconnection of multiple synchronous IPs onto the same chip. CHAIN consists of multiple wrappers and adapters that guarantee the interconnection of different type of peripherals on the same chip.

3.4.1 DLX architecture of ASPIDA

Including also the on-chip network would be behind the scope of the present work, hence it has been decided to focus only on the microprocessor core. The DLX architecture has been introduced by [11] and the architecture devised with ASPIDA is presented in figure 3.1. It includes the multiplexed clocks and five architectural pipeline stages, four of which actually correspond to circuit blocks (at the circuit level, WB is merged with ID). Each block is controlled by its own latch controller. The arrows of the latch controllers correspond to the Ro and Ao signals, and illustrate the datapath dependencies. Stages ID, EX, and MEM form a ring. ID is the heart of the processor containing the register file and all hazard-detection logic. It also synchronizes instructions leaving MEM (for WB) with instructions coming from IF. Data hazard detection takes place in ID by comparing the output register of instructions in other pipeline stages and their opcodes, and deciding on inserting the correct number of NOPs [5]. The architecture implements the entire integer DLX ISA, that is floating-point instructions are not included.

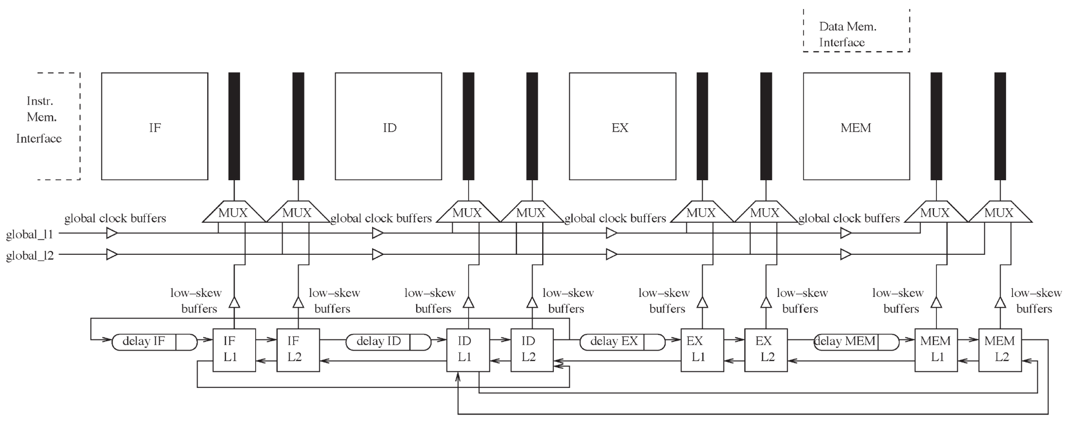

Figure 3.1: Desynchronized DLX core within the ASPIDA chip.[5]

3.5 Developed desynchronization flow

Despite the advantages already listed, the post-synthesis netlist (essential for the testing strategy) provided in [22] are technology-dependent and the technology library used is proprietary and no longer available. Also the desynchronization scripts were missing. Thus, the first step was to derive from the available documentation and papers (e.g. [5]) a suitable desynchronization flow. The final desynchronized DLX differs from the one used in the ASPIDA chip since:

1. there is not any clock multiplexing, it is used only for testing the datapath.
2. the delay elements are not tunable but fixed at synthesis-time, since they are meant for tuning the device at the end-of-manufacturing process.

The proposed desynchronization method starts with a synthesized gate-level netlist, derived from the initial RTL description of the processor with the standard synchronous synthesis methodology. Before analyzing the steps composing it, details on the basic elements implementation (C-element, latches, controllers, delay element) will be provided.

3.5.1 Clocked memory element structure

Each memory element clocked by a controller has the custom structure shown in figure 3.2. The memory element is composed of two latches in series, Master and Slave. Depending on the adopted technology, it is important to specify the polarity of latches reset signal. Since this memory element will replace the flip-flop of the design (implemented with a standard memory cell of the technology library), in order to have the same behavior, the reset signal should behave in the same way in both designs. The control signal of each latch must be exposed to the outside, hence the two ports CLKm and CLKs are needed. The former is the clock directed to the master latch, whereas the latter is the clock directed to the slave latch.

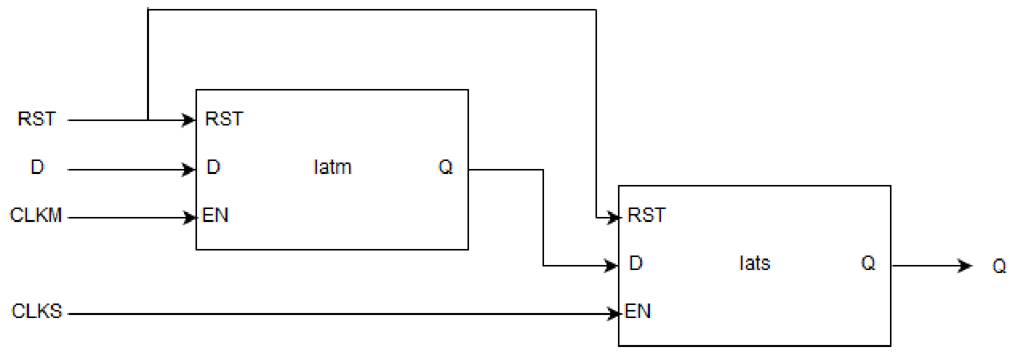

Figure 3.2: Internal structure of the memory element with Latch Master (latm) and Latch Slave (lats).

3.5.2 C-element structure

The C-element is needed since we have multi input/output blocks. There are several possible implementations. It has been used a pure asynchronous implementation, it means that the device is able to “store” previous information using combinational feedback loops with standard logic gates. Once the implementation has been chosen, the C-element structure depends on the circuit structure only. Due to the structure of the DLX, a two input C-element is needed in this case. The internal structure of the C-element is represented in figure 3.3.

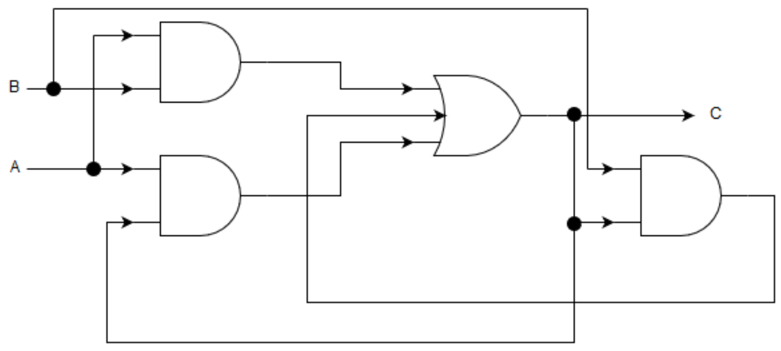

Figure 3.3: Internal structure of the C-element. It composed of an OR gate, and three AND gate. There are two inputs (A,B) and one output (C).

3.5.3 Controllers and delay element structure

The structure of the controllers has been derived from the the one presented in [5]. Each controller is made of two semi-decoupled controllers, one for controlling the master latch and one for controlling the slave latch (figure 3.4). Together form the controller for a desynchronization region.

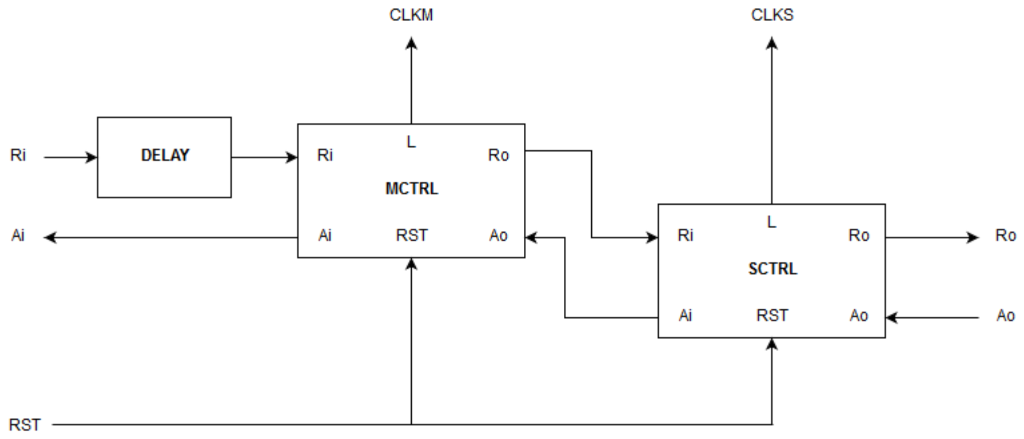

Figure 3.4: Internal structure of desynchronization region controller. It composed of two semi-decoupled controllers, one for the latch master (MCTRL) and one for the slave (SCTRL). The delay element (DELAY) delays the Ri signal.

The delay element is embedded in the controller and feds directly the the Ri of semi-decoupled master controller. In order to be add flexibility the whole controller has been designed to be parametric. The parameter to be controlled at synthesis-time is the number of element composing the delay element, represented in figure 3.5.

3.5.4 Desynchronization regions creation

Normally, for an arbitrary synchronous netlist, the first step would be identify those circuit portions that contribute to the same functionality. However, since this method applies to processor cores only, this step can be simplified and done at design time. This is reasonable since normally processor exploits pipeline in order

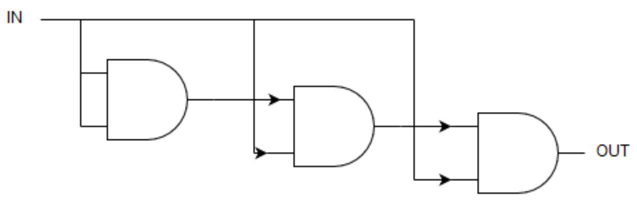

Figure 3.5: Delay element.

to improve throughput. Hence, by structuring the initial synchronous design with a clear hierarchy, it is possible to easly identify the desynchronization regions. In particular, each pipeline stage becomes a desynchronization region itself.

3.5.5 Dependency analysis

Once the desynchronization regions (i.e. pipeline stages) have been identified, the next step is extracting the interdependencies among the different blocks. This step is needed in order to automatically interconnect each other the latch controllers. The algorithm works on a synthesized synchronous gate-level netlist and the pseudo-code is shown in listing 3.1.

```
//outward map
map outward_map;

//inward map
map inward_map;

// outer loop
for (each region of desync_regions) {
  current_region = region;
  // inner loop
  for (each neighbor of current_region) {
    if (exist_outward_path(current_region, current_neighbor)
       ) {
      //add current_neighbor to current_region outward_edges
      outward_map.insert(current_region, current_neighbor);
    }

    if (exist_inward_path(current_region, current_neighbor))
        {
      //add current_neighbor to current_region inward_edges
      inward_map.insert(current_region, current_neighbor);
    }
  }
}
```

Listing 3.1: Algorithm for analyzing dependencies among desynchronization regions.

The algorithm requires a graph-based representation of the desynchronization regions. Each region is a node of the graph and the edges are the connections between the regions. For each node (region), two lists are maintained and updated at each iteration of the algorithm. The first list contains the direct successors of the current node. For example, given two nodes, A and B if it exists and outward edges directed from A to B. At circuit level, it means that it exists a direct path between an output port of A to an input port of B. The second list contains the direct successors. In the specific, two maps are used in order to organize the data, using as key the region and as value the list of successors/predecessors. The algorithms iterates until all possible connections among modules are explored. Given the DLX architecture used in this work, the resulting graph-based representation is shown in figure 3.6.

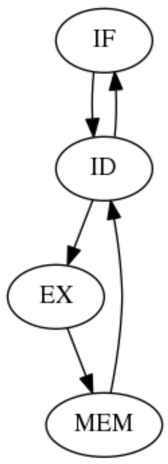

Figure 3.6: Graph representation of the desynchronization regions.

By running the algorithm, the following the lists are produced:

- Outward lists:
  - **IF:** ID
  - **ID:** IF, EX
  - **EX:** MEM
  - **MEM:** ID
- Inward lists:
  - **IF:** ID
  - **ID:** IF, MEM
  - **EX:** ID
  - **MEM:** EX

3.5.6 Timing analysis and controllers generation

Accurate timing information are needed in order to generate properly the delay element to be embedded within the controllers. This step acts on each desynchronization region of the original circuit. After synthesizing each region, any Static Timing Analysis (STA) or synthesis tool (depending on the desired accuracy) can be used for gathering these information.

Once the delay of each region has been measured, the controllers can be synthesized. It is a crucial step, since the controllers are normally implemented with combinational feedback loops, which can cause issues when dealing with synthesis tool for synchronous circuits. The controllers are the very same that can be found in the ASPIA project [22]. Another important aspect are the delay chains. They are based on a redundant implementation, which is normally optimized-out by the synthesis tool. Thus, for avoiding synthesis optimizations both in controller structure and in the delay element, is important to instruct the synthesis tool to perform a simple mapping without acting on the structure of these elements. According to the proposed desynchronization flow, the user can manually specifies the length of the chain for delay element, leveraging the informations provided by the timing analysis and the parametric structure of the controllers. This can be done at synthesis-time, since normally all the parameters have to be fixed before starting the synthesis process.

3.5.7 Desynchronization

Previous steps are summarized in picture 3.7. The step described in this section is the core of the whole design process, the desynchronization. Previous steps outputs become an input for this stage.

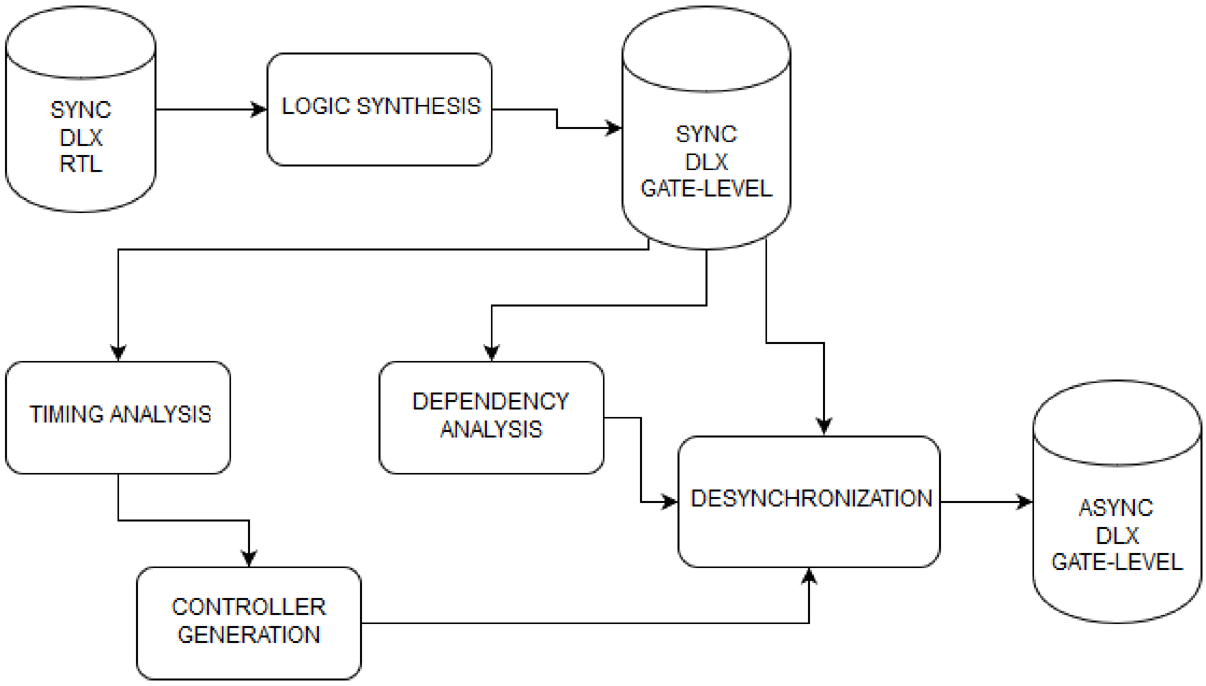

Figure 3.7: The desynchronization flow.

The desynchronization algorithm can be also divided in further steps:

1. **Creation of Clock Master and Clock Slave:** the initial gate-level netlist is modified in order to allow the insertion of the custom clock networks. For each desynchronization region, two input port are added (namely Clock Master and Clock Slave). For each input port that has been created, a net is inserted as well. This net will route the local clock signal from the controller to the region controlled by that controller.

2. **Conversion from Flip-Flop to Latches:** within each region, the synchronous clock tree and the respective port are removed. Then, internally to each region, two nets (Clock Master and Clock Slave) are inserted and connected to the Clock Master and Clock Slave ports. Each sequential element is then removed and substituted with the custom memory element. During the substitution process all the original names and connections must be preserved in order to maintain the equivalence with the synchronous version. Then each clock port (Master and Slave) of the new sequential element is connected to the respective net.

3. **Controllers insertion:** the generated controllers are inserted in the design. Each controller is connected to the respective region through the clock nets, and the reset signal is routed as well. The networks for Ri, Ai, Ro, Ao are generated an left dangling for the moment. Synthesis constraints are also applied to all controllers. In particular, the synthesis tool must be instructed to preserve the hierarchy of the controllers and the internal structure (in particular, the delay chains). Furthermore, the control network is a fully-asynchronous circuit, that is it contains combinational feedback loops. This cycles must be removed from the graph used for the STA. This step is necessary in order to avoid that the synthesis tool cuts the cycles in arbitrary points of the circuit that may not be the one causing the loops. This cause performance penalty only (the circuit will operate correctly). Hence it is necessary to manually specify portions of the controllers to be ignored from the STA tool. It is worth noting that at the end of this process, the controllers are just connected to the desynchornization regions, but not interconnected each other.

4. **Clocks creation:** in order to perform future STA on the netlist and allow synthesis optimizations, each clock generated by a controller is labeled as virtual clock. By specifying these signals as virtual clocks, the back-end tools process the netlist as it was a conventional master-slave latch based design. Therefore, for each region there will be 2 virtual clocks (one for the master and one for the slave). The period of these clock must be non-overlapping (at least at synthesis time). During the real operational life, the clock signal may overlap, but this will not cause any hazard due to the structure of the controllers. In order to ease the synthesis process, the clocks must be labeled as ideal networks, so that the complex computation about signal strength and buffer insertion are skipped and postponed to the place and route phase.

5. **Controllers interconnection:** the informations extracted by the dependency analysis are now used in order to interconnect the controllers. The algorithm is divided in two parts. The first part properly connects the Ao port of the target controller with the Ai ports of the other controllers. The second part does the same, but connects Ri port with Ro ports. Listing 3.2 shows the pseudo-code of the algorithm used.

```
//outward/inward map gathered from the dependency
    analysis
map outward_map;
map inward_map;

// First part

for (each region of desynchronization_region) {

  create C_element_Ao with number of inputs =
      sizeof_list (outward_map.lookup(region));

  for (each element of outward_map.lookup(region)) {
    connect element_controller.Ai to C_element_Ao.in;
  }

  connect C_element.out to region_controller.Ao;

}

// Second part

for (each region of desynchronization_region) {

  create C-element with number of inputs = sizeof_list (
      inward_map.lookup(region));

  for (each element of inward_map.lookup(region)) {
    connect element_controller.Ro to C_element.in;
  }

  connect C_element.out to region_controller.Ri;

}
```

Listing 3.2: Algorithm for interconnecting controllers.

Using as reference the fist part only (the second part is exactly the same), for each desynchronization region a C-element is inserted in the design. The number of inputs of this new C-element depends on the number of regions the current region depends to (basically the size of the associated list). Then, the Ai port of the controller of each region belonging to the list is connected to one of the input port of the C-element. Finally, the output port of the C-element is connected to Ao port of current region controller. The process is repeated for each region. Figure 3.8 shows the resulting interconnections for DLX controllers after running the algorithm.

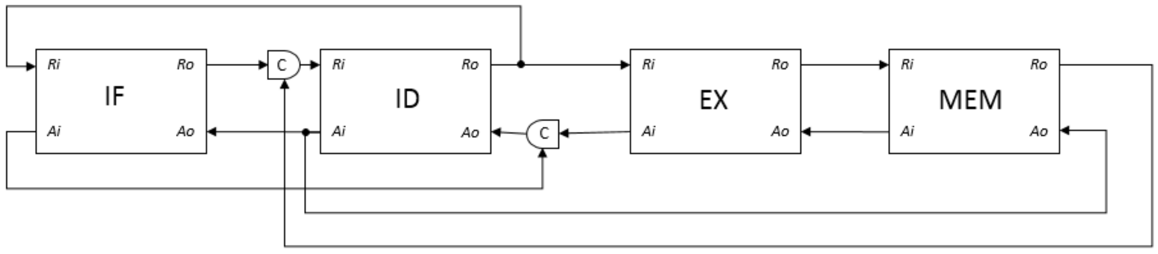

Figure 3.8: Controllers interconnected.

6. **Final synthesis:** the final step is a complete synthesis with the controllers. The synthesis constraints have been applied to controllers (step 3) and to the combinational network as well. After the final synthesis, Standard Delay Format (SDF) file is generated along with the gate-level netlist.

3.5.8 Memories Design and Interface

With synchronous design the processor core interact with the two memories (Instruction and Data Memory) leveraging the clock signal. In particular, the read operation can be done without the necessity of a clock edge, while write operations are completed at the clock edge. In this work memories are assumed to be ideal (zero delay during write and read operations), and are used only for verifying the asynchronous architecture. However, even if they are ideal, suitable scheme has been derived to allow the interaction between the processor and the memories. In particular, write operation happens at the falling edge of clock slave, while the data are read (latched) at the falling edge of clock master. The interaction with instruction memory is identical to the synchronous counterpart, because it is read-only. In order to synchronize the operations with the data memory, since there is not any global clock signal, the clock slave signal is used (namely the one generated by the MEM stage controller). Although physically separated, it has been assumed to be logically driven by the same clock slave. Thus an extra port to DLX interface must be added to allow the routing of the clock signal from the processor to the memory. The designer of the DLX used in this work did very few assumptions on the type of byte ordering used. The DLX operations are totally independent on the type of byte ordering, hence either little endian or big endian can be used. For the development of this work, the data memory has been designed to be little endian compliant.

3.5.9 Post-synthesis Verification

After the synthesis, the asynchronous netlist has been verified by means of post-synthesis simulation. Since the main issues are related to delay elements, the post-synthesis must guarantee that the asynchronous control network has been correctly implemented with the right timing. For doing so, the SDF file with the real gate delay is needed. Typically this kind of simulation is quite slow and requires more CPU time that the classical simulations, but still needed to check the correctness of the implementation. The devised testbench is shown in figure 3.9.

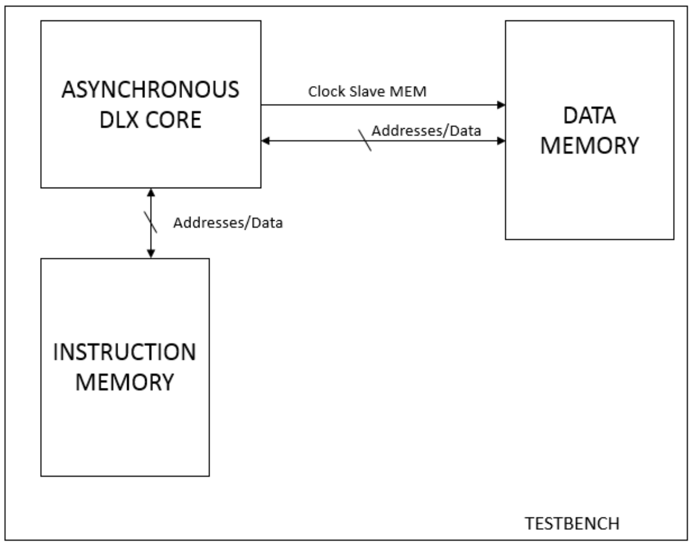

Figure 3.9: Simplified view of the testbench used for verifying the asynchronous DLX.

It includes two memories (Instruction and Data) and the DLX core. The interface with the memories is the very same described in the previous section. The verification flow used for checking the correctness of the design (and its compliance with synchronous counterpart) is:

1. RTL simulation;
2. Gate-level synchronous simulation;
3. Gate-level asynchronous simulation;

The strategy for verifying the correctness of the implementation was to provide to the three versions of the processor (RTL, synchronous and asynchronous gate-level) an assembly program to be executed, and then comparing the obtained results among the three different implementations. A given program is structured such that a set of instructions are executed, and then the results of those instructions are moved to the memory (through a set of store instructions). Seven programs have been developed, each of them targeting a different aspect of the ISA and hazard-handling mechanism. The programs can be divided into:

- **Arithmetic-logic instructions:** given a set of data to be processed, all the (integer) arithmetic operations and the logic operations are executed and the result stored into the data memory.
- **Branch instructions:** the program is made of branch only. To check whether the right flow of instructions is executed, for each possible branch different store instructions are executed. Hence, depending on whether a branch is taken or not, a different data memory content is expected to be found. This verification program aims also at checking the correct behavior in case of control hazards (branch hazards).
- **Jump instructions and procedure call:** the program is very similar to those developed for the branch instructions, but with jump and procedure call

instructions. The very same approach has been also used for verifying the correctness of the instructions flow.

- **Load and store instructions:** the strategy was to perform some arithmetic instructions and then store the result or part of the result (byte,half-word, word) in the data memory. The result is then retrieved from the memory and used for other computations and then stored again.
- **Register File forwarding and hazard detection:** the programs aims at checking whether the data-hazard-detection mechanism have been correctly implemented. In particular, in the DLX architecture, only two types of data hazards have to be checked (namely RAW and WAR).

Chapter 4

Fault simulation

4.1 Challenges of asynchronous circuits fault simulation

Through fault simulation the effectiveness of a given test strategy is normally evaluated. Most of the modern fault simulators use event-driven logic simulation as simulation core, and parallel fault simulation to inject and evaluate the effect of the faults. Further optimizations, oriented to improving the performances, targeting synchronous sequential circuits are usually applied.

Asynchrony poses several issues, namely hazards, feedback loops and so on. As a matter of example, some input values may cause infinite oscillation of a given value, causing infinite execution in zero simulation time. There is a wide consensus that software tools are not easily portable to the asynchronous circuits. This is due to different synchronization methods and, as consequences, different fault effects.

What is more, there is not a unique design approach for asynchronous circuits. Over the years, different design styles have been presented (those described in Chapter 2) based on the underlying timing model. Delay-Insensitive (DI), Quasi-Delay-Insensitive (QDI), speed-independent circuits are just some of the most relevant. For each of them, different architectures have been developed as well. Along with the architectures, custom synthesis flows appeared. Therefore, theoretically, one should develop a fault simulation techniques for each of the aforementioned design styles.

In the literature, there are several example of fault simulation techniques for asynchronous circuits. Authors of [13] have developed a concurrent fault simulator, for testing hyperactive[1] faults. In [27] a gate-level fault simulator for stuck-at and delay faults in asynchronous sequential circuits has been introduced. The simulator is able to handle classical gate, asynchronous blocks (e.g. C-elements), and domino gates. An example of fault simulator targeting speed-independent circuits is [26], while for QDI is [9]. However, none of them is compatible with modern EDA flow, since they require a custom simulation algorithm to be implemented.

[1]Fault that drive the circuit in an unstable state, causing the machine to oscillate.

4.2 Developed fault simulator

To date, the only available fault simulator for the development of this thesis was TetraMax ATPG[2]. Some preliminary experiments on simple asynchronous sequential circuits were performed in order to check whether the tool could be used for running the experiments for the more complex asynchronous processor. A simple 2-bit pipelined full adder has been designed using the design technique MOUSETRAP. However, from the beginning the asynchronous structure of this circuit was rejected by TetraMax. The reason is that TetraMax is not a true fault simulator. Indeed it is mainly an ATPG, designed targeting a specific type of circuits, full-scan sequential circuits or combinational circuits. The fault simulator is just part of the ATPG engine used for verifying the fault coverage of the obtained test patterns. Furthermore, in order to have meaningful logic simulation, real gate delays should be used (namely upload the SDF file). This could be partially done in TetraMax, since that kind of file can be loaded, but actually used for delay fault simulation only.

For these reasons it has been decided to develop a prototypical fault simulation environment, specifically developed for targeting processor functional test via SBST. The architecture is shown in figure 4.1.

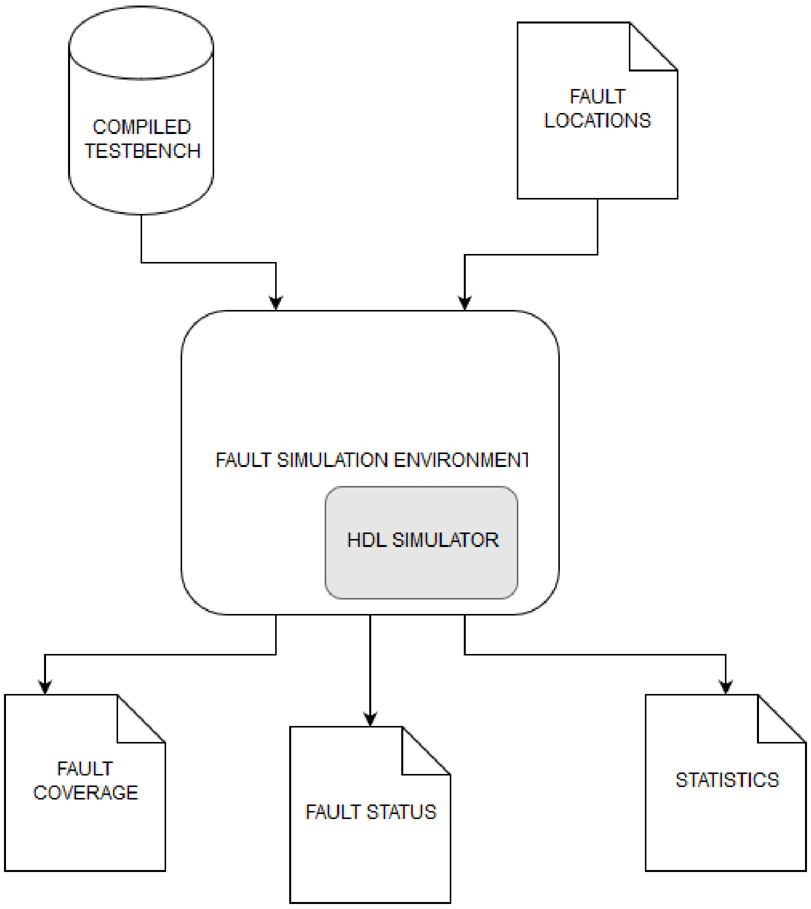

Figure 4.1: Fault simulation environment.

The compiled testbench is the very same testbench shown in Chapter 3 (in-

[2]Commercial ATPG, embedding a fault simulator. Developed by Synopys.

cludes the processor core and the two memories) used for design verification. The data memory has been slightly modified, enabling the dump of its content to a file. Whenever a write operation takes place, the address and data that are going to be written to the memory are also written to a file. This file will be then used for detecting the effect of a fault, and it is inspired to what happens in real-case SBST, that is a fault is detected by examining the content of the data memory. It is not necessary to provide also test stimuli to the fault simulator, the test program (test stimuli in SBST) is embedded into the instruction memory (in turn embedded in the testbench). For a new test program, a new testbench must be generated. It is worth mentioning that the compiled testbench includes also the SDF file with the real gate delays and the technology library. The fault locations contains the locations where the fault have to be injected. It can be generated through a fault manager.

These two inputs are processed by the fault simulation environment. It is build on the top of any HDL simulation kernel. Before starting the fault simulation, the timeout value (i.e. how long each simulation should last) is derived from a fault-free simulation (that should execute the test program). This step, in addition to the fact that each module within the design as an associated delay, is needed for avoiding endless simulation due to possible deadlocks (e.g. infinite computation in zero time caused by a combinational feedback loop). The fault simulation algorithm is shown in listing 4.1 :

```
// list containing the fault locations
list fault_locations;
//list containing the status of each fault
list fault_status;
//data structure containing the simulation statistics
simulation_statistics stats;
// number of detected faults
int detected;

// build the fault locations
build_fault_locations(fault_locations);
// set timeout vaulue
set(timeout_val);
// run fault-free simulation
run_simulation(timeout_val);
gather_statistics(stats);
save_golden_output();
restart();

// fault simulations
for (each location in fault_locations) {
  // stuck-at fault injections
  for (sa_i = 0; sa_i < 2; sa_i++) {
    inject_fault(location, sa_i);
    run_simulation(timeout_val);
    gather_statistics(stats);

    if (!(compare(golden_output, faulty_output))) {
      //fault not detected
      update_fault_status(fault_status, location, sa_i, N);
    } else {
      // fault detected
      update_fault_status(fault_status, location, sa_i, D);
      detected++;
    }
    remove_faulty_output();
    restart();

  }
}

compute_fault_coverage(fault_locations, detected);
```

Listing 4.1: Algorithm for fault simulation.

The first step is, after acquiring the fault locations, the execution of good circuit simulation. Then, good responses are stored for the comparisons during the fault simulations. The simulation environment builds dynamically the fault list at each iteration of the fault simulation loop. For the purposes of this work, only stuck-at faults are considered. At each iterations, the HDL simulator is invoked and the testbench (with the fault injected) is simulated for the amount of time specified in the timeout value. Whenever the circuit is simulated, a dump of data written to the memory is stored into a file by the data memory. At the end of one fault simulation, the new dump file is compared with the golden one (faulty-free circuit), recording whether the current fault has been detected or not. At each iteration it keeps also trace of the number of detected fault, in addition to some useful statistics (namely execution time and CPU time for each simulation). All these information become the simulator output.

The developed fault simulation algorithm belongs to the family of serial fault simulation, since only one fault at a time is injected. Furthermore, the output is compared only at the end of each simulation. The underlying simulation algorithm strictly depends on the chosen HDL simulator. Compared to commercial fault simulators, it is expected to be slower. Normally multiple faults are simulated simultaneously exploiting parallel fault simulation techniques and other optimizations are performed which reduce the simulation time. However, despite these drawbacks, the advantage of this approach is that it allows the usage SDF fault for testing stuck-at faults. Normally simulations with SDF file tend to be slower due to the higher computational power required. But, in order to develop a functional test strategy, the simulator should mimic the real behavior of the circuit. Applying this reasoning to asynchronous circuit, it means that real delays have to be included. So for long fault simulation campaigns on a synchronous processor or circuits it would not be the best choices, but for a preliminary evaluation of the devised test strategy for specific modules within the an asynchronous processor it is a good fit.

Chapter 5

Test program

In order to develop a first functional the test strategy for modules embedded within the asynchronous processor core and to characterize (namely measure the execution time) the developed fault simulation environment, it has been decided to run experiments on the register file memory element, which in general represent a good amount of the total sequential elements of the processor and thus it represent a satisfying benchmark for evaluating the overall performances. Moreover, the register file represent a crucial module for each processor, both synchronous and asynchronous. A second experiment consider instead the whole set of sequential elements (including the register file memory elements). For doing that, a test program is required.

5.1 March algorithms overview

The developed test program specifically target stuck-at faults of the register file and it is inspired to the well-known March algorithm MATS+[16]. The theory and efficiency of March algorithms in general has been documented in [32]. March algorithms have been developed for testing memories, whose structure is regular. A testing strategy based on a March algorithm is called March test. It consist of a sequence of March elements, each of them is made of a sequence of operations to be performed on each memory cell before moving to the next cell.

The order in which cells are tested is determined by the address order of the chosen algorithm. In general, such order can be:

- **Increasing:** normally denoted with a $\Uparrow$ in the description of the algorithm, in this case addresses goes from cell 0 to cell $N-1$ (assuming N the maximum number of cells).
- **Decreasing:** denoted with a $\Downarrow$, is the opposite of the previous order.
- **Indifferent:** can be any of the previous, denoted with a $\Updownarrow$.

Given a cell, the following operations can be performed (with the relative symbols used):

- write a 0 (w_0).
- write a 1 (w_1).

- read a 0 (r_0).
- read a 1 (r_1).

A full March test is delimited by a {...}, whereas a March element is delimited by (...). Depending on the target faults, it is possible to identify a set of conditions that a March test must satisfy to detect them.

The MATS+ algorithm has a complexity equal to $5n$, since the number of operations performed on each cell are five and n is the number of cell composing the memory. It has been demonstrated [32] to be the shortest test able to detect all the Stuck-At-Faults (SAFs) and Address decoder Faults (AFs). The algorithm is:

- **M1:** $\Updownarrow (w_0)$
- **M2:** $\Uparrow (r_0, w_1)$
- **M3:** $\Downarrow (r_1, w_0)$

It is composed of three March element (denoted with **M1-3**). The fist March element writes 0 to each cell in any order. The second, for each cell, reads the content of the cell and expect to find a zero (written in the precedent March element) and then it writes a 1. Cells are scanned in increasing order. The third one is the opposite of the second. Scanning the cells in different order and read/write operations on the same cell ensure is necessary in order to detect AFs, while writing and reading the opposite value the SAFs.

The previous algorithm is valid for memories whose cells are composed of a single bit only. They can be applied to m-bit memories, by extending the 0 or 1 value to m bits. However, whenever the test targets the coupling faults between bits in the same word a modification of the algorithm is required. Since the MATS+ is not a good fit for that class of faults (other algorithms outperform it) and it is not necessary for the comprehension of the developed test program, the extension to deal with coupling faults is neglected.

5.2 Developed test program

Starting from the algorithm of MATS+, an algorithm for testing the register file has been developed. The target faults were the stuck-at faults of each cell (i.e. the memory cell holds only one binary value). Listing 5.1 shows the developed algorithm. Since the address decoder is assumed to be fault free, the scanning oder is indifferent and is not necessary to read and write the same cell before acting on the following cell (unlike the MATS+).
The first phase consists in moving the content of each general purpose register to the data memory. Normally (unless faults in reset network) each sequential element is initialized during the reset phase of the processor core and the surrounding logic blocks (e.g. addresses decoder) are assumed to be fault free, therefore is not necessary to write all 0s in each register. After this step, the data memory locations are expected to store words containing 0s.

```
// first phase
for (each GP_register in GP_register bank) {
  move GP_register to data memory;
}

//second phase
for (each writable_GP_register in GP_register bank) {
  write all 1s in writable_GP_register;
}

//third phase
for (each GP_register in GP_register bank) {
  move GP_register to data memory;
}
```

Listing 5.1: Algorithm testing SAFs in General Purpose register file.

During the second phase, a word of all 1s is moved in each register (again never mind the scanning order). This step applies to the writable registers only. If the goal is to test also whether each register can be correctly accessed (or not accessed), then also the writable registers must be included. Finally, the content of each register is moved again to the data memory. Listing 5.2 is an example of such algorithm using the instruction available in the DLX ISA.

```
; ---------------------
; Phase 1
; ---------------------
sw 0(r0),   r0
sw 4(r0),   r1
sw 8(r0),   r2
sw 12(r0),  r3

; ---------------------
; Phase 2
; ---------------------
addi r1, r0, 0xFFFF
addi r2, r0, 0xFFFF
addi r3, r0, 0xFFFF

; ---------------------
; Phase 3
; ---------------------
sw 16(r0),  r0
sw 20(r0),  r1
sw 24(r0),  r2
sw 28(r0),  r3
```

Listing 5.2: Example of the developed test program applied to registers 0 to 3. Register 0 cannot be written according to the ISA.

Chapter 6

Case of study and experimental results

With this chapter the intent is to provide the implementation of the concepts shown in previous chapters, along with the experimental results gathered. As already mentioned in the Chapter 3, the starting point was the DLX designed during the ASPIDA project. After its successful desynchronization, two experiments with the proposed fault simulation environment were performed in order to asses the performances of the framework and provide a first methodology for testing via SBST modules of an asynchronous processor.

6.1 Logic synthesis and desynchronization

The initial synchronous DLX (written in Verilog) was synthesized with a 65nm CMOS technology library provided by ST Microelectronics using commercial synthesis tool Synopsys Design Compiler. The entire desynchronization flow (Chapter 3) has been implemented in 4 TCL scripts, in approximately 550 lines of code. The scripts were then executed on Desing Compiler. The additional modules required by the flow (namely asynchronous controllers, custom sequential elements and so on) were written in Verilog (the same HDL used for the DLX). The timing analyses were done on Synopsys Prime Time (commercial STA tool). However, for sake of completeness the same timing results were obtained using also Design Compiler, but Prime Time performs faster.

The dependency analysis was implemented on Design Compiler as well. The tool uses a graph-based representation of any design, and the algorithm detailed in Chapter 3 was implemented using the available commands provided by the tool for analyzing the various paths in the design. All the steps included in the desynchronization core (e.g. flip-flop substitution, controllers insertion) were implemented using the commands for manipulating the netlist in memory. The final post-synthesis asynchronous design has been verified using Modelsim.

Table 6.1 shows the result after a basic logic synthesis (i.e. without any particular optimization). The clock period is remained essentially the same, since desynchronization does not affect the combinational part of the circuit and hence the critical path. Asynchronous designs count an area overhead with respect to the synchronous ones mainly due to the control network. Indeed, more logic gates are found in the final netlist. It is worth noting that only few of them (451 gates) are due the control network, while the remaining are due to the substitution of the flip-flops with the latches. Each flip-flop is substituted with two latches, hence, the total number of basic sequential element is doubled. But, since the basic element of the target technology library implementing the latches does not have a built-in reset (as it happens with flip-flop), the synthesis tool infers the needed logic gates for implementing it. As result, additional logic gates are inserted for each latch.

Parameter	**Synchronous DLX**	**Asynchronous DLX**
Logic gates	6349	16919
Sequential elements	1603	3206
Clock period	5	5

Table 6.1: Result of a basic logic synthesis against the synchronous version. The clock period is expressed in nanoseconds.

The netlist obtained with the above synthesis is useful for performing experiments and debugging, however the area overhead is not negligible. Further syntheses have been performed targeting area optimizations, with the goal of combining the additional logic inserted due to the latches with the logic inherited by the flip-flops. The result of such design space exploration is shown in table 6.2, which shows the best implementation in terms of area. It has been obtained by completely removing any form of hierarchy in the design, and enabling area-oriented optimizations of the synthesis tool. Indeed, by removing the hierarchy, logic synthesis algorithms have more degree of freedom for merging together the redundant logic gates and thus reducing the overall number of gates.

Parameter	**Synchronous DLX**	**Asynchronous DLX**
Logic gates	6349	9510
Sequential elements	1603	3206
Clock period	5	5

Table 6.2: Result of the optimized logic synthesis against the synchronous version. The clock period is expressed in nanoseconds.

6.2 Fault simulation

The very same HDL simulator (Modelsim) used for the verification of the asynchronous processor was used for implementing the fault simulation environment. The data memory, being part of the testbench, was written in Verilog as well. The functionality of dumping its content to file was implemented by using a Verilog system task. The creation of such file is always guaranteed, since an initial block statement was added in the memory description that, at the beginning of each simulation, creates the dump file.

The fault simulation algorithm was implemented again using TCL, since Modelsim embeds a TCL interpreter that can be used for issuing commands. The fault injection is done by using the Modelsim commands for manipulating the value of nets or port within the design. For the two experiments the fault locations were generated using Design Compiler, which allows to retrieve the name of a given standard cell or module in the design.

6.3 Experimental Results

The first experiment, for which the test program has been developed, address the fault simulation of the stuck-at faults within the cells of the register file. In the second experiment, all the sequential elements within the processor where considered with the same test program.

6.3.1 Fault model

The sequential element within the processor have an internal structure which is quite different compared to the flip-flops. Normally, flip-flop cell implementation depends on technology library used, and only a behavioral description is used for the simulating the gate-level design. The same approach is used for fault simulations as well, and fault locations are normally located at the ports rather then within the cell. The custom sequential elements used in this work have an internal structure which is quite different with respect to a classical flip-flop, not just be cause the two enable signals, but also because the behavioral description of each latch composing is translated by the synthesis tool in a standard sequential element and some additional logic gates, implementing functionalities such as reset and so on. The same approach used for standard cells have been used. Since the intent was to model the inability of a given cell to change its value, the internal faults have been neglected and have been modeled as a SAF at the output port of the sequential element(figure 6.1).

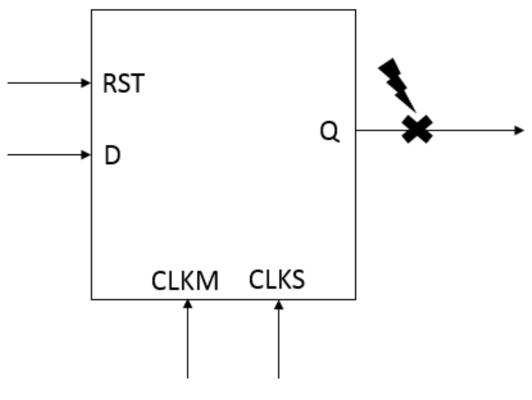

Figure 6.1: Fault model used during the experiments.

6.3.2 Experiments

Before running the experiments with the developed test program, a generic program has been used for both register file and sequential elements. The program of interest is an assembly program which emulates the behavior of a LFSR register via software. LFSRs are digital circuits widely used since they implement pseudo-random number generators in hardware. The maximum number of random numbers is given by the register bitwidth, and normally all the possible numbers (representable on that bitwidth) are generated before starting again from the initial value. Software implementation count several arithmetic-logic instructions (addition, subtraction, shift left/right, and, xor and so on) and branch instructions. In total there are 18 instructions repeated 10 times, since the program is basically a loop, and at each iteration a random number is generated. Let the LFSR generating all possible random numbers would be too time expensive (in terms of simulation time). Thus, the maximum number of random number that the program generates is 10. Although it does not specifically target the test of neither register file, nor sequential element, it represents a generic application program that could be executed by the processor. The results are shown in table 6.3. As expected, the fault coverage is quite low in both cases. This is reasonable, since just a subset of all registers is used and only a subset of random numbers is generated. Moreover, the execution time is extremely high (almost 20 hours, for the memory elements) and these figures discouraged us from executing other general-purpose programs and targeting directly the elements of interest.

Experiment target	**# faults**	**# detected**	**fault coverage**	**CPU time**
Register File	2048	149	7.27	13
Memory Elements	3206	778	24.26	~ 20

Table 6.3: Results of the experiments with LFSR program. Fault coverage is expressed in percentage and CPU time in hours.

Table 6.4 shows the gathered results with the MATS-like test program, which counts 93 assembly instruction. By comparing the two tables, the first interesting result is the CPU time which is greatly reduced (thus, also the time required for executing the test is also reduced) . Secondly, the devised test strategy for the register file has been quite successful. Only 32 faults escaped the test, out of 2048. By analyzing the statistics generated by the fault simulator for the simulated faults, it emerges that the only faults labeled as not detected are stuck-at 0 related to the register 0. This is reasonable, since it is worth to remember that in the DLX architecture the register 0 is hardwired to 0, and the value cannot change. Hence, those faults are untestable via a functional test, since the functionalities offered by the circuit are not modified. With the developed test program, also the stuck-at 1 fault of the flip-flops belonging to the register 0 are detected. Although not writable, the test program uses the register 0 as base address for generating the memory address and for miming the move operations. Therefore, in this case, the faults are detected by observing the mismatch in the addresses and the in the output values.

For the second experiments, although the test program does not explicit target the other sequential elements, it yields a good fault coverage, not so high as in the

Experiment target	**# faults**	**# detected**	**fault coverage**	**CPU time**
Register File	2048	2016	98.43	6.5
Memory Elements	3206	2504	78.10	10.5

Table 6.4: Results of the experiments with MATS-like test program. Fault coverage is expressed in percentage and CPU time in hours.

previous case, but however still acceptable. Only the 22% of the whole set escaped. It is important to note that the test program target the GP registers. Within a processor other registers are present, and some of them cannot be modified directly by the programmer, but are set as consequence of a particular event (e.g. overflow). Thus in order to excite the faults present in those registers, different approach should be used. It should be taken into account that those faults have to be observed as well, hence since they could not be directly accessible, a more complex strategy should be used. Anyway, it quite difficult to develop a general test strategy that targets all the sequential elements (status registers and so on) since it would be strictly processor-dependent. For this reason it has been decided to avoid further investigations on that aspect.

As final remark, the experiments confirmed the initial insights about the fault simulation time. Considering the CPU time in table 6.4, it emerges that each simulation (including the golden run) took approximately 11 seconds (for the MAST-like test program) in a 2GHz quad-core workstation in both fault simulation campaigns.

Chapter 7

Conclusions and future works

In this report it has been proposed a first step towards the functional test of asynchronous processor via SBST with commercial EDA tools (synthesis and testing). First of all, a suitable design technique has been identified among those known in the literature. Criteria for identifying such technique were compatibility with modern EDA tools and real chip manufactured with such technique (if any). Desynchronization is a perfect fit for the above characteristics. It has been conceived to be compatible with EDA tools and there has been real chips produced during the ASPIDA project. The only components which are required by the flow that cannot be designed with conventional tools are the controllers, which are obtained from a Petri nets and then the relative gate-level description via the tool *Petrify* [6]. However, such controllers can be synthesized with any synthesis tool. With such method, an asynchronous DLX processor has been designed, starting from the Verilog description of the ASPIDA DLX core. The final synthesized netlist counts an overhead in terms of logic gates. However, this is mainly due to the flip-flop substitution with latches, since the technology library that has been used does not contains latches with built-in reset. As a consequence, additional logic gates are needed for implementing that functionality.

From the testing viewpoint, a custom fault simulation environment has been developed, to cope with the lack of a commercial fault simulation tool (for the moment) able to fully support asynchronous design. The environment has been used for evaluating the effectiveness of the developed functional test strategy via SBST of the register file of the processor and the whole set of sequential elements. In the former a fault coverage of 98% was achieved, while in the latter 78%. The fault simulation campaigns also confirmed the initial insights about the slowness of the fault simulation environment, which it is a good solution for verifying the effectiveness of test strategy targeting small portions of the circuits, but not enough for the acceptance of the methodology by industries (the fault simulator is not a commercial EDA tool).

The natural evolution of this work is to repeat the same result but with a commercial fault simulator. Insights suggest that a successful fault simulation might depend on the structure of the fault simulator itself. Migrating to a more flexible functional-oriented fault simulator could yield successful results.

Once the fault simulator is identified, the remaining functional blocks has to be tested. Countless functional test algorithms have been developed for synchronous processors (ALU, Decode units, register file forwarding). They reach high percentages of fault coverage in synchronous processors, and whether these algorithms are applicable to asynchronous processors as they are is still unknown.

Within this type of processors there are also other blocks that are quite interesting, that are not found in the synchronous processors. A testing strategy based on functional test of the controllers and control network in general has not been defined yet.

Finally, most of the asynchronous circuits works only under tight timing assumptions. As a matter of example, consider the desyncrhonized processor of this work. Setup constraints of the latches are met thanks to the matched delays inserted that mime the delays of the combinational networks they are associated to. In order to cope with variability, matched delays are designed to be tunable at the end of the manufacturing process. However, in order to tune them, the processor must be tested at its nominal speed. As technology scales, some defects are becoming more evident due to the physical behavior of the transistors which is not fully predictable as it happened with older technology nodes. In particular, delay faults are becoming a real issue.

In order to detect them, the proper paths must be excited to excite the faults themselves and propagate their effects, making them observable. SBST is a good fit since it allows at-speed test with relative low costs, but if the test program is not generated such that it stimulates all the paths that are considered critical, then some faulty devices may be labeled as good when they are not or wrongly tuned. Or, even worse the fault might arise during the lifetime of device, when deployed in-field. In that case, wrong results might be delivered to the user of the device (e.g. braking system of a car). Also here, a suitable in-field test strategy is needed to identify those faults during the self-test routines.

Delay fault testing is an open issue also for synchronous processor. Design-for-Testing circuitries are effective against stuck-at faults, but the same is not true for delay faults and in any case they cannot be used for on-line test. Hence hypothetical test algorithms devised for asynchronous processor might be also adapted to the synchronous counterparts, with undoubtedly benefits for both end-of-manufacturing and on-line test.

Bibliography

[1] N. Andrikos et al. “A Fully-Automated Desynchronization Flow for Synchronous Circuits”. In: *Design Automation Conference* (June 2007).

[2] J. Bainbridge and S. Furber. “Chain: a delay-insensitive chip area interconnect”. In: *IEEE Micro* (2002).

[3] I. Bayraktaroglu, J. Hunt, and D. Watkins. “Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues”. In: *IEEE International Test Conferences (ITC 06)* (2006).

[4] P. Bernardi et al. “Development Flow for On-Line Core Self-Test of Automotive Microcontrollers”. In: *IEEE Transaction on computers* (Mar. 2016).

[5] J. Cortadella et al. “Desynchronization: Synthesis of Asynchronous Circuits from Synchronous Specifications”. In: *IEEE Transaction on computer-aided design of integrated circuits and systems* (Oct. 2006).

[6] J. Cortadella et al. “Methodology and tools for state encoding in asynchronous circuit synthesis”. In: *IEEE Design Automation Conference* (1996).

[7] FORTHICS. *ASPIDA Project.* URL: `http://www.ics.forth.gr/carv/aspida/`.

[8] S.B. Furber and P. Day. “Four-phase micropipeline latch control circuits”. In: *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* (June 1996).

[9] B. Ghavami, A. Tajary, and H.R. Zarandi. “High-level fault simulation methodology for QDI template-based asynchronous circuits”. In: *TENCON 2009 - 2009 IEEE Region 10 Conference* (2009).

[10] D. Gizopoulos. “Online Periodic Self-Test Scheduling for Real-Time Processor-Based Systems Dependability Enhancement”. In: *IEEE Trans. Dependable and Secure Computing* (2009).

[11] J.L. Hennessy and D. Patterson. “Computer Architecture: A Quantitative Approach”. In: *Morgan Kaufmann* (1990).

[12] C.A.R. Hoare. “ Communicating Sequential Processes”. In: *Prentice-Hall Int.* (1985).

[13] A. Lioy et al. “Testing Hyperactive Faults in Asynchronous Circuits”. In: *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing* (1995).

[14] D.E. Muller and P. Day. “Theory of asynchronous circuits”. In: *Report no. 66, Digital Computer Laboratory, University of Illinois at Urbana-Champaign* (1995).

[15] T. Murata. “Petri Nets: Properties analysis and applications”. In: *Proc. IEEE* (Apr. 1989).

[16] R. Nair. “Comments on An Optimal Algorithm for Testing Stuck-at Faults in Random Access Memories”. In: *IEEE Transactions on Computers* (1979).

[17] L. Necchi et al. “Theory of asynchronous circuits”. In: *An ultra-low energy asynchronous processor for Wireless Sensor Networks* (2006).

[18] S. Nowick and M. Singh. “Asynchronous Design - Part 1: Overview and Recent Advances”. In: *IEEE Design and Test of Computers, special issue on asynchronous design* (Oct. 2011).

[19] S. Nowick and M. Singh. “Asynchronous Design - Part 1: Overview and Recent Advances”. In: *IEEE Design and Test* (June 2015).

[20] S. Nowick and M. Singh. “Asynchronous Design - Part 2: Systems and Methodologies”. In: *IEEE Design and Test* (June 2015).

[21] S. Nowick and M. Singh. “MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2007).

[22] OpenCores. *ASPIDA OpenCores.* URL: `http://opencores.org/project,aspida`.

[23] A. Paschalis and D. Gizopoulos. “Effective Software-Based Self-Test Strategies for On-line Periodic Testing of Embedded Processors”. In: *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* (2005).

[24] M. Psarakis et al. “Microprocessor Software-Based Self-Testing”. In: *IEEE Design and Test* (2010).

[25] P. Shepherd et al. “A robust, wide-temperature data transmission system for space environments”. In: *IEEE Aerospace Conference* (May 2013).

[26] F. Shi and Y. Makris. “SPIN-SIM: Logic and fault simulation for speed-independent circuits”. In: *2004 International Test Conference* (2004).

[27] S. Sut-Kolay et al. “Fsimac: A Fault Simulator for Asynchronous Sequential Circuits”. In: *Ninth Asian Test Symposium* (2000).

[28] J. Teifel and R. Manohar. “Highly pipelined asynchronous FPGAs”. In: *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays (FPGA 04)* (Feb. 2004).

[29] R. Thian et al. “An automated design flow framework for delay-insensitive synchronous circuits”. In: *2012 Proceedings of IEEE Southeastcon* (2012).

[30] C.H. van Berkel et al. “The VLSI-programming language Tangram and its translation into handshake circuits”. In: *European Conference on Design Automation* (1991).

[31] K. van Berkel, J. Kessels, and R.W.J.J. Saeijs. “VLSI programming”. In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors* (1988).

[32] A.J. van de Goor. “Testing Semiconductor Memories: Theory and Practice”. In: *John Wiley and Sons, Chichester, England* (1991).

[33] H. van Gageldonk et al. “An asynchronous low-power 80C51 microcontroller”. In: *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems* (1998).

[34] Wikipedia. *Asynchronous Circuits*. URL: `https://en.wikipedia.org/wiki/Asynchronous_circuit`.

[35] Wikipedia. *Latch and Flip-Flop*. URL: `https://en.wikipedia.org/wiki/Flip-flop_(electronics)`.

[36] T.E. Williams and M.A. Horowitz. “A zero-overhead self-timed 160 ns 54 b CMOS divider”. In: *IEEE Journal Solid-State Circuits* (Aug. 1991).