

# Speeding up Fault Injection for Asynchronous Logic by FPGA-based Emulation

Marcus Jeitler, Jakob Lechner  
Institute of Computer Engineering  
Vienna University of Technology  
Vienna, Austria  
{jeitler, lechner}@ecs.tuwien.ac.at

**Abstract**—While stability and robustness of synchronous circuits becomes increasingly problematic due to shrinking feature sizes, delay-insensitive asynchronous circuits are supposed to provide inherent protection against various fault types. However, results on experimental evaluation and analysis of these fault tolerance properties are scarce, mainly due to the lack of suitable prototyping platforms.

Using a soft-core processor as an example, this paper shows how an off-the-shelf FPGA can be used for asynchronous Four State Logic designs, on which future fault injection experiments will be conducted.

**Keywords**—Four State Logic; Asynchronous Design; Fault Injection; Asynchronous Processor Design

## I. INTRODUCTION

Due to shrinking feature sizes and increasing complexity (in terms of area), semiconductors become more and more susceptible to faults. As transient error rates keep growing, the usage of fault tolerance mechanisms for coping with these faults seems inevitable. While in synchronous logic design TMR (triple modular redundancy) is well-known and wide-spread, we want to investigate the ability of asynchronous logic to tolerate transient faults. Four State Logic (FSL) is naturally well-equipped for withstanding upsets because it combines two essential properties: FSL is a delay-insensitive design style and therefore immune to delay-faults. Secondly, data signals are dual-rail encoded, i.e., each bit of data is transmitted by two physical wires. This encoding is needed for achieving delay-insensitivity. Although dual-rail encoding incurs a significant area overhead, it offers inherent redundancy at gate-level in return.

In order to evaluate the robustness of asynchronous circuits, we plan to conduct fault injection experiments. The results of these experiments might provide valuable information about which faults can be tolerated and whether there are structural weaknesses in the FSL design methodology. As test circuit for observing asynchronous behavior in the presence of faults we will use a small processor named SPEAR2, which is a conventional synchronous soft-core processor designed for FPGAs. To facilitate exhaustive testing within a reasonable time frame, we need to employ hardware implemented fault injection rather than simulation-based approaches. This requirement makes FPGAs the most

suitable technology for hosting both the DUT (design under test) as well as the fault injection framework.

Using a soft-core processor as an example, this paper shows how we were able to transform a synchronous circuit into its asynchronous counterpart. As these designs will be used together with our fault injection platform in future experiments, the implementation on standard FPGAs will be addressed as well. Section II gives a short introduction to fault injection placing the focus on HDL-based approaches. The remaining sections consider the asynchronous processor design, starting with an overview of the FSL design style in Section III. The subsequent section introduces our tool chain and explains the asynchronous design flow for FPGAs. Section V then focuses on the processor design itself, describing the specific challenges we had to face when transforming the synchronous VHDL design into its FSL representation (with the help of the available design tools). The results and the conclusion can be found in Section VI and VII, respectively.

## II. FAULT INJECTION

While fault injection in synchronous environments is a well-researched topic, only little information about asynchronous circuits is available. Fault injection tools are usually developed for a single purpose, e.g., a specific system or processor. However, in the course of the RADIAL<sup>1</sup> project a synchronous fault tolerant (TMR) processor shall be compared to its non-fault tolerant asynchronous counterpart. It has therefore been necessary to develop a generic fault injection architecture that supports any arbitrary synchronous or asynchronous VHDL circuit.

### A. Introduction to Fault Injection

The term *fault injection* refers to artificially introducing faults into a DUT in order to test fault-tolerance mechanisms and to assess the robustness of fault-tolerant (FT) systems versus non FT systems. While fault injection itself has evolved over the years and spawned many different techniques, the basic goal is still the same: The injection of faults as well as the observation of their effects. Depending on the actual intention of fault injection, respective tools have

<sup>1</sup>This work is partially funded by the FFG Bridge program: Project “RADIAL” Project Nr: 815458

to cope with completely different requirements. In contrast to an ideal tool which always provides low intrusiveness, high visibility and high performance, available tools are only specialized on a subset of these requirements.

In literature, fault injection is classified as *hardware implemented (HWIFI)*, *software implemented (SWIFI)* or *simulation-based* [1]. This traditional classification is mainly considered with respect to ASIC development. However, due to the increasing interest in HDL languages, which are accompanied by powerful tool chains, an additional type has evolved: HDL-based fault injection. In this paper, we will focus on the latter type of fault injection, as it offers new and promising options. Further information on the classification problem and an advanced metric can be found in [2].

### B. HDL-based Fault Injection

Hardware description languages and their well established tool chains offer sophisticated simulation and analysis capabilities. Furthermore, while the underlying architecture of an FPGA differs from an ASIC, the behavior of the circuit is the same, starting at the RTL level and going up to the system level. Although hardware description languages together with modern FPGA architectures are widely used as prototyping environment, their combined use for fault injection is still uncommon.

Most of the HDL fault injection tools like Mephisto-L [3] use a simulator as execution environment. While these approaches offer great observation capabilities, they can only cover short periods of realtime as the simulation speed decreases with the complexity of the model, the complexity of the workload and the number of injections.

Approaches like the one presented in [4], which speed up the execution by emulating the circuit on an FPGA are often limited in their functionality with respect to the supported fault types and observation capabilities.

### C. FuSE

With FuSE (Fault Injection using SEmulation), which has already been introduced thoroughly in [2], we developed a tool that tries to overcome the limitations of current HDL-based fault injection tools and is able to fulfill the demands made by the RADIAL project. In this context FuSE has already shown its strength with a synchronous soft-core processor that was already at our disposal. However, the required asynchronous counterpart still had to be developed.

In order to derive meaningful results from the fault injection experiments, we had to assure that the structure of the asynchronous version is virtually the same as of the synchronous equivalent. As FuSE uses an FPGA for accelerating the fault injection process, the asynchronous processor description also needed to be synthesizable. The achievable acceleration is a key point of our work, since the performance of a purely simulation-based approach would be unacceptable for such a complex design.

The accurate transformation and the actual implementation of this asynchronous processor will be presented throughout the rest of this paper.

## III. ASYNCHRONOUS LOGIC

### A. Classification

During the past decades, numerous asynchronous design methodologies have been developed. The easiest criterion for categorizing asynchronous circuits is the timing model they employ [5]. E.g., bounded-delay circuits use the same timing model that is applied to synchronous circuits. In this model, it is assumed that gate delays and wire delays are bounded, i.e., the maximum propagation time of a signal is known to the designer. For speed-independent (SI) circuits, gate delays may be unbounded and the interconnect delays are assumed to be negligible. Quasi-delay-insensitive (QDI) circuits further alleviate these assumptions: Both gate and wire delays are unbounded but all forks are isochronic, i.e., the delay difference on forking wires is negligible. Finally, there is the group of delay-insensitive circuits for which no timing assumptions are made at all. Gate and wire delays are considered unbounded. For a more detailed discussion of asynchronous design styles the interested reader may be directed to [5].

### B. Four State Logic

With regard to fault-tolerance, we chose to examine delay-insensitive circuits because of their robust timing properties. Since the transmission of data may be delayed for an unknown amount of time, the recipient needs to perform some form of completion detection on the input data. Obviously completion detection cannot be conducted in the value domain on ordinary binary data words. *Null Convention Logic*, e.g., enables completion detection by separating successive data words with a spacer value, a so-called NULL data cycle [6]. Thus after a valid data value has been processed the entire circuit is reset to a NULL state. The downside of this method is that only every second cycle is a computation cycle. Hence, another possibility is the adoption of 2-phase data encodings such as the *Level-Encoded two-phase Dual-Rail (LEDR)* scheme [7]. In LEDR, consecutive data words are encoded in alternate "phase" representations (even and odd). No reset transitions are needed between cycles. The key concept of LEDR is to extend regular boolean data values to determine whether a certain value is valid in the current context. For representing binary signals in two different phases, a *dual-rail* encoding can be employed, i.e., two physical wires are needed for transmitting a single bit of information. Figure 1 illustrates the LEDR coding scheme and shows the corresponding transition diagram. *Even* phases are denoted with  $\varphi_0$ , *odd* phases with  $\varphi_1$ . Note that for every transition exactly one rail has to change. Due to this property unwanted glitches are prevented.

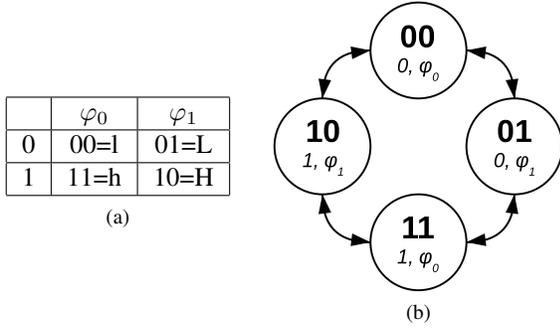


Figure 1: LEDR encoding scheme.

*Four State Logic (FSL)* is a LEDR-based implementation method for delay-insensitive circuits. Another LEDR-based approach similar to FSL is *Phased Logic* as presented in [8]. Compared to Phased Logic we wanted to keep our synthesis procedure as simple as possible, allowing us to generate understandable delay-insensitive circuits that can easily be analyzed in the course of our fault-injection experiments. Thus, FSL circuits only contain a few basic combinational gates and sequential registers, which are synchronized using a simple handshaking protocol.

### C. Combinational Gates

Since all data signals in FSL are dual-rail encoded, suitable gates are necessary that directly operate on LEDR signals. We have designed four basic gates which are the building blocks of the combinational parts of FSL circuits: INV and 2-input AND, OR, and XOR. All combinational gates apply their particular boolean function on the inputs, and output the result in the current phase encoding of the inputs. For 2-input gates it is possible that the inputs differ in their phase. This means that one of the inputs still holds an old data value and therefore must not be combined with the new input value. In this case the old output of the combinational gate must be preserved to avoid unwanted glitches. For saving the previous output memory elements are needed. Figure 2a shows the extended truth table for an FSL AND gate and Figure 2b depicts the respective implementation for FPGAs based on 4-input LUTs.

The (\*) characters in the truth table mark input combinations with inconsistent phases, where the previous output value needs to be preserved. This is done by two RS-latches (one for each rail), each of which can be implemented with a single 4-input LUT. Two inputs are driven by the R and S signals, while the third input is the feedback of LUT's output. The other four LUTs are required for computing the Set/Reset signals of the two latches. Note that these signals will only be active if the inputs A and B have the same phase.

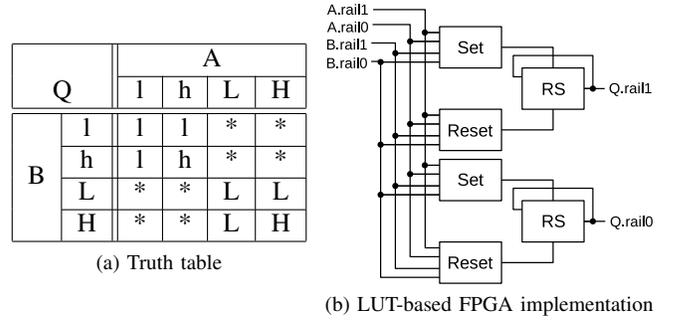


Figure 2: FSL AND gate.

### D. FSL Registers

Like in synchronous logic design, registers are needed for storing intermediate computation values and for building sequential pipelines. Generally speaking, a register is allowed to switch only if new data becomes available from the preceding registers (upstream stages) and if the currently stored data has been consumed by the successors (downstream stages). In synchronous circuits the clock signal marks this point in time. As we have no clock in FSL circuits, the registers are enabled by comparing the predecessors' and the successors' phases (local handshake). The phases of the upstream stages can be directly derived from the transmitted (FSL encoded) data, the phases of the downstream stages are provided by additional feedback (acknowledge) signals. A formal definition of the *firing rule* for FSL registers can be given:

*Definition 1:* A register  $r$  is allowed to replace the currently stored value with its current input value if and only if the following two conditions are met:

- 1)  $\forall w \in N^+(r) : \Phi(w) = \Phi(r)$
- 2)  $\forall w \in N^-(r) : \Phi(w) \neq \Phi(r)$

$N^+(r)$  ... set of successor registers,

$N^-(r)$  ... set of predecessor registers,

$\Phi(r)$  ... current phase of register  $r$

Based on this formal specification we have developed an implementation consisting of FSL latch cells (see Figure 3).

The internal structure of these cells closely resembles the structure of FSL AND gates: In both cases there are two LUTs used for RS latches and four LUTs that drive the Set/Reset signals. In contrast to the combinational gates, the registers' latch cells also need a reset input. The Set/Reset logic is derived from the data input (D) as well as the phase of the successor stages ( $\varphi_{succ}$ ) and the current phase of the register ( $\varphi_{own}$ ). These phase signals are generated by the control logic of the FSL register component (Figure 3b). Thus, the control logic regulates the data flow of the FSL register.

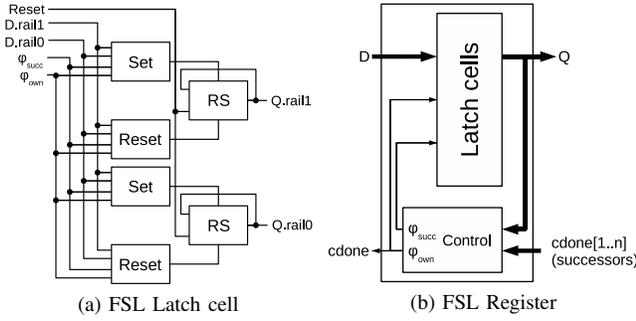


Figure 3: Register implementation.

#### IV. FSL DESIGN FLOW

This section briefly describes the design flow and the respective tool we have developed for implementing FSL circuits. A more detailed description can be found in [9].

The starting point is a conventional synchronous VHDL design. With the help of a synthesis tool (e.g. *Synopsys*) a synchronous gate-level netlist is generated. For this synthesis step a customized target library is used. This library contains gate descriptions restricting the netlist generation to the primitives that can be replaced with the presented FSL components later on.

Subsequently the netlist is parsed by our transformation tool and the circuit is converted into a directed graph, where the nodes represent the registers of the circuit. Edges from one register to another show data dependencies between these registers. The graph is used for all further transformation steps and contains all the necessary information for the (semi-)automated generation of the final FSL design. Only the registers' phase initialization (i.e. the phase encoding of the registers' initial value) and the insertion of buffer registers, in order to break up cycles that would cause deadlocks at runtime, need to be specified by the designer. The phase initialization basically determines which initial values are used for computation and which are discarded. Consider the simple example of two registers  $R_1$  and  $R_2$  in a pipeline. Register  $R_1$  provides data for register  $R_2$ . If the initial phase of  $R_1$  differs from  $R_2$ , the initial value of  $R_1$  will be processed by  $R_2$  after the system reset. In our FSL conversion tool this behavior is represented with tokens drawn on the edges of the register graph.

A problem concerning the assignment of initial tokens arises in the presence of cycles as illustrated in Figure 4. On the left side all registers in the cycle hold useful initial data, thus both edges carry initial tokens. Unfortunately, this situation causes a deadlock since both registers wait for their successor to acknowledge the reception of their initial data. This deadlock can easily be resolved by adding a buffer register to the cycle. Note, that the buffer register is configured without initial token, i.e., it is immediately ready

for new data. Therefore, register  $R_2$  is able to pass on its data value to the buffer register. Afterwards  $R_2$  itself can accept the token at its input. Thus, the three registers get enabled one after another and the deadlock is resolved.

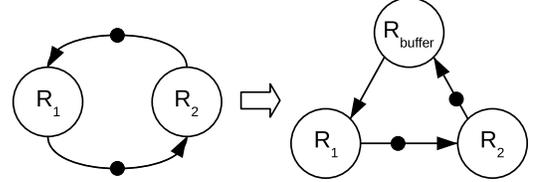


Figure 4: Initial token assignment, deadlock removal.

After the designer has decided about the right token assignment, the asynchronous FSL design is completely defined and can be generated based on the synchronous netlist. The output of this procedure is a netlist built of FSL primitives, that can be processed by conventional place & route tools (e.g. Altera Quartus).

#### V. BUILDING AN ASYNCHRONOUS PROCESSOR

##### A. SPEAR2

The SPEAR2 [10] soft-core processor has been developed at our institute for scientific and educational purposes. It comprises a 16-bit RISC architecture, which executes instructions in a four-stage pipeline. A strong focus has been placed on a modular and configurable architecture. Therefore, the core components (processor core, instruction memory, register file,...) are separated into different modules. For the interaction with its environment SPEAR2 offers an interface for custom extension modules as well as an AMBA interface. An extensive configuration framework allows to adapt SPEAR2 to the requirements of a concrete application.

##### B. ASPEAR2 - Structure

Figure 5 outlines the basic structure of the ASPEAR2 processor, which contains the very same components as the synchronous version.

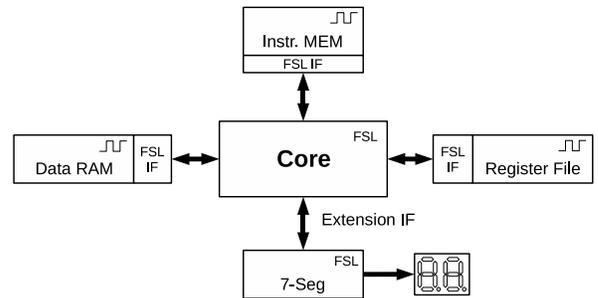


Figure 5: Block diagram of ASPEAR2.

The central block in the middle of the figure represents the processor core unit. This component is a purely asynchronous implementation. Around the core three memory-based blocks are arranged: *Instruction Memory*, *Data RAM* and *Register File*. These modules store data words in the internal RAM blocks of the FPGA. Since most modern FPGAs only contain synchronous memories (we use Altera’s Stratix II FPGAs), we had to keep the original synchronous implementation of the three memory components. This is indicated by the small clock symbol in the right corner of the particular boxes in Figure 5. As a result, the communication of the asynchronous processor core with the synchronous memory components needs to be handled by a special FSL interface, which will be explained in-depth later in this section. Additionally a peripheral module controls a 7-segment display on our FPGA prototyping board. It is attached to the extension interface of the processor and is implemented entirely in FSL logic.

### C. FSL Transformation

With the help of our design flow the transformation of the synchronous processor description into an FSL circuit only takes a few steps. Most of the original synchronous VHDL source files could be used without any changes. The only exception were the memory components which we introduced in the previous section: Due to their particular architecture they have to be handled separately. In order to exclude them from the regular translation process, we created empty stubs with the same interface description, thus, allowing us to connect the original synchronous components to the asynchronous processor core after the transformation has been completed.

The synchronous netlist produced by Synopsys is processed by our FSL conversion tool, where the initial token allocation needs to be done. If the resulting FSL circuit should preserve the behavior of the original circuit, this assignment is straightforward: In a synchronous system the initial value of all registers is passed on to the successor stages. Thus, an initial token is assigned to every register in the asynchronous design in order to retain the semantics of the circuit. For every cycle in the register graph this token placement will cause a deadlock. Therefore, buffer registers have to be inserted. Since tracking down all cycles manually can be tedious, our tool is able to detect cycles with deadlocks.

### D. Interfacing Memory Components

As explained in the former section, the implementation of the memory components for ASPEAR2 had to be taken over from the original synchronous implementation. This, however, requires a special interface which allows to bind the synchronous elements to the asynchronous environment. The corresponding architecture for a simple memory cell is

outlined in Figure 6: Figure 6a thereby presents a macroscopic view with respect to the I/O ports of the component. Beside the dual-rail implementation of the original ports, additional capture done ports for each input as well as an acknowledge port for the output are required.

In Figure 6b, the internal structure of the FSL memory cell is illustrated. Like in every other synchronous circuit, a synchronizer is needed for each asynchronous input (ENABLE, ADDR, DATA\_CDONE). A completion detection function then determines whether the FSL inputs connected to the memory are consistent (wrt. to their phases). In that case the logical input values (To\_Std) as well as the input phase are latched and kept stable until the next consistent input is available. The memory cell’s single rail output is converted into its dual-rail representation with respect to the input phase (To\_FSL), i.e., the FSL output and the corresponding input belong to the same context. As the FSL memory cell behaves like an FSL register, the control unit (CTRL) has the same functionality as described in Section III-D. As a result, only if the *firing rule* holds, the memory output (OUTPUT\_REG) and the capture done signals (CDONE\_REG) are updated, which regulates the data flow according to the FSL specification. Analogous to the output of the synchronous memory cell, which is available after a single clock cycle, the output of the FSL implementation must be associated with the subsequent phase in order to behave the equivalently. The phase inverter (PHASE INV) at the output is therefore essential for the correct semantic behavior of the circuit.

While the concept of the presented interface is a guideline for connecting any synchronous circuit to an FSL design, the actual implementation depends on the particular behavior of the original circuit.

Now the memory components can be combined with the asynchronous processor core, so that the design is ready for place & route. Since our prototyping boards are equipped with Startix II FPGAs, we used *Altera QuartusII* for this final step.

## VI. RESULTS

Table I presents a comparison of the resource utilization of ASPEAR2 and SPEAR2. The numbers are based on the compilation report of *QuartusII*.

Table I: Resource utilization.

	ALUTs	ALUTs (%)	Register	Register (%)
SPEAR2	2179	1.5	576	< 1
ASPEAR2	60341	42	534	< 1

The largely increased number of LUTs can be explained with the implementation overhead of the basic FSL gates: For a 2-input AND gate 6 LUTs are needed. Another reason for the high resource utilization in comparison to the synchronous circuit is the highly inefficient implementation

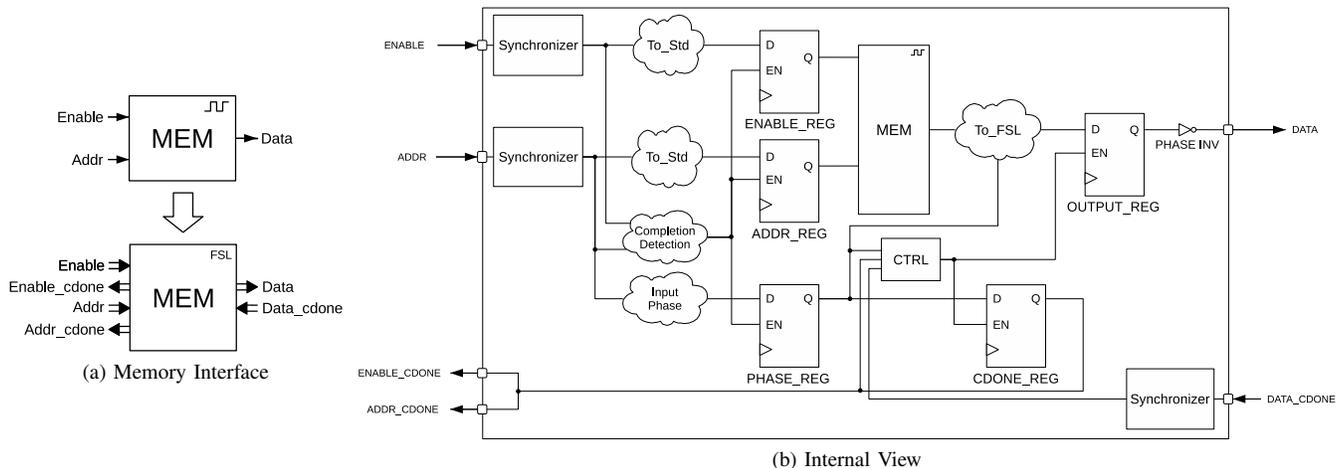


Figure 6: Interfacing FSL with synchronous components.

of arithmetic functions. In case of FSL these functions are built with a high number of distinct combinational gates.

Although our ASPEAR2 implementation consumes a considerable amount of hardware resources, these expenses are tolerable considering the actual goal of our work: A synthesizable asynchronous processor for efficiently conducting fault injection experiments.

While the actual fault injection experiments still need to be done, a fault-free execution of ASPEAR2 (with embedded saboteur units) within our fault injection framework already delivered a significant speed-up. Note, that the configuration of the saboteur units has no significant impact on the overall execution time. The results of our performance measurements are presented in Table II.

Table II: Fault injection performance.

	Instructions	Execution Time	Speed-up
Simulation	127	7103 ms	1
HW emulation	127	13 ms	546

## VII. CONCLUSION

In this paper we have presented a method for automatically transforming a complex synchronous logic design into an asynchronous circuit. The resulting circuit is synthesizable for FPGAs and therefore can be used with the hardware emulation on our fault-injection platform.

Even though the asynchronous implementation of a processor consumes considerable hardware resources of the FPGA, the accomplished emulation support is much faster than any simulation-based approach. This makes our solution well-suited for conducting a large number of fault injection experiments in a reasonable time frame.

## REFERENCES

- [1] A. Benso and P. Prinetto, *Fault Injection Techniques and Tools for Embedded Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.
- [2] M. Jeitler, M. Delvai, and S. Reichor, “FuSE - A Hardware Accelerated HDL Fault Injection Tool,” in *5th Southern Conference on Programmable Logic, 2009. SPL.*, April 2009, pp. 89–94.
- [3] J. Boue, P. Petillon, and Y. Crouzet, “Mefisto-1: a vhdl-based fault injection tool for the experimental assessment of fault tolerance,” *Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, 1998. Digest of Papers.*, pp. 168–173, Jun 1998.
- [4] S.-A. Hwang, J.-H. Hong, and C.-W. Wu, “Sequential circuit fault simulation using logic emulation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 8, pp. 724–736, Aug 1998.
- [5] S. Hauck, “Asynchronous Design Methodologies: An Overview,” *Proceedings of the IEEE*, vol. 83, no. 1, pp. 69–93, Jan. 1995.
- [6] K. Fant and S. Brandt, “NULL Convention Logic,” 1997.
- [7] A. J. McAuley, “Four State Asynchronous Architectures,” *IEEE Transactions on Computers*, vol. 41, no. 2, pp. 129–142, Feb. 1992.
- [8] D. H. Linder and J. C. Harden, “Phased Logic: Supporting the Synchronous Design Paradigm with Delay-Insensitive Circuitry,” *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 1031–1044, Sep. 1996.
- [9] J. Lechner, “Implementation of a Design Tool for Generation of FSL Circuits,” Master’s thesis, Vienna University of Technology, Austria, December 2008.
- [10] M. Fletzer, “SPEAR2 - An Improved Version of SPEAR,” Master’s thesis, Vienna University of Technology, Austria, February 2008.