TU WIEN Informatics

Evaluierung von verschiedenen Tools für Design und Fehlerinjektion von Asynchronen Schaltungen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Martin Schwendinger, BSc
Matrikelnummer 01633080

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Dipl.-Ing. Florian Huemer

Wien, 29. August 2022

Martin Schwendinger | Andreas Steininger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Evaluation of different Tools for Design and Fault-Injection of Asynchronous Circuits

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Martin Schwendinger, BSc
Registration Number 01633080

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Dipl.-Ing. Florian Huemer

Vienna, 29th August, 2022

Martin Schwendinger	Andreas Steininger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Martin Schwendinger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. August 2022

Martin Schwendinger

Acknowledgements

Here I want to express great gratitude to my advisor Professor Steininger for his dedicated support of my thesis. He was constantly reachable for my questions and coordination for further progression.

I also want to thank the other institute members, which were available for considerations a well. Hence, I especially thank Florian Huemer for his guidance related to the Python production rule package (pypr), Robert Najvirt for his effort of refactoring the `autosetup.py` script and Zaheer Tabassam for testing the whole refactored flow and discussing result extraction with me. Then the current flow would be unthinkable without Patrick Behal, a former student, who was still there to answer question to his impressive base work. Therefore I thank him. It was a pleasure to work with you all.

Abstract

Asynchronous circuits (ACs) are currently aced out by synchronous circuits (SCs) in industry purposes. Nevertheless there were prominent innovations utilizing ACs in the field of brain-inspired hardware like SpiNNaker [1], Neurogrid [2] and the TrueNorth [3]. Researchers claim also a theoretical performance advantage of ACs in terms of speed and energy efficiency [4] [5]. However, one handicap when designing ACs is the current lack of tools designated especially to AC design. Often tools originally targeted to SC design are (mis)used. This thesis presents two flows designed especially to AC design. The first one is developed by the Embedded Computing Systems (ECS) group at TU Wien. It is focused on fault-injection experiments at gate level, but provides also Python scripts for high level AC generation. The second one is the Asynchronous Circuit Toolkit (ACT) developed by the asynchronous VLSI and architecture (asyncVLSI) group at Yale University. It aims for a complete coverage of chip design from high level description to fabricable GDSII format. This thesis will extensively present both these flows, and then proceed with their integration into a combined flow. In particular a translation script of the production rule set (PRS) format in its concrete implementation at TU Wien to ACT has been developed. Additionally the fault-injection engine, which is part of the TU Wien flow, has been overhauled to now also support the Prsim simulation software, which is part of the Yale flow. Afterwards, as a proof of concept, for the integrated flow about a million fault-injections have been performed with Modelsim, which was previously without alternative for the TU Wien flow, and Prsim. While one should initially expect the two tools to deliver the same results, mismatches are spotted that could be tracked back to aspects where Modelsim and Prsim operate differently.

Contents

CHAPTER 1

Introduction

Today synchronous circuits (SCs), which use a (global) clock signal to synchronize state changes in memory elements, dominate the field of digital circuit design. Their alternatives, asynchronous circuits (ACs), which rely on handshaking between individual components to negotiate state changes, were around since the mid-1950s. Despite still being aced out by SCs in the industry, recently there were popular innovations in the field of brain-inspired hardware. The Spiking Neural Network architecture (SpiNNaker) [1], the neuromorphic system for simulating large-scale neural models Neurogrid [2] and the neurosynaptic processor TrueNorth [3] are examples. They all use asynchronous design techniques to some extent. The communication between neurons is naturally asynchronous, so with ACs a more realistic brain model can be emulated. Beyond their usage for implementation of brain-inspired hardware, some also claim that ACs can be more performant in terms of speed and energy efficiency than SCs [4] [5].

As e.g. some of the authors of the TrueNorth [3] explicitly state, the design of competitive ACs is handicapped by the lack of established standard tools designated especially to them. For SCs there are plenty of commercial tools. Proprietary tools for Field Programmable Gate Array (FPGA) design are e.g. Intel Quartus Prime and Modelsim or Xilinx Vivado. Prominent open-source tools for mere register-transfer level (RTL) synthesis or simulation are e.g. Yosys or GHDL. For ASIC design companies like Synopsys, Cadence Design Systems, Siemens or Silvaco are tool providers. Currently for the design of ACs often a mix of tools originally intended for SCs design is (mis)used. Tools especially intended for AC design are mostly prototypes usually of University origin. Section 2.2 describes a selection of them.

Two design flows for ACs actually serve as a basis to this thesis. The design flow of the Embedded Computing Systems (ECS) group at TU Wien and the one of the asynchronous VLSI and architecture (asyncVLSI) group at Yale University. They are presented comprehensively in chapter 3 and chapter 4. The design flow of TU Wien can be split into two components, the Python production rule package (pypr) [6], which is

responsible for AC generation from e.g. high level dataflow descriptions, and a fault-injection engine developed alongside [7]. Fault-injection experiments have been the main motivation to develop especially the later part of the flow. In contrast the flow of the Yale University aims for a complete coverage of all design steps needed from conceptual high level description to fabricable GDSII format. Therefore besides multiple design entries for (high level) descriptions of data flows, gates or even transistors, several components are dedicated to physical design. So tools for cell generation, gridded layout, routing etc. are provided as well. The Yale flow is open-source [89], while publication of the TU Wien flow is planned for mid- to long-term.

The centerpiece of this work will present an integration of the two flows. Therefore a translation script from pypr to Yale's Asynchronous Circuit Toolkit (ACT) was written, as well as the fault-injection engine of the TU Wien flow has been complemented to also support simulation of ACT code with the respective simulation software Prsim. The integrated-flow aims mainly for fault-injection, but preserved the physical design capabilities of the Yale flow. Hence, as a demonstration of the new capabilities results from fault-injection experiments will be presented. The target circuits for this experiments will be created by alternating paths through the integrated design flow. Afterwards an extensive comparison and conclusion about the capabilities of the individual flows and their integration follows. Benefits and downsides of each flow and achievements by complementing these two will be discussed in detail.

The structure of the thesis is as follows: In chapter 2 first a brief history of ACs is presented (section 2.1). Prominent state of the art examples mainly from the field of brain-inspired hardware design follow, as well as a selection of tool chains designated to especially AC design (section 2.2). Then section 2.3 provides an introduction to the fundamentals of AC design. In chapter 3 the design flow of TU Wien is discussed in detail. First section 3.1 provides an overview over the design flow and its capabilities. Then section 3.2 traces down the scientific literature about it or about results provided by it. In chapter 4 the design flow of the Yale University is discussed. Section 4.1 provides again an overview over the flow and section 4.2 presents the scientific literature about it. Thereby also complementing the published/documented flow with components described in literature. Chapter 5 presents the integrated flow in section 5.1. Section 5.2 covers the integration of Prsim. Then section 5.3 examines possible parameter spaces for fault-injection experiments and performance issues in. Afterwards section 5.4 presents the results of these experiments with a representative target circuit. Section 5.5 presents an comprehensive comparison of all design flows. Finally chapter 6 concludes about the thesis.

CHAPTER 2

Related Work

This chapter will start with a section about the history of asynchronous circuits (ACs). ACs are around since the mid-1950s and were used quite extensively in the past before they were clearly overshadowed by synchronous design strategies. The first section will contain a selection of historic examples and will try to tell an overall enlightening story for the path of ACs trough history. Afterwards a *State of the Art* section follows, where current outstanding examples of AC developed for research and industry purposes are presented, as well as further sources of more complete lists of asynchronous chips of the past and today. Design flows specifically designated to AC design are also listed there. The last section will cover all fundamentals about ACs needed to follow the thesis further. The difference to synchronous design will be outlined and then most common handshaking protocols, logic and buffer styles for ACs will be presented.

2.1 History of Asynchronous Circuits

The first theoretical mention of AC design was by David E. Muller in 1955 [8]. It already mentioned the intuitive definition of ACs as clockless circuits. Two advantages of ACs were claimed. First that each operation follows directly its preceding one without delay to wait for a clock signal and second that ACs feature sometimes a fault-stop behavior by design. In general the mathematical foundation seems to stem from the University of Illinois and were there further associated with David E. Muller, the scientist also first describing C elements [8], which therefore consequently are referred to as Muller C-elements (MCEs). The ILLIAC I (Illinois Automatic Computer) [90] in 1952 and ILLIAC II [9] in 1962, which are among the oldest processors in history are two applications of AC design. Later the famous book *Switching Theory* in 1965 by Raymond Miller [10] included design techniques for ACs (chapter 9 and 10). It was overall a very comprehensive reference for digital circuit design at that time. It starts with mathematical basics like boolean algebra and other more theoretical considerations before the design of sequential and ACs is discussed. A book containing a brief history of AC design is *Asynchronous Circuit Design* by Chris J. Myers [11] (chapter *Brief History of Asynchronous Circuit Design*). Therein other early mainframe computers implementing ACs like the Atlas (in service from 1962 to 1972) [12] and the MU5 [13] (from 1974 to 1982) designed at the University of Manchester are mentioned. Furthermore it is stated there that the Washington University in St. Luis designed so called asynchronous *macromodules*, which are best described as building blocks like e.g. registers, adders, memories etc. Then AC design was used in the first operational dataflow computer DDM-1 in the 1970s at the University of Utah, as well as in the design of the first commercial graphics system developed at *Evans and Sutherland* (company). For the late 1980s the development of data-driven processors by Matsushita, Sanyo, Sharp and Mitsubishi is mentioned. Then again at the University of Manchester in 1994 the AMULET1 [14][15], which was the first asynchronous processor code-compatible with a synchronous ARM processor, was developed. It was later followed by the AMULET2e [16][17][18] and the AMULET3 [19][20][21]. A very significant theoretical work was done in 1989 by Ivan E. Sutherland describing *Micropipelines* [22]. One of these micropipelines will be discussed in section 2.3. Additional to the already mentioned *Evans and Sutherland* commercial use of research about ACs of the previously mentioned institutes and people was made by e.g. Sun Microsystems or Philips (Research Laboratories) at that time.

Another source of innovations was and is the California Institute of Technology (Caltech). In 1989 the first fully asynchronous microprocessor was designed there according to [11]. It was later followed by the first high-performance asynchronous microprocessor, the MIPS R3000 [23], developed from 1995 to 1998 at Caltech according to the language history [1] of the official documentation page of the Asynchronous Circuit Toolkit (ACT) [91]. The MIPS R3000 is listed there and so the documentation page coincidences with [11], because the predecessor of ACT the Caltech Asynchronous Synthesis Tools (CAST) were used in the development of the MIPS R3000 microprocessor. Further examples,

[1] https://avlsi.csl.yale.edu/act/doku.php?id=history:start

which consequently were developed with predecessors of ACT, are e.g. the first pipelined asynchronous Field Programmable Gate Array (FPGA) architecture *Programmable Asynchronous Pipeline Arrays* [24] at Caltech, the low power microcontroller for sensor networks SNAP developed at the Cornell University [25] and the famous Lutonium [26] processor at Caltech. A good reference for an overview over asynchronous processor generations developed at Caltech is [27]. The previously already mentioned language history of ACT is discussed in much more detail in section 4.2. Commercial use of the innovations stemming (partially) from Caltech especially affiliated with Rajit Manohar, who works currently at the Yale University, was made by e.g. Achronix Semiconductor Corporation or Fulcrum Microsystems [1].

Aside from chip development an important theoretical contribution, the NULL-Convention-Logic (NCL), was delivered by Theseus Research [2]. The book *Logically Determined Design* [28] provides an introduction to NCL and promises it as a new methodology for designing clockless circuit systems. In 1996 the company Theseus Logic was founded to commercialize NCL and it was acquired by Caspian Networks in 2007.

2.2 State of the Art

This section will present prominent and especially more recent examples for AC design than section 2.1. Many of the examples from section 2.1 were quite prominent at their publication time e.g. the MIPS R3000 [23]. Furthermore this section will not include references discussed later in the special reference sections (section 3.2 and section 4.2) of the flow of TU Wien or of the Yale University. These two sections review plenty of related work, which served as a basis for this thesis, even going so far to order them chronologically, relating them to each other and provide an overall story about the research progression at the respective institutes. Also this section focuses on practical examples of chips/circuits developed using asynchronous design principles, which are as already mentioned recent and prominent in some sense.

SpiNNaker (Spiking Neural Network Architecture) is a "massively parallel computer system designed to provide a cost-effective and flexible simulator for neuroscience experiments" [1]. It is able to model a billion neurons, trillion synapses and still matches biological real time, therefore the computational demands to it were challenging. For processing there are two different models suggested, the Izhikevich [29] and the more complex Hodgkin–Huxley [30]. The overall architecture is designed as globally asynchronous locally synchronous system. So locally there are still synchronous ARM968 chips at work, but it utilizes some asynchronous design techniques. In particular it implements Silistix's asynchronous custom protocol. Silistix Limited was a company, which commercialized research results of the Advanced Processor Technologies group headed by Steve Furber at Manchester University [3]. Hence, it is referenced as an innovation in the

[2] http://www.theseusresearch.com/
[3] https://apt.cs.manchester.ac.uk/people/sfurber/

asynchronous domain (e.g. by the TrueNorth paper [3]) and it established inter alia the use of asynchronous design principles for brain-inspired hardware.

Neurogrid [2] is a neuromorphic system that simulates large-scale neural models in real time. A million neurons with billions of synaptic connections are simulated in real time. Specific design choices for this implementation are:

- All neural elements except the soma are emulated.
- Analog circuits are used except for the axonal arbors.
- Neural arrays are interconnected in a tree network.

Asynchronous design principles, which stem from [31] and [32] as the paper explicitly states, are used to model the communication of a neuron to its parent and children neurons. It does reference the SpiNNaker [1] and is referenced by the TrueNorth [3] paper. Overall the paper dives deep into neuronal details and their emulation possibilities in electronic circuits. ACs are only briefly mentioned, but nevertheless the paper contributes to the narrative that ACs have a solid stand, if there is a *natural* asynchrony like the biological behavior of neurons.

The TrueNorth is a "65 mW real-time neurosynaptic processor that implements a non-von Neumann, low-power, highly-parallel, scalable, and defect-tolerant architecture" [3]. Its presentation paper [3] further states that the chip "with 4096 neurosynaptic cores ... contains 1 million digital neurons and 256 million synapses tightly interconnected by an event-driven routing infrastructure" and its "fully digital 5.4 billion transistor implementation leverages existing CMOS scaling trends". Seven principles guided the design of the TrueNorth: *Minimizing Active Power*, *Minimizing Static Power*, *Maximizing Parallelism*, *Real-Time Operation*, *Scalability*, *Defect Tolerance*, *Hardware–Software One-to-One Equivalence*. To meet all expectations custom tools and design approaches were developed, which intersect with or influenced the design flow of the Yale University (chapter 4) significantly. The chip effectively implements a mixed asynchronous-synchronous design approach to meet the expectations put on it. In particular for all communication and control circuits an asynchronous design style, while for computation logic a synchronous one, has been chosen. In the conclusion the authors explicitly state that they are now turning to Computer-aided-design (CAD) research community to contribute to the development of a sophisticated design flow for ACs. Eventually the design flow the Yale University was published on GitHub [89].

The previous examples from the field of brain-inspired architecture design are all prominent and therefore in this section. In contrast now an article [33] is advertised, which is not prominent in a narrow sense, but seems to be an excellent reference for the basic techniques used in AC design, various design tools/languages dedicated to them and it provides an extensive history of asynchronous microprocessors from one of the older microprocessors (the CAM) constructed at Caltech in 1989 to more recent ones like the MSP430 [34] developed at the University of Utah in 2017. The history obviously also includes famous examples, which are in this thesis already mentioned in the last section or this section like the AMULET1/2/3, the MIPS R3000 and the Lutonium. It in general list 37 different

asynchronous microprocessors. So a very good reference, if someone is interested not only in the prominent examples. Hence, [33] alongside with [11] are recommended sources for backtracking the history of ACs, especially microprocessors. However, it is very significant how asynchronous microprocessors, which relate to brain modeling in some sense, seems to stand out from other fields of applications.

Finally at least a paragraph should be dedicated to not the production of asynchronous chips, but the CAD tools out there, which have and hopefully will even more ease their design noticeably. Two design flows for ACs are presented in whole chapters (chapter 3 and chapter 4) in this thesis, because they are a basis to the integrated flow designed alongside this thesis. Especially the reference section of the design flow of the Yale University (see section 4.2) cites several alternative design flows all mentioned in the main article [35] presenting the Yale flow. These flows complemented with those referenced in [33], which is still recommended to consult for a more elaborated listing of flows/tools dedicated to ACs design, are listed here as following:

- **Petrify** [36][92] is a tool for synthesis of Petri nets and asynchronous controllers. It provides many translations options from e.g. final state machines to free/safe/irredundant Petri nets or also from Communicating Sequential Processes (CSP) and more. Additionally to that it also offers the option to generate ACs in combination with a gate library. It was used in the design of the asynchronous TinyRISC TR4101 microprocessor [37].
- **Workcraft** [4] is a framework for interpreted graph models. Inter alia it offers the capability to model speed independent controllers from Signal Transition Graphs or even self timed pipelines from dataflow structures. Papers out there about it are e.g. [38] or [39].
- **TiDE** is a design environment for ACs design with the underlying language Haste, which was former known as Tangram [40][41]. Descriptions in the Haste language are synthesized by syntax-directed translation into a hardware implementation. The TiDE design flow supports the generation of technology mapped netlists of standard cells from a behavioral textual description or a control-data-flow-graph. The flow was utilized in the production of the ARM996HS processor [42].
- **Tiempo** [43] is a design flow for ACs based on the synthesis tool Asynchronous Circuit Compiler (ACC). ACC synthesizes SystemVerilog code to an asynchronous Verilog gate level netlist. This netlist can then be processed further by standard commercial tools for Place & Route operations etc. Correct timing is ensured by the constraints in a Synopsys Design Constraint (SDC) file, which the ACC passes on alongside the actual Verilog netlist. The Tiempo design flow has been used to produce the TAM16 microcontroller [93].
- **Balsa** [44] is a language and framework to synthesize asynchronous hardware via syntax-directed translation. Balsa allows high level description of ACs, which are then translated downward. First the description is synthesized to the intermediate language Breeze, before it is further translated to a gate level netlist and finally

[4] https://workcraft.org/

commercial tools are utilized to generate a layout description. The model for the processor SAMIPS was described in Balsa [45].

- **Proteus** [46] is a flow that provides a syntax-directed translation of high level CSP programs into synthesizable register-transfer level (RTL). From thereon commercial standard synthesis tools are adapted to create a synchronous-image netlist. The in [46] newly presented component ClockFree then translates this image netlist into its asynchronous target netlist. The Proteus flow was used in the design of the uaMIPS processor [47].

The listing is not complete. [33] references even more, but these were not featured especially by it like the others and this thesis simply follows the prioritization of [33] here.

2.3 Fundamentals for Asynchronous Circuit Design

This section gives an introduction to the design of ACs. It starts by an explanation what ACs are in contrast to synchronous circuits (SCs). Basic components and the most common protocols used between components of ACs are presented. The dominantly used ACs in this thesis are quasi-delay-insensitive (QDI) 4-phase dual-rail (DR) circuits, but alternatives are presented as well. The following should be enough to follow the whole thesis, but in general recommended as more comprehensive introductions to ACs are [48] or alternatively [49] and [50] or [51], which also serve as a reference for the following explanations. Also some content may be influenced by similar sections in [7] or [6].

Difference of Synchronous and Asynchronous Circuits
Components of a SC as shown in Figure 2.1 are synchronized by a clock signal (*CLK*). Usually at every rising edge of the clock signal each D-Flip-Flow (i.e. most common type of register for SCs) catches the data on its input and holds it until the next rising edge. Hence, the Flip-Flops in Figure 2.1 form a pipeline, where data at the input affects the output after exactly three clock cycles. If there is a logic cloud between two Flip-Flops, then consequently the time between two rising edges of the clock (i.e. the clock period) needs to be long enough so that the output of the logic cloud is stable before the second Flip-Flop catches it. This constraint mainly influences the maximum clock frequency possible.

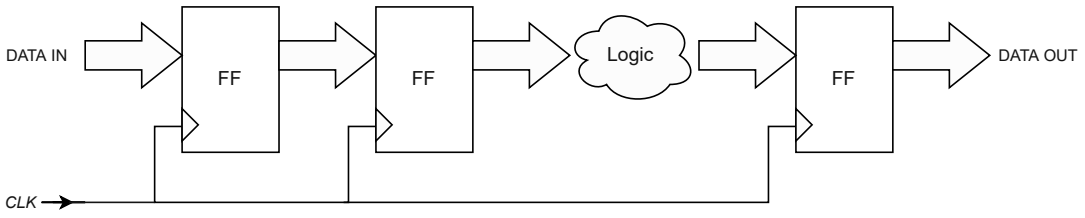


Figure 2.1: Synchronous circuit.

In contrast an AC as shown in Figure 2.2 needs no clock signal for synchronization, but uses a request (req) and acknowledge (ack) signal for handshaking between components. In particular with the request signal the sender indicates that there is new valid data applied and with the acknowledge signal the receiver indicates that the data has been caught. Note that each data passing can be executed individually and as fast as possible and nothing has to wait for a clock edge. Exchange speed between components therefore may vary significantly in an AC. For some types of ACs no request signal is used, but a special data encoding contributes to the handshake operation, therefore the *req* signal in Figure 2.2 is dashed. The *classic* version with request and acknowledge is usually referred to as bundled-data (BD) protocol. The most common encoded AC protocol is referred to as 4-phase DR protocol.

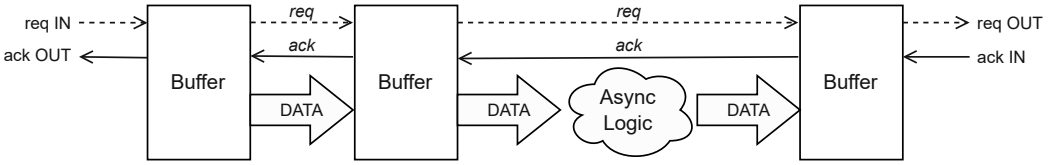


Figure 2.2: Asynchronous circuit.

4-Phase vs. 2-Phase Handshake Protocol

The number of phases corresponds to the number of transitions needed to complete one data transaction. So as shown in Figure 2.3 the 4-phase handshake protocol needs a rising and a falling edge for each, request and acknowledge signal for one transaction. In particular as soon as valid data is applied the sender can raise the request signal. If the request signal is high, the receiver catches the data and raises the acknowledge signal. The high acknowledge indicates to the sender that the request signal can be lowered again and new data can be prepared. After the acknowledge signal follows the request signal to zero, the process can repeat with the next data transaction. The 4-phase protocol always returns to zero between transactions and it can also be interpreted as value indicating. I.e. the high/low values of the signals indicate the current state of the protocol.

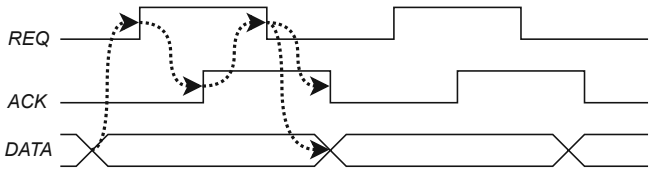


Figure 2.3: 4-phase handshake timing diagram.

In contrast the 2-phase handshake protocol as shown in Figure 2.4 needs only one edge for each request and acknowledge signal for one transaction. Therefore the 2-phase protocol can perform two data transactions per one 4-phase transaction. In particular for the 2-phase protocol the request signal performs an edge so that the receiver knows valid data

is applied. When the receiver caught the data it performs an edge on the acknowledge signal to indicate this. An edge is an edge here, rising or falling does not matter at all. Hence, the 2-phase protocol is transition/edge indicating.

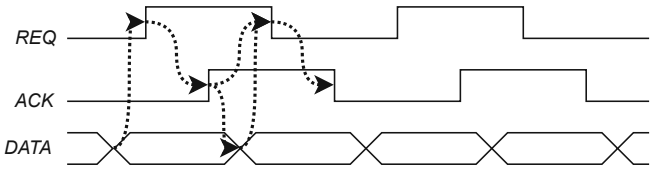
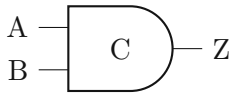


Figure 2.4: 2-phase handshake timing diagram.

Muller C-Element

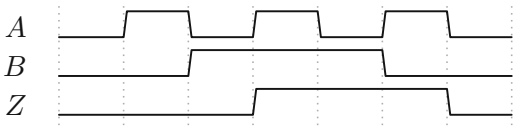
A core component for ACs is the Muller C-element (MCE). It has been already described in the oldest source for AC design by Muller [8]. It is the *only* state holding element in pure ACs. Verbally described it changes the state of its output signal, only if all input signals match, otherwise it holds the previous state. So if all inputs are '1', then the output will rise to '1', but if now only one input becomes '0', the MCE will still hold '1' until bot are '0' again. For a further characterization see Figure 2.5.



(a) Gate schematic of MCE.

A	B	Z
0	0	0
0	1	hold state
1	0	hold state
1	1	1

(b) Truth table of MCE.



(c) Timing diagram of MCE.

Figure 2.5: Characterization of the MCE.

The Muller Pipeline

Figure 2.6 shows the Muller pipeline for 4-phase operation. The figure shows two registers with combinational logic between them. The above part is the control path. The data path below is implemented with D-Latches. Strictly the Muller pipeline is only the control part consisting of MCEs and inverters. Therefore the Muller pipeline itself is adequate for both 4-phase and 2-phase operation. It is only a matter of signal interpretation. While for 4-phase operation the D-Latch is just fine, for 2-phase operation as in Figure 2.7 a Capture-Pass-Latch from the *Micropipelines* paper of Ivan E. Sutherland [22], which originally introduced this concept, is used. Figure 2.8 shows a schematic for the Capture-

Pass-Latch. Note that a delay element (Δ) is needed in the control path to ensure that the second latch does not catch still unstable data from the logic cloud. The signals of the individual control elements of the Muller pipeline correspond to the according timing diagrams Figure 2.3/Figure 2.4 from before.

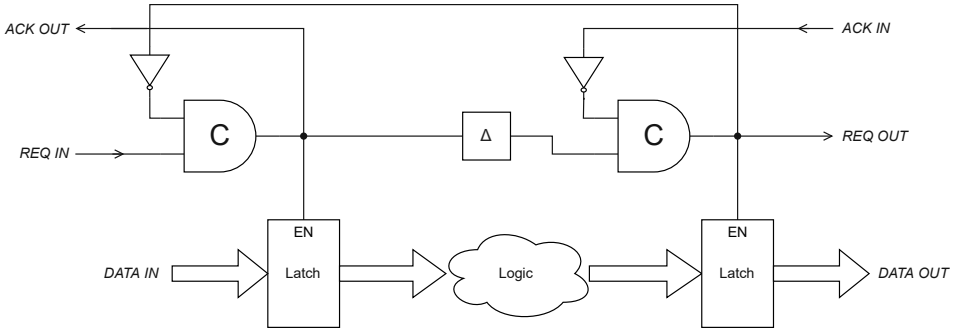


Figure 2.6: 4-phase Muller pipeline.

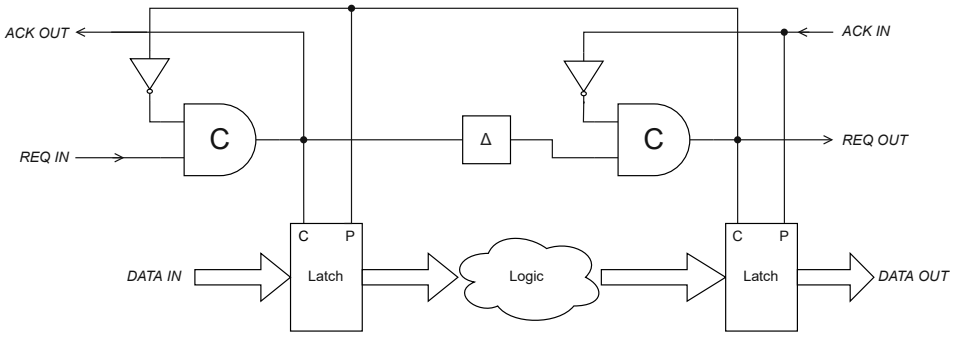


Figure 2.7: 2-phase Muller pipeline.

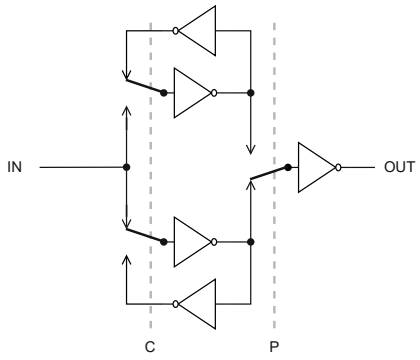


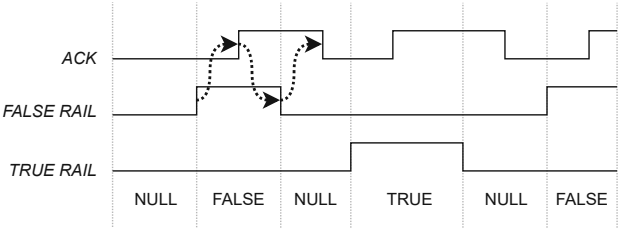
Figure 2.8: Capture-Pass-Latch schematic [22].

4-Phase Dual-Rail Protocol

The 4-phase DR protocol uses a special encoding to make the request signal obsolete. In theory there exists also a 2-phase version, but the 4-phase version is much more common. As the table in Figure 2.9 shows a single bit is represented by two rails. If the true/false-rail is high the bit value is true/false. At no point in time both rails are allowed to be high. As the timing diagram further shows between each data transaction there is a spacer, the NULL-phase. If one of the two rails is high, then the protocol is in the data-phase. In particular, if the sender wants to initiate a transaction it raises one of the rails, so the data content is also already encoded. The receiver recognizes this procession from NULL- to data-phase usually with a completion detector (CD) component and will raise the acknowledge signal as soon as it has caught the data. If the acknowledge signal is high, the sender again enters the NULL-phase by setting both rails to zero. Then after the acknowledge signal falls to zero, the next data transaction can occur. See therefore also the timing diagram Figure 2.9b. The dominant buffer type in this thesis is the weak-conditioned-half-buffer (WCHB) as show in Figure 2.10 already in action with logic between two WCHBs. The preceding WCHB has two bits (a and b) as input/output, so 4 input/output rails in total. The AND gate then reduces this two bits to one, which is then passed to the succeeding one-bit WCHB. The MCEs of the WCHBs provide the buffer functionality. Then there are two CDs in Figure 2.10. Each CD generates the acknowledge signal of its buffer by OR-reduction of both rails (false-rail OR true-rail) of each output bit to effectively one bit, which indicates, whether the current bit is in NULL- or data-phase, and then passes all these bits to a MCE, if the WCHB holds multiple bits. A MCE with three inputs or a tree MCEs would be used for a WCHB holding three bits. A DIMS AND gate is an AND gate, which is synthesized by the delay-insensitive-minterm-synthesis (DIMS) approach. More about DIMS and alternatives to it follows in the next paragraph.

Value	false rail	true rail
False	1	0
True	0	1
NULL	0	0
Illegal	1	1

(a) Encoding.



(b) Timing diagram.

Figure 2.9: 4-phase DR protocol.

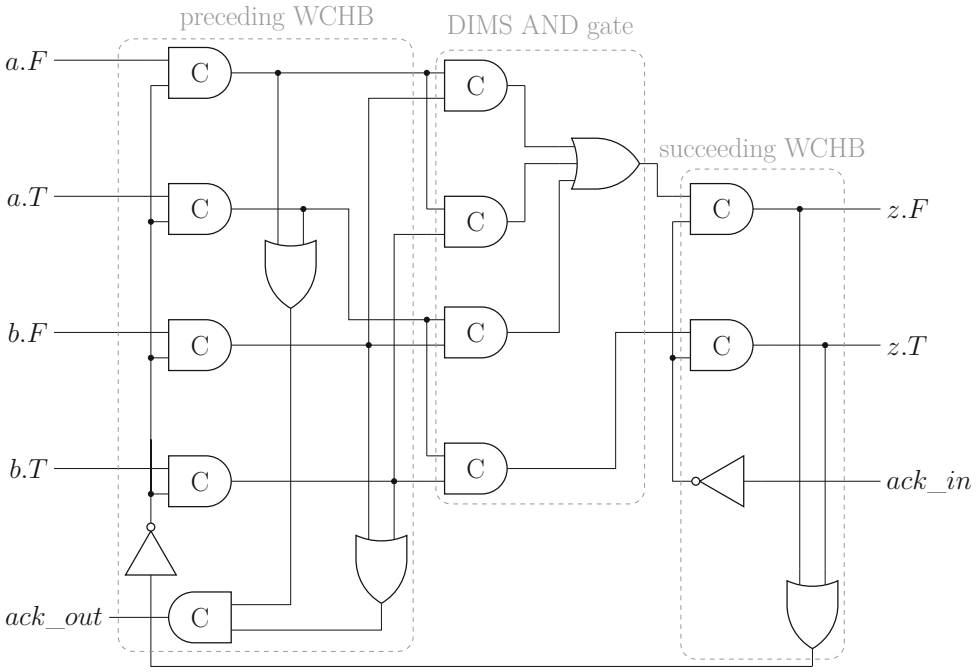


Figure 2.10: WCHB pipeline with DIMS AND gate as combinational logic.

Different Logic Styles for Dual-Rail

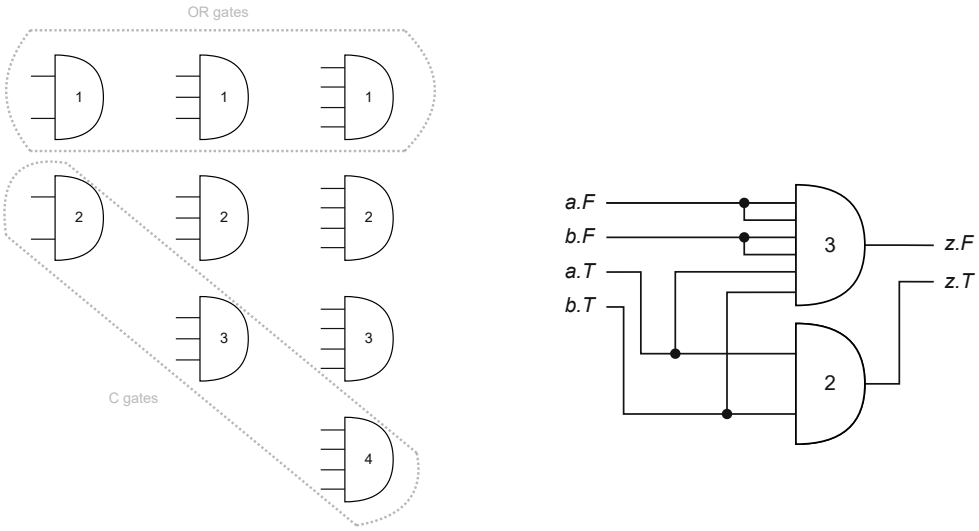
A trivial logic gate for the 4-phase DR protocol is the inverter. Just switching the false-rail with the true-rail is enough. For other logic like AND or OR gates more effort is needed. Hence, there are three common approaches to design arbitrary logic compatible with the 4-phase DR protocol:

- **Delay-Insensitive-Minterm-Synthesis (DIMS)**: As already shown by the exemplary AND gate in Figure 2.10 this approach needs one MCE for each element in the Cartesian product of all input bits, where each bit is the set of its rails. So for our example this can be described by the following formula:

$$\{a.F, a.T\} \times \{b.F, b.T\} = \{(a.F, b.F), (a.F, b.T), (a.T, b.F), (a.T, b.T)\}$$

  MCEs of the AND gate in Figure 2.10 from top to bottom follow the order given by the chosen representation of the resulting set above. Then common logic gates can be used to implement a function that raises the relevant output rail depending on the outputs of the MCEs.

- **Null-Convention-Logic (NCL)**: This approach introduces threshold gates, which can be seen as a generalization of MCEs. The number annotated on each threshold gate in Figure 2.11a denotes the number of inputs that must agree so that the output changes. Threshold gates, for which all inputs must agree, are equivalent to MCEs. Figure 2.11b shows an AND gate implemented with threshold gates. NCL logic is more efficient compared to DIMS, but one downside of it is that it is not trivial to automate the generation of arbitrary logic functions.



(a) Threshold gates for NCL. (b) NCL AND gate.

Figure 2.11: 4-phase DR protocol.

- **NCL with Explicit Completeness (NCLX)**: A first optimization, which aimed to top the previous approaches, was proposed by [52] and then refined by [53], where it is called NCL with Explicit Completeness (NCLX). In theory only one AND gate and one OR gate are needed to generate a DR AND gate. But this yields the risks that the gate already generates an output, even if one bit is still in the NULL-phase. Therefore the acknowledge signal of the succeeding buffer is maybe set too soon and so the late bit is neglected at all. To correct this a CD can be added before and the output can be masked with a MCE. Now usually when using WCHBs the CD at the input is already given by the preceding buffer and the masking by the succeeding buffer. To prevent a possible deadlock of this optimized circuit an additional MCE is added with the output of the CD of the preceding buffer and the acknowledge signal of the succeeding buffer as inputs. Figure 2.12 shows this, please note that the CD shown corresponds to that of the preceding buffer and may compare with Figure 2.10

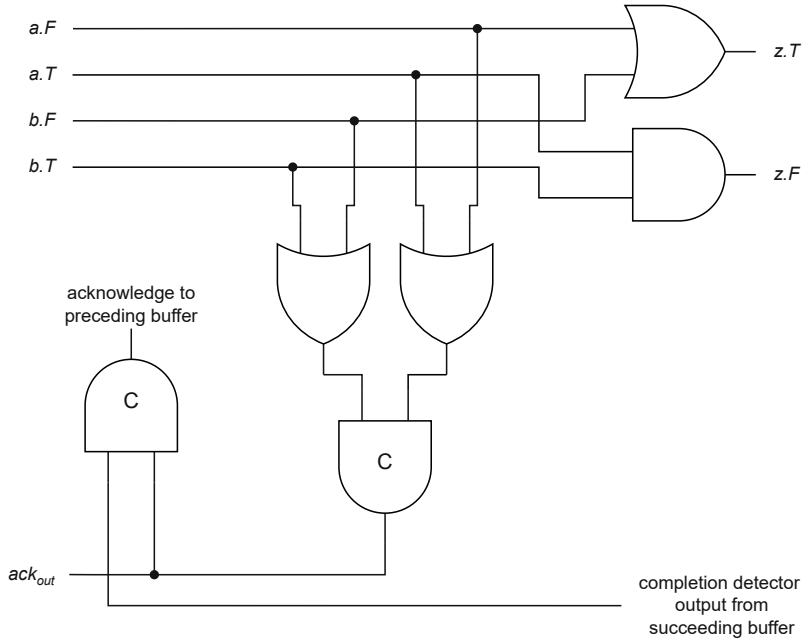


Figure 2.12: NCLX AND gate.

Delay Insensitivity of Asynchronous Circuits

A circuit is (truly) delay-insensitive (DI), if for correct operation the only timing restriction is that each gate and wire delay is positive and finite. Unfortunately the class of such circuits is limited to those constructed only with inverters and MCEs as shown by [54]. Hence, the Muller pipeline is delay insensitive. A less strict classification is speed independence, i.e. all wire delays are assumed to be zero, but the gate delays can be arbitrary. This is basically a less realistic phrasing, because interconnect delays are quite significant in modern circuits, for specification of QDI circuits. For QDI it is assumed that gates have arbitrary delays, but for all wires an *isochronic fork* [55] condition applies. This condition says that the delay of each wire after a fork must be equal. Hence, a QDI circuit can be viewed as a speed independent one, if all wire delays are just moved (i.e. added) to the gate delay of their source. QDI circuits are dominantly used in this thesis.

CHAPTER 3

Design Flow by TU Wien

The Embedded Computing Systems (ECS) group, which is a part of the Institute of Computer Engineering at TU Wien and whose research ranges by self definition from "dependable and power-efficient digital circuits to future generation computer architectures to networked embedded systems and fault-tolerant distributed systems" [94], recently developed a design flow starting with the *high level* design of asynchronous circuits (ACs) and going down to gate level simulation with fault-injection. The design of the ACs is intended to be modular and highly configurable, so for example a buffer type like the weak-conditioned-half-buffer (WCHB) can easily be exchanged with another possibly enhanced version. Therefore, the fault-resilience of many circuit variations can be analyzed by exposing them to fault-injections. The design flow consists mainly of Python scripts, which delegate some work to external tools like Modelsim for simulation or communicate to a Structured Query Language (SQL) database to store results and query next tasks. Overall the design flow is quite focused on fault-injection, however development is still in progress and so with the progression of research interests of the ECS group group it will evolve. Incremental publishing of parts as open-source code is planned for mid- to long-term. The following sections will provide an overview over the design flow, and then examine all scientific reference work related to the flow itself, its developments process and the future plans for it.

3.1 Overview

The design flow can be considered as a collection of Python scripts utilizing some additional tools. Especially for AC generation the Python production rule package (pypr) backs the front-end scripts shown in Table 3.1, which further describes them briefly, as well as Table 3.2 does for preexistent external tools.

Python script	Description
`simple_qdi_synth.py`	Converts provided combinatorial single-rail Verilog designs into a quasi-delay-insensitive (QDI) dual-rail (DR) production rule set (PRS).
`bmc.py`	Bounded-model-checker, which checks a circuit described in PRS format for specific properties.
`prscom.py`	Handles the conversion of PRS format to a hardware description language (HDL) like Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog.
`tbgen.py`	Generates VHDL testbenches for ACs featuring fault injections according to a YAML file configuration.
`dbplotter.py`	A tool that plots results of simulation jobs executed by `dbworker.py`.
`autosetup.py`	Wraps around the `dbworker.py` to calibrate some settings for fault-injection experiments.
`dbworker.py`	Communicates with the SQL database and according to configuration adds open tasks to the database or executes one task from it and then stores the results in the database. Uses multithreading.

Table 3.1: Front-end Python scripts of the design flow of TU Wien.

External tool	Description
Modelsim	Is used for gate level simulation of VHDL code and applying the fault injections using the force command.
Yosys	Provides a framework for register-transfer level (RTL) synthesis. For now mainly used by `simple_qdi_synth.py`.
Z3	A theorem prover developed by Microsoft Research [95]. Is mainly used by `bmc.py`.
GHDL	An analyzer, compiler, simulator and synthesizer for VHDL. GHDL is mainly planned as simulator alternative to Modelsim.
OSVVM	A VHDL library used for random number generation in the testbenches generated by `tbgen.py`.

Table 3.2: External tools used by the design flow of TU Wien.

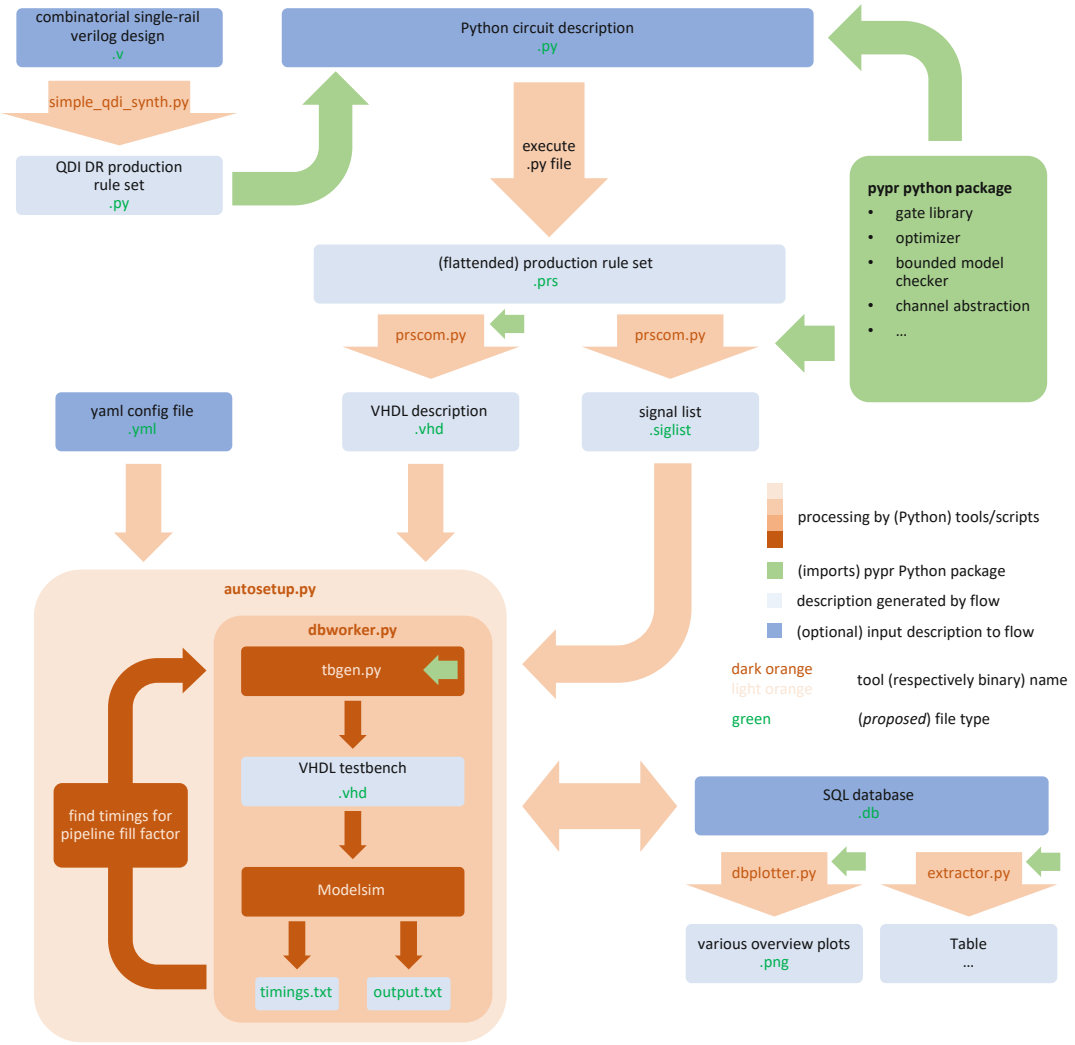


Figure 3.1: Design flow for gate level fault-injection analysis.

Figure 3.1 visualizes the design flow. The darker blueish boxes represent descriptions or configurations that should be present as an input to the flow. The lighter represent intermediate descriptions or outputs generated by the flow. Additionally each box is labeled (in black) with what it represents. *Proposed* file types are labeled green. Production rule set (PRS) is a special circuit description format originally established at California Institute of Technology (Caltech), but here the acronym usually refers to a concrete implementation of such a PRS description developed at TU Wien (see [6]). The application of a (Python) tool/script to progress from one circuit or result representation to another is usually shown by orange arrows or in case of more comprehensive processing they lead to orange boxes. Script/tool names are written in dark or light orange. Darkening of orange boxes or arrows is only for visual contrast, as well as lighter and

darker orange lettering. Most functionality for describing and further processing circuits with Python scripts is provided by the Python package pypr. Therefore this package is included in practically every other Python script. This is visualized by the green arrows.

The flow starts by describing an AC with a Python script. The pypr package will provide appropriate classes and functions for this description. Combinational logic described by Verilog code can be translated (by `simple_qdi_synth.py`) and then be loaded into the main Python circuit description. By execution the Python script conventionally stores its circuit as a (flattened) PRS representation. From there the representation can be further translated to e.g. VHDL, or some properties like a list of signals can be extracted using `prscom.py`. The section of Figure 3.1 consisting of `autosetup.py` and `dbworker.py` basically visualizes how the former executes the latter multiple times to first calibrate various settings (most importantly a pipeline-fill-factor) for the following fault-injection experiments (see [7]). Setup instructions of an experiment are stored to the SQL database by the first execution of `autosetup.py`/`dbworker.py` and then queried by `dbworker.py` later to actually perform all the simulations. Results will be stored back to the database. The `tbgen.py` script creates the VHDL testbench according to the calibration and settings in the YAML config file. `dbplotter.py` and `extractor.py` are just examples for further Python tools, which perform SQL queries on the database to analyze results. However, the flow is still work in progress, so not every (proposed) component is presented in Figure 3.1. For example the bounded-model-checker (`bmc.py`), which at the moment just checks for deadlock, orphans or some set assertions, is missing.

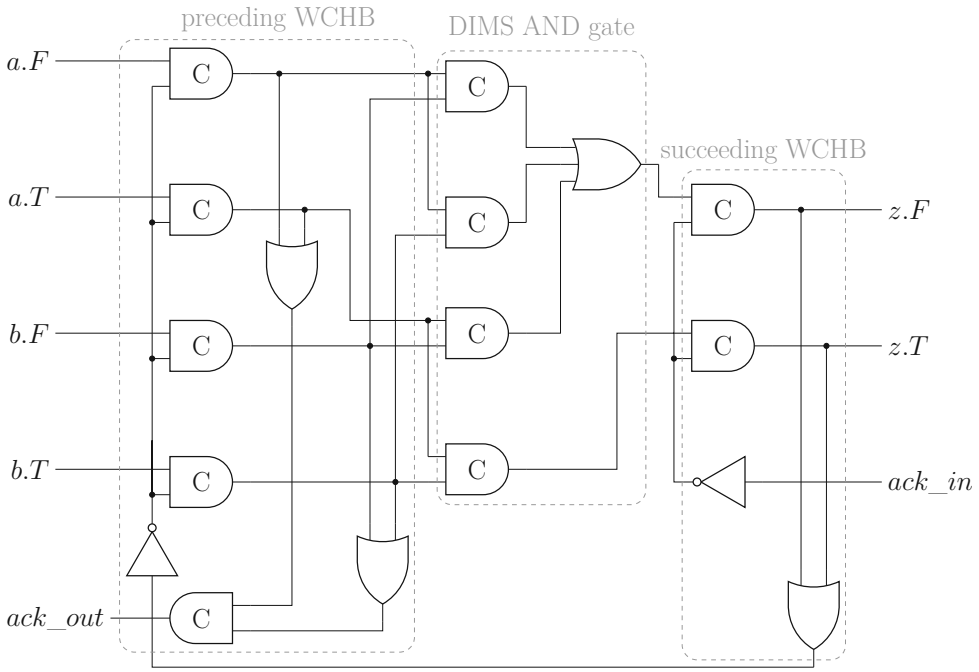


Figure 3.2: QDI 4-phase DR circuit featuring WCHBs and a DR AND gate.

To further demonstrate how a usual path through the design flow looks like, a comprehensive path example follows. Listing 3.1 shows Python code describing a QDI 4-phase DR circuit featuring WCHBs and a DR AND gate. The circuit is visualized in Figure 3.2. The circuit consists of a delay-insensitive-minterm-synthesis (DIMS) AND gate preceded and succeeded by WCHBs. The Python code in Listing 3.1 starts with declaring a production rule set library (`PRSLib`) in Line 2, a packing of all input/output signals to an input/output channel (`chin`/`chout`, Line 4/5) and declaring of two intermediate channels for the connection from the preceding WCHB to the DIMS AND gate and from there to the succeeding WCHB. Afterwards the individual components of the circuit itself are added to the library (line 12 onward). The two buffers are elements of the built-in library and the DIMS AND gate is imported as PRS description from the `bitwise_and.prs` file. The file has been generated by the Python script `simple_qdi_synth.py`, which translated the Verilog description shown in Listing 3.2 to the corresponding production rule set shown in Listing 3.3. In Line 13 of Listing 3.1 the final circuit object is declared as data-flow-graph. The following lines add the previously declared channels, buffers and the DIMS AND gate to the circuit. From Line 21 to 24 the `Connect` and `Pipeline` functions are used to connect the different blocks.

```
1  ...
2  lib = PRSLib() # initialize production rule set library
3  # declare channels
4  chin = DIDRChannel(name="chin", direction=ChannelDirection.Input, ack="ack_out"
       , data=["a", "b"])
5  chout = DIDRChannel(name="chout", direction=ChannelDirection.Output, ack="
       ack_in", data=["z"])
6  b1Out = DIDRChannel(name="b1Out", data=["a", "b"])
7  b2In = DIDRChannel(name="b2In", data=["z"])
8  # add needed components to library
9  lib.AddPRS(ParsePRSFile("bitwise_and.prs"))
10 lib.AddPRS(DRBuffer(name="buf1", input_channel=chin, output_channel=b1Out))
11 lib.AddPRS(DRBuffer(name="buf2", input_channel=b2In, output_channel=chout))
12 # declare and add components to circuit bufferedAnd
13 bufferedAnd = DFG()
14 bufferedAnd.AddIOChannel([chin, chout])
15 bufferedAnd.AddLogicBlock("op_and", lib["bitwise_and"])
16 bufferedAnd.AddHandshakingBlock("b1", lib["buf1"])
17 bufferedAnd.AddHandshakingBlock("b2", lib["buf2"])
18 # connect components of circuit
19 bufferedAnd.Pipeline("chin", "b1")
20 bufferedAnd.Connect("b1:b1Out", "op_and", {"a_out":"a", "b_out":"b"})
21 bufferedAnd.Connect("op_and", "b2:b2In")
22 bufferedAnd.Pipeline("b2", "chout")
23 # generate production rule set, flatten and optimize it
24 lib.AddPRS(bufferedAnd.CreatePRS("bufferedAnd"))
25 flat_prs = Flatten("bufferedAnd", lib)
26 Optimize(flat_prs)
27 # save it to file
28 with open("bufferedAnd_flat.prs", "w") as file:
29     file.write(flat_prs.ToCode())
30 ...
```

Listing 3.1: Python description of a QDI 4-phase DR circuit featuring WCHBs and a DR AND gate.

```
1  module bitwise_and #(parameter
       DATA_WIDTH = 4) (a, b, z
       );
2      input [DATA_WIDTH-1:0] a;
3      input [DATA_WIDTH-1:0] b;
4      output [DATA_WIDTH-1:0] z;
5      assign z = a & b;
6  endmodule
```

Listing 3.2: Combinational single-rail Verilog design.

```
1  prs bitwise_and is
2  inputs
3      a : DRBit attributes(needs_cd :=
           false);
4      b : DRBit attributes(needs_cd :=
           false);
5  outputs
6      z : DRBit;
7  begin
8      cell_0__onehot00 := cgate(b.F, a.F);
9      cell_0__onehot01 := cgate(b.F, a.T);
10     cell_0__onehot10 := cgate(b.T, a.F);
11     z.T := cgate(b.T, a.T);
12     z.F := or_gate(cell_0__onehot00,
           cell_0__onehot01,
           cell_0__onehot10);
13 end prs;
```

Listing 3.3: QDI production rule set description.

Afterwards the circuit is flattened, optimized and finally stored in PRS format to the file `bufferedAND_flat.prs`. The content of this file is shown in Listing 3.4. The PRS format represents a circuit similar to the module representation in Verilog. The circuit is named (`prs <name> is ...`) and then declarations of inputs and outputs follow before local variables can be defined. Finally logic gates with their connections are listed. Concerning the description of an inputs, outputs or locals consider the declaration of the input signal `a` on Line 3 in Listing 3.4:

```
a :  DRBit attributes (channel := chin , role := data );
```

- `<signal_name> :  <signal_type> [attributes(...)];` is how a signal declaration line looks in general.
- `DRBit` is the type of a DR bit used for QDI circuits.
- With the optional `attributes` keyword an input/output can be affiliated to a communication channel and its role therein can be specified. Obviously this matches the channel declarations of the Python description.
- Various other attributes, which e.g. mark the need of a completion detection for local variables, are available as well.

Nevertheless, attributes are optional as they are only preserved as comments when translated to a HDL. From Line 14 onward logic gates are described. Note that single bit boolean local variables (like e.g. `op_and__cell_0__onehot00`) do not need to be declared. This is considered a feature. Logic gates can be described as software-like functions taking input signals as parameters and return the output signal. A line like `op_and__cell_0__onehot00 := cgate (op_and__b.F, op_and__a.F);` should be self-explanatory. `.F` and `.T` refer to the two rails of a DR bit. Then rules can also have attributes like the following:

- `init`: specifies the reset signal and value for a state holding gate.
- `delay`: specifies a transport delay for gates (not used in Listing 3.4).

```
1  prs bufferedAnd is
2  inputs
3      a : DRBit attributes(channel := chin, role := data);
4      b : DRBit attributes(channel := chin, role := data);
5      ack_in : Bit attributes(channel := chout, role := ack, channel_type := DIDR
           );
6      reset : Bit attributes(role := reset);
7  outputs
8      ack_out : Bit attributes(channel := chin, role := ack, channel_type := DIDR
           );
9      z : DRBit attributes(channel := chout, role := data);
10 locals
11     op_and__a : DRBit attributes(needs_cd := false);
12     op_and__b : DRBit attributes(needs_cd := false);
13     b2__z_in : DRBit attributes(channel := "b2->b2In", role := data);
14 begin
15     op_and__cell_0__onehot00 := cgate(op_and__b.F, op_and__a.F);
16     op_and__cell_0__onehot01 := cgate(op_and__b.F, op_and__a.T);
17     op_and__cell_0__onehot10 := cgate(op_and__b.T, op_and__a.F);
18     b2__z_in.T := cgate(op_and__b.T, op_and__a.T);
19     b2__z_in.F := or_gate(op_and__cell_0__onehot00, op_and__cell_0__onehot01,
           op_and__cell_0__onehot10);
20     b1__c_a_out := or_gate(op_and__a.F, op_and__a.T);
21     op_and__a.F := cgate(a.F, b1__en) init(0, reset);
22     op_and__a.T := cgate(a.T, b1__en) init(0, reset);
23     b1__c_b_out := or_gate(op_and__b.F, op_and__b.T);
24     op_and__b.F := cgate(b.F, b1__en) init(0, reset);
25     op_and__b.T := cgate(b.T, b1__en) init(0, reset);
26     ack_out := cgate(b1__c_a_out, b1__c_b_out);
27     b1__en := nor_gate(z.F, z.T);
28     z.F := cgate(b2__z_in.F, b2__en) init(0, reset);
29     z.T := cgate(b2__z_in.T, b2__en) init(0, reset);
30     b2__en := inv(ack_in);
31 end prs;
```

Listing 3.4: Production rule set description of a QDI 4-phase DR circuit featuring WCHBs and a DR AND gate.

For a more detailed examination of all the features and tools provided by pypr for describing ACs including the custom PRS format of TU Wien consider the Ph.D. thesis of Florian Huemer [6]. In general the thesis elaborates extensively about QDI circuits.

Now using `prscom.py` this PRS description can be further translated to VHDL in order to perform fault-injection experiments using Modelsim. Listing 3.5 shows an exemplary translation of one rule in the PRS to VHDL. Due to its scope neither the whole VHDL representation of the circuit, nor any further explanation of the translation process is considered appropriate for discussion here. So a very simple way to proceed would be feeding the `dbwoker.py` script directly with a YAML configuration file and the corresponding VDHL description of a circuit. Listing 3.6 shows the clipped content of such an exemplary YAML file. Currently the YAML file defines properties for gate level simulation as following:

- Properties like circuit type, pipeline style, logic type, data width etc. mainly to categorize simulation results in the SQL database.
- Input channel timings, i.e. timing of data-phase and NULL-phase alternation. Usually randomized for each single bit signal within a set range.
- Output channel timing for the acknowledge signal of the sink (provided by testbench).
- Perhaps a check function, which the testbench utilizes to check correctness of logic operations and log violation.

- The number of tokens (i.e. data and NULL-phase alternations), which should be passed through the circuit until simulation is done.
- Fault-injection configurations like when to inject for what duration and how many iterations (i.e. repetitions of simulation with altered fault-injection details).

Additionally to the properties above the YAML file contains for now build instructions as shell commands, because the build process (i.e. advancing a Python file description to a ready-to-simulate VHDL circuit) is not enough standardized yet, so needed flexibility is given by the option to include arbitrary shell commands to the build process. Alongside that there are also various file naming related configurations considered not worth explaining explicitly.

```
1  --op_and___a.F := cgate(a.F, b1___en)
       init(0, reset);
2  \\RULE:op_and___a.F\\ : process(all)
3  begin
4    \\op_and___a\\.F <= \\op_and___a\\.
       F;
5    if ( a.F and \\b1___en\\ ) then
6      \\op_and___a.F <= '1';
7    end if;
8    if ( not (a.F or \\b1___en\\) )
       then
9      \\op_and___a\\.F <= '0';
10   end if;
11   if ( reset ) then
12     \\op_and___a\\.F <= '0';
13   end if;
14 end process;
```

Listing 3.5: Exemplary translation of one rule in the PRS to VHDL code.

```
1  ...
2  build:
3    tb:
4      channels:
5        chout:
6          log_tokens: true
7          ack_delay:
8            up:
9              type: random
10             min: 10 ns
11             max: 20 ns
12             create_generic: true
13 ...
14 # The fault injection parameters
15 fault_injection:
16   victim:
17     mode: random
18     file: alu.siglist
19   injection_value:
20     mode: random
21   injection_time:
22     mode: random
23     range: [100000, 2000000]
24   injection_duration:
25     mode: fixed
26     value: 1000
27   iterations: 100000
28 ...
```

Listing 3.6: Clipped YAML configuration file.

For some of these properties, especially the timing of data- and NULL-phase alternation and acknowledge edges, one might want to generate them in an automatic way to get e.g. a desired fill level of the pipeline, so that the fault-injection experiments will be performed on an appropriately utilized pipeline. Hence, the `autosetup.py` script is there to perform this task. The script takes basic configurations via the YAML file and of course a VHDL circuit as input and then sets everything up for meaningful fault-injection experiments. In detail `autosetup.py` simulates the circuit multiple times. Each time the timing of the testbench is a bit altered to approximate desired characteristics, which usually have been empirically determined for a certain circuit type so that the final fault-injection experiments yield significant results. For more details on that consider the master's thesis of Patrick Behal [7]. The thesis presents an analysis of transient faults of

QDI circuits with different pipeline styles. Results of more than 1 billion fault-injections, for which the design flow of TU Wien was used, have been evaluated.

Finally the interaction of `autosetup.py/dbworker.py` with the SQL database should be elaborated here. First note that `autosetup.py` usually only executes `dbworker.py`, which then executes `tbgen.py`, whose scope is also beyond appropriate for being discussed in detail here, and ultimately Modelsim to perform a simulation. Hence, `dbworker.py` actually communicates with the SQL database or more specifically it inserts scheduled tasks, queries them later to execute them and eventually inserts the results back to the database. The following enumeration elucidates the process further:

1. `autosetup.py/dbworker.py` is executed with a configuration YAML file, which also specifies (how to build) the VHDL entity to simulate.
2. (`dbworker.py` is executed multiple times by `autosetup.py` to perform simulations for calibration of the pipeline fill factor. Temporary VHDL testbenches for this step are created by `tbgen.py`.)
3. Appropriate simulation configurations are inserted as an open task to the SQL database.
4. When `dbworker.py` is executed in run mode, it will query open tasks from the SQL database and execute the simulations. In case a simulation is done, its results are stored back to the database.
5. The database results can be queried by various means. Two example tools are the `dbplotter.py` and `extractor.py` script.

It should be emphasized that after step 3 another or even multiple other workstations can proceed with step 4 and query the (partial) task from an online SQL database. Hence, in combination with the multiprocessing capability, which the `dbworker.py` script already provides, a larger fault-injection experiment for research can be distributed over multiple CPU cores on multiple workstations. This is intentional design from the start to aim for a comprehensive fault-injection framework suitable for very large scale experiments. The storage strategy for simulation tasks and their results in the SQL database is also designed to occupy minimal disk space.

Table 3.3 shows exemplary (clipped) content of significant tables of the SQL database. In particular the subtables present the following:

- Table 3.3a shows two independent fault-injection experiments (denoted as simulations). One performed with a multiplier circuit and the other one with an ALU circuit. Each one is described by its corresponding YAML file, which describes also the build process to obtain the circuit in VHDL alongside with properties for testbench generation.
- Table 3.3b shows the splitting of the two simulations into 4 tasks (2 each). Thereby actual Modelsim usage can be distributed better to different workstations.
- Table 3.3c shows the simulation results. So each time when a fault-injection resulted in one tracked error type (here timing, coding, value or glitch), it is logged as one entry in the results table.

- Table 3.3d stores the relation of a signal ID to the actual signal name in VHDL circuit description.

id	githash	settings	settingspath	...
1	b18f...	<content of YAML file>	settings_mul.yml	...
2	b18f...	<content of YAML file>	settings_alu.yml	...

(a) Table of simulations.

id	simulation_id	status	seed	iterations	...
1	1	DONE	718...	250	...
2	1	DONE	638...	250	...
3	2	DONE	610...	250	...
4	2	DONE	167...	250	...

(b) Table of tasks.

id	simulation_id	task_id	injected_signal_id	inject_time	error_type	...
1	1	1	44	132062	timing	...
2	1	1	287	171471	coding	...
3	1	2	59	191181	timing	...
			...			
7	2	3	593	205707	coding	...
8	2	3	550	467798	value	...
			...			
14	2	4	600	476654	coding	...
			...			

(c) Table of results.

id	simulation_id	name
1	1	umul4x4_tb/uut/ack_out_lvl0_0
		...
44	1	umul4x4_tb/uut/b1___a_in(3).F
287	1	umul4x4_tb/uut/logic_ls___cell_7___a.T
59	1	umul4x4_tb/uut/b1___c_s_out_3
		...
593	2	alu_tb/uut/op_addsub___cell_10___onehot11
550	2	alu_tb/uut/op_addsub___a(2).T
600	2	alu_tb/uut/op_addsub___cell_12___b.T

(d) Table of signals.

Table 3.3: Exemplary content of significant tables of the SQL database.

The two scripts `dbplotter.py`, which plots a basic overview of applied fault injections and resulting deviation from the golden run, and `extractor.py`, which converts the results stored in the SQL database to a more processable format, are really just exemplary and with given specification of the SQL database the results may be specifically extracted according to actual desired usage.

3.2 Related Literature

This section aims to provide an overview over *scientific* literature, which relates to the design flow of TU Wien or specifically to pypr or `dbworker.py`. The ECS group has been interested in ACs and fault-injection experiments for years now. A few of the first publications are [56], [57] and [58], which elaborate about delay-insensitive (DI) 4-phase data transmitting, benefits and downsides of their various coding schemes and present a novel one, respectively. Considerations about timing robustness and fault-sensitivity follow ([59], [60]). Eventually the design flow around pypr and `dbworker.py` evolved with the master thesis of Patrick Behal [7], picking up the pipeline-load-factor (PLF) already described in [60] as pipeline-fill-level and creating the automatic fault-injection engine with workload distribution referred to as `dbworker.py`. pypr is presently best examined by the Ph.D. thesis of Florian Huemer [6]. However, prior versions of it have existed since 2018 and so ACs designed by pypr (or results of experiments with them) were already published since then.

Table 3.4 provides a backwards chronologically ordered and commented list of the significant publications related to the designs flow of TU Wien. The table starts at the bottom with [60], because this paper established basics like the pipeline-fill-level (then further examined as PLF in [7]) or the inter- and deadlocking variant of the WCHB (visualized in Figure 3.7). For [60] still an one-time-setup was used for fault-injection and retrieving results. [7] lifted it to the status presented in section 3.1 (visualized by Figure 3.1). The papers listed above [7] in the table then rely on the automatic fault-injection engine developed alongside [7] and usually exploit its capabilities to run large-scale experiments to advance knowledge about QDI circuits. The backwards listing ends with [6] as the most recent work comprehensively covering QDI circuits from theory to practice, presenting also various fault-injection experiments to characterize certain QDI circuit variations better and in general is the reference for pypr and the PRS format.

Now the publications of Table 3.4 should be discussed a bit more. Hence, a short paragraph dedicated to each follows in bottom to top order.

Identification and Confinement of Fault Sensitivity Windows in QDI Logic [60]
The paper studies the natural resilience of different QDI circuits. Therefore with extensive fault-injection experiments the sensitivity windows are visualized by a novel approach to retrieve detailed information about the sensitivity of individual signals and its dependence on the pipeline-fill-level, implementation details, path delays etc. For the fault-injection experiments the same fault categorization scheme

as later supported by the `dbworker.py` was used. Further the paper compares different methods from literature that claim to enhance the resilience of WCHBs. Finally after identifying the main vulnerabilities of conventional WCHBs the paper proposes two enhanced WCHBs, the inter- and deadlocking variant better illustrated in Figure 3.7. So synoptically the paper already introduced key elements of the parameterization strategy (pipeline-fill-level, variations of WCHB, input/output/path delays, etc.) later used for billions of fault-injection experiments performed with the `dbworker.py` for other publications.

Title	Description	Authors	Publication date and reference
Contributions to Efficiency and Robustness of Quasi Delay-Insensitive Circuits	Ph.D. thesis of Floarian Huemer. It discusses theoretic foundation of DI communication, sync to async domain crossing, various (efficient) QDI logic and pipeline styles and presents results of fault-injection experiments to elaborate on fault-tolerance. Main reference for pypr, PRS and synthesis to it from Verilog.	F. Huemer	May 2022 [6]
On SAT-Based Model Checking of Speed-Independent Circuits	Presents how SAT-based bounded-model-checking can be used to prove function correctness of ACs. Paper to the `bmc.py` tool. Utilizes the open-source Z3 theorem prover [95].	F. Huemer, R. Najvirt, A. Steininger	April 2022 [61]
Towards Explaining the Fault Sensitivity of Different QDI Pipeline Styles	Analyzes fault sensitivity of QDI circuits subjected to single-event-upsets (SEUs). Experiments with different multiplier implementations and buffer styles. Explicitly mentions the design flow tools of TU Wien as setup.	P. Behal, F. Huemer, R. Najvirt, A. Steininger, Z. Tabassam	September 2021 [62]
An Automated Setup for Large-Scale Simulation-Based Fault-Injection Experiments on Asynchronous Digital Circuits	Presents also the design flow of the TU Wien as toolset (visualized in Figure 3.5) for fault-injection experiments. Elaborates on the huge parameter space (visualized in Figure 3.6) to be considered to obtain representative results. In general focused on the toolset itself, its automation concerning parameter selection and simulation run time speedup by workload distribution.	P. Behal, F. Huemer, R. Najvirt, A. Steininger	September 2021 [63]
Quantitative Comparison of the Sensitivity of Delay-Insensitive Design Templates to Transient Faults	Master thesis of Patrick Behal. Tools for automated parameter generation for fault-injection experiments and the distribution of workload were developed alongside this thesis. The PLF was refined by it. Over one billion fault-injection experiments on QDI circuits with alternating pipeline styles have been performed for it.	P. Behal	May 2021 [7]
Identification and Confinement of Fault Sensitivity Windows in QDI Logic	Aims to analyze the natural resilience of different QDI circuit types and then determines the window of vulnerability using fault-injection experiments. It provides a visualization of sensitivity windows related to the pipeline-fill-level and eventually presents the inter- and deadlocking variant of the WCHB, as in Figure 3.7.	F. Huemer, R. Najvirt, A. Steininger	October 2020 [60]

Table 3.4: Selection of more significant literature related to the design flow of TU Wien.

Quantitative Comparison of the Sensitivity of Delay-Insensitive Design Templates to Transient Faults [7]

Alongside the master thesis of Patrick Behal all core tools (`autosetup.py`, `dbworker.py`) for automated parameter generation for fault-injection experiments

and the workload distribution of the needed simulations to multiple cores on multiple workstations have been developed. As motivation the thesis states that until then to the authors awareness a satisfying, comprehensive and quantitative comparison of the robustness of different design styles for ACs to transient faults was not published. So to change this the respective tools were developed to perform over one billion simulations with injections of transient faults on QDI 4-phase DR circuits with varying pipeline styles. The thesis deepens the theoretical foundation for simulation parameters like the PLF or the fault categorization and the developed tools consequently establish them after their just experimental use in [60]. After an extensive statistical evaluation of the simulation results further plans to expand the research to other types of ACs, which are indeed realized by following publications, are already stated as an outlook.

An Automated Setup for Large-Scale Simulation-Based Fault-Injection Experiments on Asynchronous Digital Circuit [63]

This paper followed thesis [7] to presents the tool chain for highly automated fault-injections into ACs in a condensed format. It again highlights the capabilities of the tool chain to perform billions of meaningful fault-injection experiments. It mentions the features of `autosetup.py` to generate parameters (like e.g. the PLF) automatically for simulations. Then the extensive parameter span needed to achieve representative results is quite well illustrated. See also Figure 3.6. The paper categorizes the individual parts of the flow as following:

- *Task generation*: A given or partially auto-generated parameter set is added to the SQL database alongside the AC to simulate.
- *Simulation execution*: Open tasks are executed with workload distribution over multiple workstations and CPU cores. Results are stored back to the database.
- *Data extraction*: Various routines to extract and process results in the database were developed and are utilized accordingly to favor sophisticated conclusions.

So basically the presented tool flow examined by this paper is already very close to the whole design flow described in the last section of this thesis. For circuit generation also the pypr for describing an AC is mentioned.

Towards Explaining the Fault Sensitivity of Different QDI Pipeline Styles [62]

Here another paper focusing on fault sensitivity of QDI ACs to single-event-transients. Experiments were done on ACs with varying multiplier logic and buffer styles. The paper shares the great vision of performing large-scale fault-injection experiments in order to narrow down vulnerabilities accurately to finally develop enhancements to buffer or logic style from the gained insight. The paper again elaborates about the parameter space and provides a minimal visualization of the design flow. See therefore Figure 3.6 and Figure 3.5. Of course the existing tools from pypr for circuit generation to the known automatic fault-injection engine (`autosetup.py/dbworker.py`) are advertised as well. As an outlook, after better insight into fault-sensitivity relations of QDI circuits is achieved and therefore parameter space can be reduced

to focus on only parameter choices that are then deemed significant, a progression to more complex target circuits is announced.

On SAT-Based Model Checking of Speed-Independent Circuits [61]
This is the paper to the tool `bmc.py`. The tool implements a bounded-model-checker, so it converts a circuit into a SAT problem to model its state progression for a certain (bounded) time. The Z3 theorem solver from Microsoft Research [95] is used for the implementation. `bmc.py` checks a PRS description for various custom properties stated as static assertions in the circuit description, if the circuit deadlocks or for gate orphans (a gate whose output may changes to some input pattern, but this is not reflected by the observable output of the circuit). DI 4-phase DR ACs are used as target circuits for the experiments.

Contributions to Efficiency and Robustness of Quasi Delay-Insensitive Circuits [6]
The Ph.D. thesis of Florian Huemer covers crossings from synchronous to asynchronous domains and vice versa, elaborates about efficiency of different DI communication schemes, AC description methods (so pypr, PRS and more) and of course the fault-tolerance of QDI circuits. The thesis is the reference for the PRS format and other pypr related content including the synthesis of Verilog code to PRS and the conversion of PRS to VHDL. As expected for a Ph.D. thesis it covers each topic comprehensible, from theoretical considerations (like how to transmit information in a in a DI way in general) to presenting the results of practical case studies with varying QDI logic and buffer designs. It also relates itself to various other papers from Florian Huemer et al. not listed ([64], [57], [56], [58], [59]) and listed ([61], [60], [62]) in Table 3.4. General work leading up to this thesis has started long before the tools for automation of the fault-injection procedure (`autosetup.py`/`dbworker.py`) were developed, nevertheless (predecessors of) pypr were developed much earlier and so also used for mere AC generation.

Hopefully the above commentary to related work has deepened the understanding of the reasons for the creation of the design flow of TU Wien and showed the knowledge gain already achieved by the referenced work, as well as the amount of reasonable investigations that come within reach by the flow's capabilities. While the overview of the last section presented the design flow with original content, what follows now is a recap of a historic representation of the design flow alongside with consideration about the parameter space investigated in past research.

The visualization of the design flow of TU Wien in Figure 3.1 in the last section is original to this thesis. A more minimalist version here shown in Figure 3.5 was already presented in [7], [63], [62] and [6]. It hides all concrete tools and just shows a conceptional view, which is explained and related to the actual tools in the following:

- It starts with PARAMETER SETS (including the circuit description). Concretely parameter sets are given in YAML config files or may be generated by `autosetup.py`.

- The SIMULATION TASK GENERATOR then synthesizes the target circuit and performs a golden run (from which deviations are considered as errors). The synthesis is implemented by the pypr package, while performing simulations is already a task for the `dbworker.py`.
- Planned simulations with all circuit details and fault-injection specifications are then inserted to a SQL database, from which later the results are extractable, as tasks.
- SIMULATION WORKER refers to the `dbworker.py` in run-mode, so when it queries tasks from the database, performs the needed simulations and inserts the results back afterwards.

An excellent illustration of the parameter span of interest for (future) research and also supported by current design flow is shown in Figure 3.6. The overall parameter span is *huge* due to being the Cartesian product of seven sets of possibly infinitely different options. For some of the options like LEVEL OF PIPELINING, BIT WIDTH, DELAY PARAM. or LOAD SCENARIO an inductive approach can be considered, i.e. experiment with a few different options and discover a pattern which the results follow when increasing this numeric option. LOGIC FUNCTION here refers to the functionality the actual AC is expected to implement. Obviously results differ for different circuits. LOGIC STYLE refers to currently three different approaches how to synthesize custom logic for 4-phase QDI DR ACs. There are two original different PIPELINE STYLEs (i.e. buffer styles) WCHB and MOUSETRAP [65]. The dead- and interlocking variant are both developed by the ECS group based on gained insight from their research to increase fault-resilience. See therefore Figure 3.7. The deadlocking variant just hinders the Muller C-element (MCE) to change its state after a coding violation (so when both rails indicate '1') occurred, hence stopping the pipeline operation until reset. The interlocking variant just prevents both output rails from indicating '1' by favoring the first arriving '1' on either rail and basically neglecting the later '1' on the other rail. For further explanation see [60]. So synoptically recent research has explored only the tip of a huge parameter set. Future work will continue to dig deeper. The comprehensive circuit generation and automated fault-injection framework provided by pypr and `autosetup.py`/`dbworker.py` is needed to deal with the enormous parameter space.

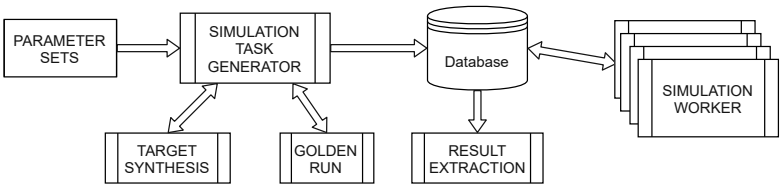


Figure 3.5: Design flow as presented in original publications ([7], [63], [62]).

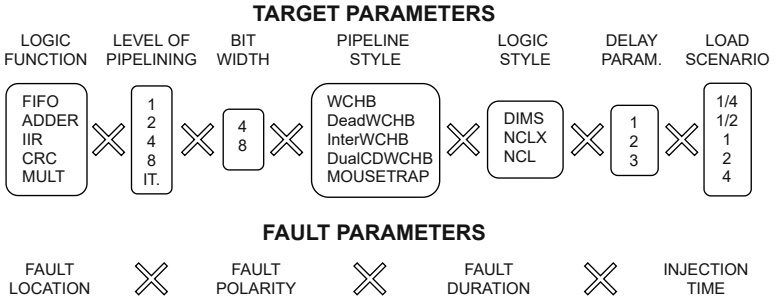


Figure 3.6: Illustrated parameter space as presented in [63].

(a) Deadlocking WCHB.

(b) Interlocking WCHB.

Figure 3.7: Modified WCHBs as in [60].

CHAPTER 4

Design Flow by Yale University

Inter alia motivated by lack of design tools for asynchronous circuits (ACs) needed for the design of the TrueNorth chip [3] (see also section 2.2), the asynchronous VLSI and architecture (asyncVLSI) group at Yale University, assembled the Asynchronous Circuit Toolkit (ACT). For ACT various tools have been specially developed, as well as preexistent open-source tools have been integrated to yield a comprehensive design flow for ACs. It provides high level design entries like dataflow descriptions and Communicating Hardware Processes (CHP), as well as notions for very low level descriptions like size and leakage of individual transistors. Combined with the open-source very large-scale integration (VLSI) layout tool MAGIC the design flow aims to output a circuit layout description in GDSII/CIF format. ACT is still work in progress, nevertheless some parts have already been published. On GitHub under asyncvlsi[1] all currently published components, as well as other open-source repositories related to ACT or other projects by the asyncVLSI group are listed. The following sections will first provide an overview over the different parts of ACT and their common operation as one design flow. Then an extensive analysis of scientific work related to ACT follows.

[1] https://github.com/asyncvlsi/

4.1 Overview

The design flow can be considered as a collection of tools or binaries. Table 4.1 lists all tools affiliated to ACT. For convenience also links to the corresponding source code and documentation are attached. Each preexistent tool (i.e. not specially developed for ACT) is denoted as *external*.

Binary	Description	External	Code Source / Documentation
`aflat`	Translates (low level) ACT code to a flat production rule representation.		https://github.com/asyncvlsi/act https://avlsi.csl.yale.edu/act/doku.php
`prs2sim`	Translates production rules to sim format (used by IRSIM).		
`prs2net`	Translates production rules to a SPICE netlist.		
`ext2sp`	Converts MAGIC extract files into a hierarchical SPICE file.		
`Prsim`	Gate level simulation tool specially for ACT.		
	Standard library for ACT code (used with binaries `aflat` and `chp2prs`.)		https://github.com/asyncvlsi/stdlib
`chp2prs`	Translates CHP code to (low level) ACT code.		https://github.com/asyncvlsi/chp2prs
`prs2fpga`	Translates production rules to synthesizable Verilog code, see also [66].		https://github.com/asyncvlsi/prs2fpga
IRSIM	Switch level simulator.	×	https://github.com/RTimothyEdwards/irsim/ http://opencircuitdesign.com/irsim/
MAGIC	VLSI layout tool.	×	https://github.com/RTimothyEdwards/magic http://opencircuitdesign.com/magic/
Xyce	Analog circuit simulator (alternative to e.g. LTspice).	×	https://github.com/Xyce/Xyce https://xyce.sandia.gov/

Table 4.1: Individual tools affiliated to ACT.

Figure 4.1 visualizes the design flow. The bluish boxes represent different description options for an AC. The shade of blue describes the design level. Simulation outputs are also represented as bluish boxes, because they also *describe* (the behavior of) a circuit. Additionally the circuit description type of each box is labeled on each box (black) along with a *proposed* file type. Furthermore the various tools of ACT, converting more high level descriptions to lower levels possibly adding some low level specifications automatically, are represented with arrows. Tool names are written in orange. Orange arrow filling indicates that the tool was specifically developed for ACT, yellow arrows that it is an external (preexistent) open-source tool. One oddity, which should be clarified here, is that various tools converting from low level ACT are usually named `prs2...` (e.g. `prs2sim`), despite the file type for this description type is `.act`. This is correct and seemingly intended by ACT developers. The `.prs` file ending is an acronym for production rule set. This is a flattened version of low level ACT, i.e. ACT only using the `prs {...}` body. The flow in Figure 4.1 is not complete, additional tools (like e.g. `prs2fpga`) are omitted to keep it simple, so the (automated) progression from high to low level description is well illustrated.

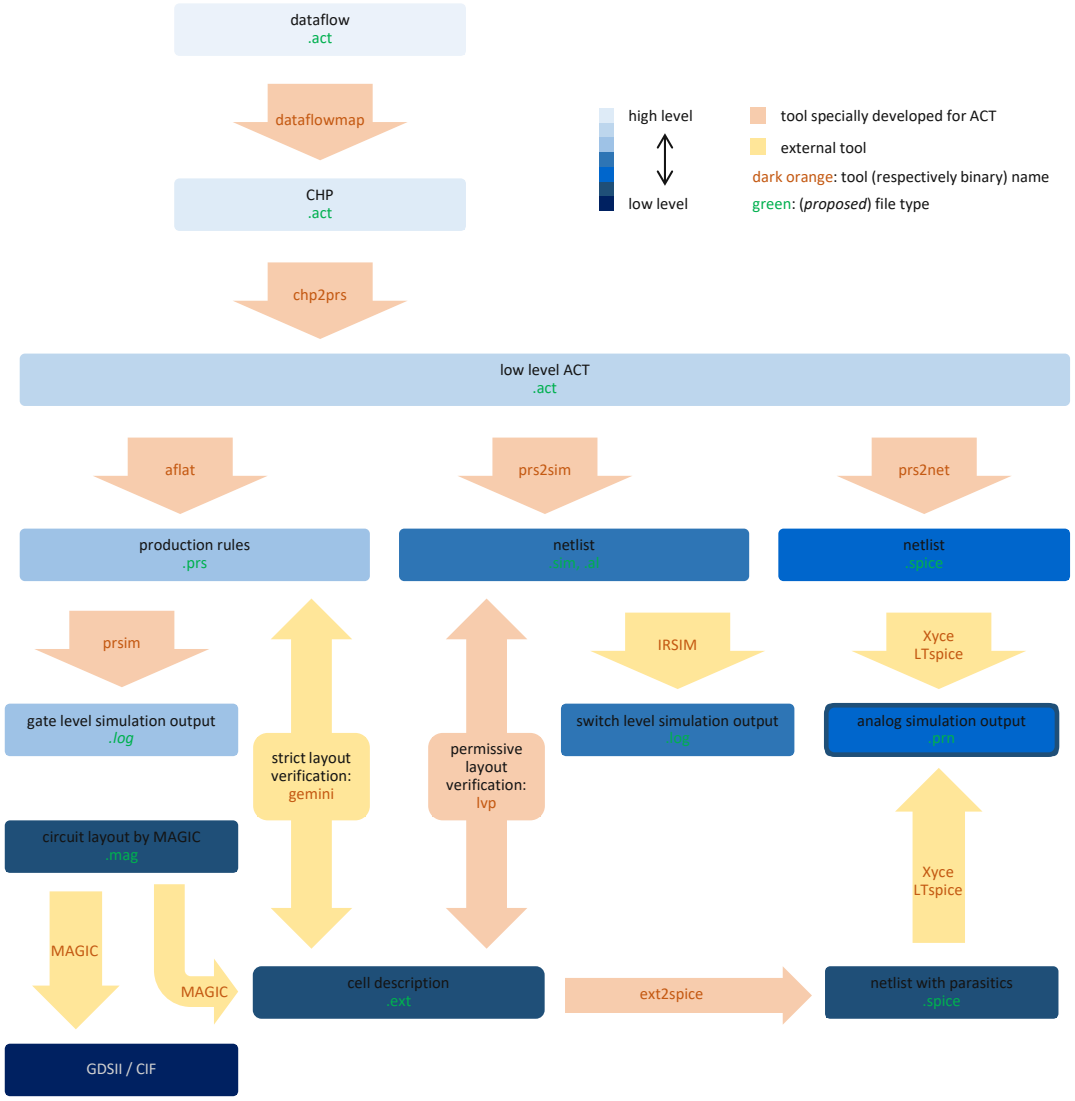


Figure 4.1: Design flow utilizing ACT.

The following examples traverse all tools of the flow in Figure 4.1 except MAGIC, `ext2sp`, `gemini` and `lvp`, so basically no layout is done, because here only a basic understanding of the different design entry points and simulation levels should be conveyed. Hence, circuit description types are shown as (clipped) code to show how descriptions are really coded and what information is complemented by the tools.

The highest level design entry available is to design a circuit by specifying a dataflow. Listing 4.1 shows how a simple adder with synchronization at input and output can be implemented. `chan?(int<32>)` denotes an input channel (`?`) transferring a 32-bit integer with 4-phase dual-rail (DR) synchronization included. `chan!(int<32>)` then denotes the corresponding output channel (`!`). When using the keyword `chan`, per default DR logic will be used to implement the abstract channel notion, but bundled-data (BD) is an option too, when tools are configured accordingly. The command `$ dflowmap dfBufferedAdder.act` will translate the dataflow description to a CHP description. The output code is very mechanical, therefore the following CHP example will continue with manually created code.

Listing 4.2 shows the same buffered adder as in the dataflow example before, but coded in CHP, i.e. the next lower design entry. The syntax of `*[I1?x, I2?y; z:=x+y; O!z]` can be disassembled as following:

- `*[...]`: repeat `...` infinitely.
- `I1?x`: perform synchronization operation for receiving data on channel `I1` and store it to `x`.
- `O!z`: perform synchronization operation for transmitting data stored in `z` on channel `O`.
- `P,Q` execute `P` and `Q` simultaneously.
- `P;Q` execute `P` and `Q` sequentially.

Further syntax is considered trivial. Hence, it can be seen here nicely that CHP requires the synchronization operations coded explicitly, while the dataflow description assumes them implicitly. The CHP description of Listing 4.2 can be translated to low level ACT using the command `$ chp2prs chpBufferedAdder.act bufferedAdder lowLevelACTbufferedAdder.act`. The code witten to `lowLevelACTbufferedAdder.act` is again very mechanical, so it is not shown here.

Listing 4.3 shows how a low level ACT description can be coded manually. The circuit is much simpler, because it is already close to transistor level, so a full DR adder would be cumbersome. Each line inside `prs {...}` is a production rule mapping, i.e. a boolean condition mapped to either a pull down or a pull up. More specifically on the left of `'->'` a logic expression evaluating to true or false is required. On the right a pull down (`'-'`) or pull up (`'+'`) rule is required. Listing 4.3 implements a CMOS-compatible Muller C-element (MCE). For CMOS-compatibility the condition on the left side is only allowed to consist of non inverted variables, if it is a pull down rule, and only inverted variables for a pull up rule. Due to that limitation a non inverted MCE has to be implemented by first implementing production rules for an inverted one, where the output is mapped to a local variable (here `b`), and then an inverter has to be implemented with additional production rules. There are also alternatives to `'->'` to reduce code a bit.

```
1  // file: dfBufferedAdder.act
2  defproc bufferedAdder (chan?(int<32>) I1, I2; chan!(int
       <32>) O)
3  {
4      dataflow {
5          I1 + I2 -> O
6      }
7  }
```

Listing 4.1: Dataflow description of a buffered adder.

```
1  // file: chpBufferedAdder.act
2  defproc bufferedAdder (chan?(int<32>) I1, I2; chan!(int
       <32>) O)
3  {
4      int<32> x, y, z; // z = x + y
5      chp {
6          *[I1?x, I2?y; z:=x+y; O!z]
7      }
8  }
```

Listing 4.2: CHP description of a buffered adder.

```
1  // file: mce.act
2  defproc mce (bool? i1, i2; bool! o)
3  {
4      bool b;
5      prs {
6          i1 & i2 -> b-
7          ~i1 & ~i2 -> b+
8          b -> o-
9          ~b -> o+
10         // simpler way to specify MCE beh:
11         // i1 & i2 #> b-
12         // ~b -> o+ will be automatically assumed:
13         // b => o-
14     }
15 }
16 // create instance of mce
17 mce mce_inst;
```

Listing 4.3: Low level ACT description of MCE.

The low level ACT description of Listing 4.3 can further be flattened to the pure production rule description of Listing 4.4 by using the command `$ aflat mce.act`. Of course the mere process definition of `mce` must also be instantiated (e.g. as instance `mce_inst`), because when flattening uninstantiated process definitions are ignored. The flattened description basically only consists of the plain production rules without any declarations of input, output, internal etc..

The production rule description of Listing 4.4 can then be simulated with the gate level simulation tool Prsim. Listing 4.5 shows the simulation using the interactive shell of Prsim executed via `$ prsim mce.prs`. The `cycle` command of Prsim maybe requires a bit explanation. It simulates until the circuit has reached a stable state, hence possibly forever. One can write `cycle <node>` to simulate only until the value of the specified node changed.

Alternatively to a gate level simulation also a switch level simulation can be performed using IRSIM. Therefore the low level ACT description must be converted to a netlist first using the command `$ prs2sim mce.act mce`. The netlist is shown in Listing 4.6. E.g. `p mce_inst/i1 Vdd mce_inst/n#6 2 5` can be disassembled as follows:

- `p` denotes a PMOS transistor.
- `mce_inst/i1` is connected to the gate.
- `Vdd` is connected to the source.
- `mce_inst/n#6` is connected to the drain. `n#6` denotes here just a new temporary wire.
- 2 and 5 denote the width and length of the transistor.

See also Figure 4.4 for a visualization of the netlist. Listing 4.7 is a list of aliases for various node names.

Using the command `$ irsim scmos30 mce.sim mce.al -irsim.inp` the netlist of Listing 4.6 can be simulated with the instructions of Listing 4.8. A short syntax explanation of Listing 4.8:

- `h`: set node to high.
- `l`: set node to low.
- `ana`: add to analyzer window.
- `w`: add to watchlist.
- `s`: perform simulation step.

IRSIM features a graphical analyzer window, hence the visual simulation output is shown in Figure 4.3.

```
1  "mce_inst.i1"&"mce_inst.i2"->"mce_inst.b"-
2  ~"mce_inst.i1"&~"mce_inst.i2"->"mce_inst.b"+
3  "mce_inst.b"->"mce_inst.o"-
4  ~"mce_inst.b"->"mce_inst.o"+
```

Listing 4.4: Production rule set (.prs) of MCE.

```
1  (Prsim) initialize
2  (Prsim) set mce_inst.i1 0
3  (Prsim) set mce_inst.i2 0
4  (Prsim) watchall
5  (Prsim) cycle
6                    0 mce_inst.i1 : 0
7                    0 mce_inst.i2 : 0
8                   10 mce_inst.b : 1  [by mce_inst.i2:=0]
9                   20 mce_inst.o : 0  [by mce_inst.b:=1]
10 (Prsim) set mce_inst.i2 1
11 (Prsim) cycle
12                  20 mce_inst.i2 : 1
13 (Prsim) set mce_inst.i1 1
14 (Prsim) cycle
15                  20 mce_inst.i1 : 1
16                  30 mce_inst.b : 0  [by mce_inst.i1:=1]
17                  40 mce_inst.o : 1  [by mce_inst.b:=0]
18 (Prsim) exit
```

Listing 4.5: Interactive shell of Prsim.

```
1  | units: 30 tech: scmos format: MIT
2  p mce_inst/i1 Vdd mce_inst/n#6 2 5
3  p mce_inst/b Vdd mce_inst/o 2 5
4  p GND Vdd mce_inst/n#8 18 3
5  n mce_inst/i1 GND mce_inst/n#3 2 3
6  n mce_inst/b GND mce_inst/o 2 3
7  n Vdd GND mce_inst/n#9 46 3
8  n mce_inst/i2 mce_inst/n#3 mce_inst
       /b 2 3
9  p mce_inst/i2 mce_inst/n#6 mce_inst
       /b 2 5
10 p mce_inst/o mce_inst/n#8 mce_inst/
       b 2 3
11 n mce_inst/o mce_inst/n#9 mce_inst/
       b 2 3
```

Listing 4.6: Netlist of MCE for switch level simulation (.sim).

```
1  = Vdd Vdd!
2  = GND GND!
```

Listing 4.7: Alias file (.al) for IRSIM.

```
1  #setup everything
2  logfile irsim.log
3  h Vdd!
4  l GND!
5  # add to watchlist and gui
6  ana mce_inst/i1 mce_inst/i2
       mce_inst/o
7  w mce_inst/i1 mce_inst/i2 mce_inst/
       o
8  # run sim
9  l mce_inst/i1
10 l mce_inst/i2
11 s
12 h mce_inst/i2
13 s
14 h mce_inst/i1
15 s
```

Listing 4.8: Simulation instructions for IRSIM.

Figure 4.3: IRSIM analyzer window showing switch level simulation of MCE.

Finally also an asynchronous SPICE simulation is possible. Therefore the low level ACT description must be translated to a SPICE netlist using the command `$ prs2net mce.act -p "mce<>"`. Listing 4.9 shows the resulting SPICE file. The netlist is basically the same as in Listing 4.6. However, to break down one line consider line `M0_ Vdd i1 #6 Vdd p W=1.5U L=0.6U`:

- `M0_`: MOSFET transistor.
- `Vdd`: source.
- `i1`: gate.
- `#6`: drain (`#6` new temporary wire).
- `Vdd`: bulk.
- `p`: model of MOSFET (specified by `model.sp` (see Line 14 in Listing 4.10)).
- `W` and `L`: width and length of MOSFET.

The netlist description of Listing 4.9 is then included into the test harness shown in Listing 4.10, as well as a concrete model for the MOSFET transistors. Test parameters (like e.g. the input pattern) are specified in the test harness. Every proper SPICE simulator can be used from hereon. The asyncVLSI group recommends Xyce on their tutorial page. Hence, using `$ Xyce mce_test_harness.sp -o results.prn` and afterwards `$ gnuplot -p -e "set style data lines; set multiplot layout 3,1; plot 'results.prn' using (
$2*1e9):3 title 'voltage in1'; plot 'results.prn' using (
$2*1e9):4 title 'voltage in2'; plot 'results.prn' using (
$2*1e9):5 title 'voltage out';"` yields Figure 4.6.

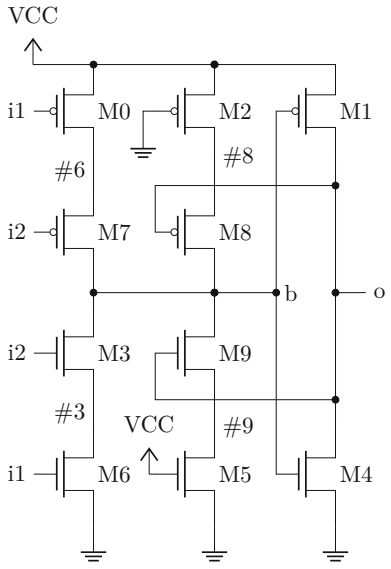


Figure 4.4: Transistor level representation of CMOS MCE.

```
1  *
2  *------ act defproc: mce<> ------
3  * raw ports:  i1 i2 o
4  *
5  .subckt mce i1 i2 o
6  *.PININFO i1:I i2:I o:O
7  *.POWER VDD Vdd
8  *.POWER GND GND
9  *.POWER NSUB GND
10 *.POWER PSUB Vdd
11 *
12 * ---- node flags ----
13 *
14 * b (state-holding): pup_reff=0.8;
       pdn_reff=1.33333
15 * o (combinational)
16 *
17 * ---- end node flags ----
18 *
19 M0_ Vdd i1 #6 Vdd p W=1.5U L=0.6U
20 M1_ Vdd b o Vdd p W=1.5U L=0.6U
21 M2_keeper Vdd GND #8 Vdd p W=0.9U L
       =5.4U
22 M3_ GND i1 #3 GND n W=0.9U L=0.6U
23 M4_ GND b o GND n W=0.9U L=0.6U
24 M5_keeper GND Vdd #9 GND n W=0.9U L
       =13.8U
25 M6_ #3 i2 b GND n W=0.9U L=0.6U
26 M7_ #6 i2 b Vdd p W=1.5U L=0.6U
27 M8_keeper #8 o b Vdd p W=0.9U L=0.6
       U
28 M9_keeper #9 o b GND n W=0.9U L=0.6
       U
29 .ends
30 *------ end of process: mce<> ------
```

Listing 4.9: Netlist of MCE for SPICE simulation.

```
1  **********************************
2  *   Xyce test harness
3  **********************************
4
5  *** Default supply nodes  ***
6  .global vdd
7  .global gnd
8
9  *** Set Vdd to 5V, GND to 0V ***
10 vd vdd 0 dc 5.0v
11 vg gnd 0 dc 0.0v
12
13 ** include nfet and pfet models ***
14 .inc model.sp
15
16 *** include circuit model ***
17 .inc mce.sp
18
19 *** set the voltage of "in" ***
20 vp1 i1 0 PAT(5 0 0n 0.1n 0.1n 4n
       b00111010)
21 vp2 i2 0 PAT(5 0 0n 0.1n 0.1n 4n
       b01100001)
22
23 *** Instance of the inverter
       subcircuit ***
24 X1 i1 i2 out mce
25
26 *** Specify simulation and options
       ***
27 .print tran format=gnuplot v(i1) v(
       i2) v(out)
28
29 *** Run a transient simulation for
       32 ns with 1ps timestep ***
30 .tran 1p 32n
31
32 .end
```

Listing 4.10: Xyce test harness for SPICE simulation.

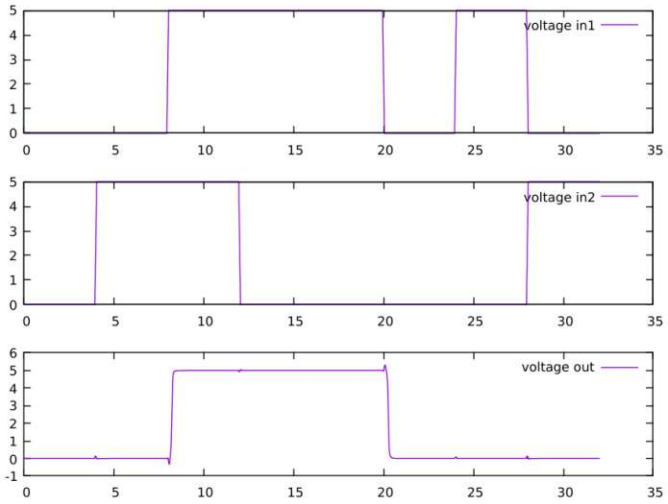


Figure 4.6: Plotted Xyce simulation output of MCE.

4.2 Related Literature

This section aims to provide an overview over *scientific* literature, which relates to ACT. Specifically for the evolution of Computer-aided-design (CAD) tools from original Caltech Asynchronous Synthesis Tools (CAST) [67] to the finally published open-source Asynchronous Circuit Toolkit (ACT) [89] a narration will be provided. Table 4.2 provides a backwards chronologically ordered and commented list of significant publications related to ACT, i.e. publications that describe parts of ACT, classify its areas of application or provide an insight into its evolution from its predecessors. The later is especially true for the older ones. An underlined title means that the reference is describing the whole ACT flow in a comprehensible way, so it is recommended as a start to read, especially [35]. Core references for the creation of the Table 4.2 and the following narration are [35] and the *Language History*[2] section of the ACT documentation page [91].

Roughly 30 years ago the development of the Caltech Asynchronous Synthesis Tools (CAST), a language for hierarchical production rules, by Alan J. Martin's research group has begun[2]. The earliest comprehensive publication about it seems to be [67] in the early 2000s. However, still back in 1991 Alan J. Martin also presented CHP [68]. CHP is based on Hoare's Communicating Sequential Processes (CSP) language [69]. A newer publication about CHP by Alain J. Martin and Chris Moore is [70]. Then in 1995[2] Rajit Manohar came into play and implemented a new CAST [96]. The first prominent usage of CAST was at the design of the MIPS R3000 Microprocessor [23]. Approximately 1998 Andrew Lines and Uri Cummings launched the startup later named Fulcrum Microsystems, which licensed various tools developed at California Institute of Technology (Caltech), developed some proprietary extensions to CAST, as well as the Proteus flow [46], before it was bought by Intel. Some more use cases of CAST according to the *Language History*[2] are the development of programmable asynchronous pipeline arrays [24], the asynchronous processor SNAP [25] and the Lutonium microcontroller [26]. Ultimately ACT was created in 2005 by Rajit Manohar's research group to overcome some limitations of CAST (as [35] states). One of ACTs first versions was published [97] under the name *Hierarchical Asynchronous Circuit Kompiler Toolkit* (HACKT) as a part of David Fang's Ph.D. dissertation [71]. [35] and [72], also suggest that ACT was used in the design of the ULSNAP microcontroller [73] and the TrueNorth chip [3]. Finally ACT had its open-source release in 2019 [89]. Gradually additional components were published as open-source (at GitHub under asyncvlsi[3]) or at least teased by literature. Table 4.2 shows the literature in an ordered fashion.

[2] https://avlsi.csl.yale.edu/act/doku.php?id=history:start
[3] https://github.com/asyncvlsi

Title	Description	Authors	Publication date and reference
General Approach to Asynchronous Circuits Simulation Using Synchronous FPGAs	Paper to the tool `prs2fpga`, which implements mapping of ACs to synchronous Field Programmable Gate Arrays (FPGAs) to provide a fast gate level simulation technique.	R. Dashkin, R. Manohar	December 2021 [66]
Fluid: An Asynchronous High-level Synthesis Tool for Complex Program Structures	Fluid is a HLS tool that translates C programs into asynchronous dataflow circuits. Compatibility to ACT dataflow representation and CHP is implied.	R. Li, L. Berkley, Y. Yang, R. Manohar	September 2021 [74]
An Open-Source EDA Flow for Asynchronous Logic	Extensive presentation of ACT as open-source Electronic Design Automation (EDA). References alternatives (Haste [41], Balsa [44], Proteus [46]). History of ACT is examined. However, it is quite theoretical, hence no manual.	S. Ataei, W. Hua, Y. Yang, R. Manohar, Y. Lu, J. He, S. Maleki, K. Pingali	January 2021 [35]
Dali: A Gridded Cell Placement Flow	Provides gridded cell layout for ACs. Associated to ACT by [35], but tool itself not published yet.	Y. Yang, J. He, R. Manohar	November 2020 [75]
Cyclone: A Static Timing and Power Engine for Asynchronous Circuits	Cyclone is a timing and power analysis engine for ACs. Associated to ACT by [35], but also not published yet.	W. Hua, Y. Lu, K. Pingali, R. Manohar	May 2020 [76]
SPRoute: A scalable parallel negotiation-based global router	Global router for ACs. Based on ideas in [77]. Also associated to ACT by [35]. Newer version at [78]. On GitHub [98].	J. He, M. Burtscher, R. Manohar, K. Pingali	November 2019 [79]
Toward a digital flow for asynchronous VLSI systems	Conference Paper teasing ACT. Seems to precede [35]. Gives a good overview over the physical design flow.	S. Ataei, J. He, W. Hua, Y. Lu, S. Maleki, Y. Yang, K. Pingali, R. Manohar	November 2019 [80]
AMC: An Asynchronous Memory Compiler	Based on OpenRAM [81]. Generates on-chip memory with an (ACT-compatible) asynchronous interface to it. Claims competitiveness. On GitHub [99].	S. Ataei, R. Manohar	May 2019 [82]
An Open-Source Design Flow for Asynchronous Circuits	Brief report describing ACT as prospectively planned. Narration closest to official documentation [91].	R. Manohar	March 2019 [72]
Timing driven placement for quasi-delay-insensitive circuits	Presents A-NTUPLACE, a timing-driven placer for quasi-delay-insensitive (QDI) circuits. Based on [83]. Associated to the ACT by [35].	R. Karmazin, S. Longfield, C. T. O. Otero, R. Manohar	May 2015 [84]
cellTK: Automated Layout for Asynchronous Circuits with Nonstandard Cells	Introduces cellTK, a tool for physical design and *nonstandard* cell generation compatible with common ASIC flows. cellTK takes production rule sets of ACT as an input.	R. Karmazin, C. T. O. Otero, R. Manohar	May 2013 [85]
Proteus: An ASIC Flow for GHz Asynchronous Designs	Alternative design flow for ACs, which was developed by Fulcrum. Proteus and ACT are both successors of CAST [67].	P. A. Beerel, G. D. Dimou, A. M. Lines	September 2011 [46]
Profiling Infrastructure for the Performance Evaluation of Asynchronous Systems	Ph.D. dissertation for which ACT v0 variant (called HACKT) was developed. HACKT on GitHub [97].	D. Fang	May 2008 [71]
Design Tools for Integrated Asynchronous Electronic Circuits	Alternative Title: *CAST: A Suite of CAD Tools for Asynchronous VLSI*. Hence, explicit source for Caltech Asynchronous Synthesis Tools (CAST).	A. J. Martin, M. Nystroem, C. G. Wong	June 2003 [67]

Table 4.2: Selection of more significant literature related to ACT.

Table 4.2 starts bottom up with three references for historic context to ACT:

Design Tools for Integrated Asynchronous Electronic Circuits [67]
Provides an example of CAST, or in a broader sense the Caltech tools for ACs design in general still in Alan J. Martin's era at Caltech. It introduces itself as a phase I study, which aims to demonstrate the feasibility of industrial CAD tools for the design of ACs. Further it surmises that there maybe is a market for ACs, where low volume high profit chips are needed, hence for instance for defense or space. For *high level* design CHP is advertised.

Profiling Infrastructure for the Performance Evaluation of Asynchronous Systems [71]
This is the Ph.D. dissertation of David Fang, who was a student in the group of Rajit Manohar at Cornell University. It represents a huge step in coming from CAST to ACT and even publishing it under the name Hierarchical Asynchronous Circuit Kompiler Toolkit (HACKT). The Ph.D. thesis in general presents a framework for analysis of simulated execution of high level concurrent programs as a basis for synthesis of ACs and further optimization.

Proteus: An ASIC Flow for GHz Asynchronous Designs [46]
Proteus is here to remind of an alternative branch of all the tools around CAST and ACT, which is now owned by Intel after they bought Fulcrum. The Proteus flow is capable of translating high level Communicating Sequential Processes (CSP) programs into synthesizable register-transfer level (RTL) and further uses standard tools intended for synchronous design to generate an image-netlist. This *synchronous* image-netlist is then translated to a really asynchronous netlist. Various performance optimizations are claimed to be done by this step too.

Starting at the fourth entry from the bottom, the literature about the actual components of present ACT and the literature providing an overall view is listed. The overall view is provided by:

An Open-Source Design Flow for Asynchronous Circuits [72]
The report presents ACT as planned. It highlights the abstraction of asynchronous data transfers as channels, so actual implementation can be achieved with either QDI or BD style. In its overview it emphasizes CHP as a high level design entry. Then it elaborates about the multiple abstraction levels (CHP, HSE, gate-level, transistor-level) ACT is capable to represent an AC. It further traverses a custom design flow (logic design → logic simulation → circuit optimizations → analog simulation → layout → post-layout simulation → verify layout) and explains how ACT aims to fits all needs, before explaining plans how to deliver the usual automated digital design flow (design entry → logic synthesis → floorplanning → placement and routing → layout finishing). Overall the report does not go so much in depth with physical design as others do and so it provides a nice overview over (planned) ACT capabilities aside from physical design.

Toward a digital flow for asynchronous VLSI systems [80]
The paper focuses more on the physical part of the ACT design flow. It again starts by explaining the importance of CHP as a high level entry point, but quickly arrives at the *gates to GDSII* part, where it is emphasized that the physical design flow is also capable of accepting common design description (design exchange format (DEF), library exchange format (LEF)) not affiliated to ACT. A conversation from and to Verilog is also provided. The usual steps of physical design of digital circuits (technology mapping → floorplanning → placement → routing → layout finishing) are traversed. Using MAGIC is here only shown as one alternative to the usage of cellTK [85], Dali [75] and Cyclone [76] for arriving at GDSII format. Hence, the compatibility of the physical part of the flow to common formats (LEF, DEF) and the alternative physical flows are here even better examined as in [35].

An Open-Source EDA Flow for Asynchronous Logic [35]
This is the most extensive literature about present ACT published yet. At first the motivation to tackle the lack of tools for ACs design by providing an open-source EDA, which is also easy to use, because it aim to hide the details of physical implementation technology from the designer, is stated. Then there is given credit to the Galois framework [100] for providing needed parallelization in circuit timing and power usage analysis. Also all the three alternative flows Haste [41], Balsa [44], Proteus [46] are mentioned here. However, ACT should outdo them, because these and usually all especially free available design flows for ACs are restricted to only a certain type of AC (e.g. QDI). ACT aims to provide a solution for a very broad range of AC styles. It provides an overview over many other papers describing components of the ACT design flow. Further it examines timing and power usage analysis [76], design partitioning, floorplanning, placement [75], routing [79] and how to provide memory cells [82]. Currently the flow works stable with BD and QDI circuits. Future work will focus on supporting more classes of ACs and provide a more general timing analysis.

The references describing individual components of (or attachable to) ACT are:

cellTK: Automated Layout for Asynchronous Circuits with Nonstandard Cells [85]
cellTK is a *nonstandard* cell generator, which provides custom cells that fit into common standard cell flows. The authors claim that it mitigates drawbacks of custom standard cell design, while still offering the flexibility of a full custom design and the utility of technology mapping. It takes a production rule set or its transistor level description as input and outputs a corresponding physical implementation. Specifically the generated customized logic is packaged into the form of standard cells, which are compatible with established synchronous tool flows.

Timing driven placement for quasi-delay-insensitive circuits [83]
This reference focuses on the problem of timing driven placement for ACs. Most publications about this attempt to fit ACs and their timing assumptions into

preexisting timing analysis tools for synchronous design flows. However, here NTUPLACE, a timing aware placer especially suited to handling QDI circuits and their timing characteristics, is presented. In contrast to tools intended for synchronous flows NTUPLACE is aware of certain timing assumptions unique to QDI circuits (e.g. isochronic forks).

AMC: An Asynchronous Memory Compiler [82]
Presents an Asynchronous Memory Compiler (AMC), which generates SRAM modules with a BD data path and a QDI control. AMC outputs fabricable GDSII layouts with corresponding SPICE, Verilog and timing/power models. The authors claim that the memory designs by AMC are competitive with synchronous and asynchronous memories in literature. It is also stated that asynchronous SRAM, instead of clocked SRAM, has the potential to improve latency, reduce sensibility to fabrication variations and provides higher throughput. So in general AMC offers high average-case performance and supports non-uniform memory access times. There is an open-source distribution of AMC even with a SCMOS reference design on GitHub [99].

SPRoute: A scalable parallel negotiation-based global router [79]
SPRoute implements a two-phase maze routing approach, which initially exploits net-level parallelism, but is also capable of switching to fine-grain parallel processing of individual nets, if necessary. It is inspired by FastRoute [77] and uses the Galois framework [100] for parallelization. Further it is quite extensively examined, how parallelization can be exploited to reduce run time, which as chips grow larger is increasing rapidly, and how to to deal with a livelock (of concurrent threads).

Cyclone: A Static Timing and Power Engine for Asynchronous Circuits [76]
Cyclone is an engine for comprehensive timing and power usage analysis of ACs in general, so not only of QDI circuits. It takes as inputs an AC netlist, cell libraries and timing constraints for the desired AC type. Then after performing a multi-corner analysis it outputs the maximum cycle ratio, power consumption, and two types of timing slacks (performance slack of each gate and correctness slack of each timing constraint). The Galois framework [100] is here used for parallelization too. Cyclone is planned to be release for open-source. Furthermore the authors claim that, to their best knowledge, this is the first engine for timing and power analysis designated to ACs in public research.

Dali: A Gridded Cell Placement Flow [75]
Presents Dali, a said open-source (still not published) gridded cell placer for ACs. It claims to combine the shape regularity of standard cells and the size flexibility of custom cells. Various legalization algorithms for gridded cells are presented as well. The authors further claim that the placement and routing quality of the gridded cell design is comparable to standard cells produced by commercial tools with even less area. The paper stays vague about input and output formats.

Fluid: An Asynchronous High-level Synthesis Tool for Complex Program Structures [75]
Fluid translates C programs into asynchronous dataflow circuits, which can be further processed by ACT. It is claimed that the work extends existing dataflow synthesis techniques to a wider class of software programs by expanding the range of supported control flow structures. Also an improvement in terms of energy efficiency and throughput is claimed.

General Approach to Asynchronous Circuits Simulation Using Synchronous FPGAs [66]
Commercially available FPGAs only provide support for synchronous circuits (SCs). There has already been quite much effort to map AC designs to synchronous FPGAs and so this reference presents another solution to map ACs to FPGAs. A synchronous model of the original AC will be build based on event drive simulation concepts. The conversation preserves the communication protocols as well as the whole gate topology. As input a ACT description of the AC is taken. The output is a Verilog model. Furthermore it should be stated that the solution targets gate level simulation rather than RTL prototyping. The tool is available at GitHub [101].

After a hopefully enlightening overview over all the ACT literature has been provided, a further visualization of the design flow by the Yale university, which is alternative to the presentation in section 4.1 (especially Figure 4.1), follows. Figure 4.1 showed the design flow as it is explained on the official documentation page [91] and therefore usable for the public at the moment. It focused on portraying the different design level representations, as well as usable simulation levels and tools. Regarding physical design it only referred to MAGIC and the layout verification tools `lvp` and `gemini`.

The flow shown in Figure 4.7 instead focuses on physical design. Hence, the path from a high level representation to a low level expanded technology mapped representation is condensed to just one arrow. Note also that a possible conversion from and to Verilog representation is shown here and is also publicly available. However, using MAGIC is here only presented as one side path in the flow. Mainly it visually relates the tools cellTK [85], Dali [75], SPRoute [79] and Cyclone [76], which are unfortunately still not open-source.

In general all the yellow arrows are paths using tools, which are not publicly available at the moment, while the orange arrows represent paths, where all (open-source) tools needed, are already published. So the main path, which automates floorplaning, placement, routing and therefore finishes the layout to achieve fabricable GDSII format is at the moment only described in literature ([80] and [35] provide a good overview). The darker blueish boxes represent (optional) preexisting inputs to the flow. So it is assumed one starts with a design either described in (high level) ACT, Verilog or design exchange format (DEF), which is then converted to the according expanded and technology mapped version, there is a preexisting standard cell library in the library exchange format (LEF) and optionally also parasitics information in the standard parasitic exchange format

(SPEF) is available. The light blue boxes represent are intermediate descriptions usually created by cellTK to describe a new cell custom made for the current AC. cellTK is quite essential to model all the cells frequently used by ACs, but usually not available in common standard cell libraries. A SPICE model of those custom cells is also provided by cellTK to characterize timing and power usage in a later step, so it can be passed to Cyclone, which mutually exchanges data with the other tools in a loop as usual for a physical design routine. The final goal is of course to reach fabricable GDSII format representation, which is therefore highlighted by using dark blue.

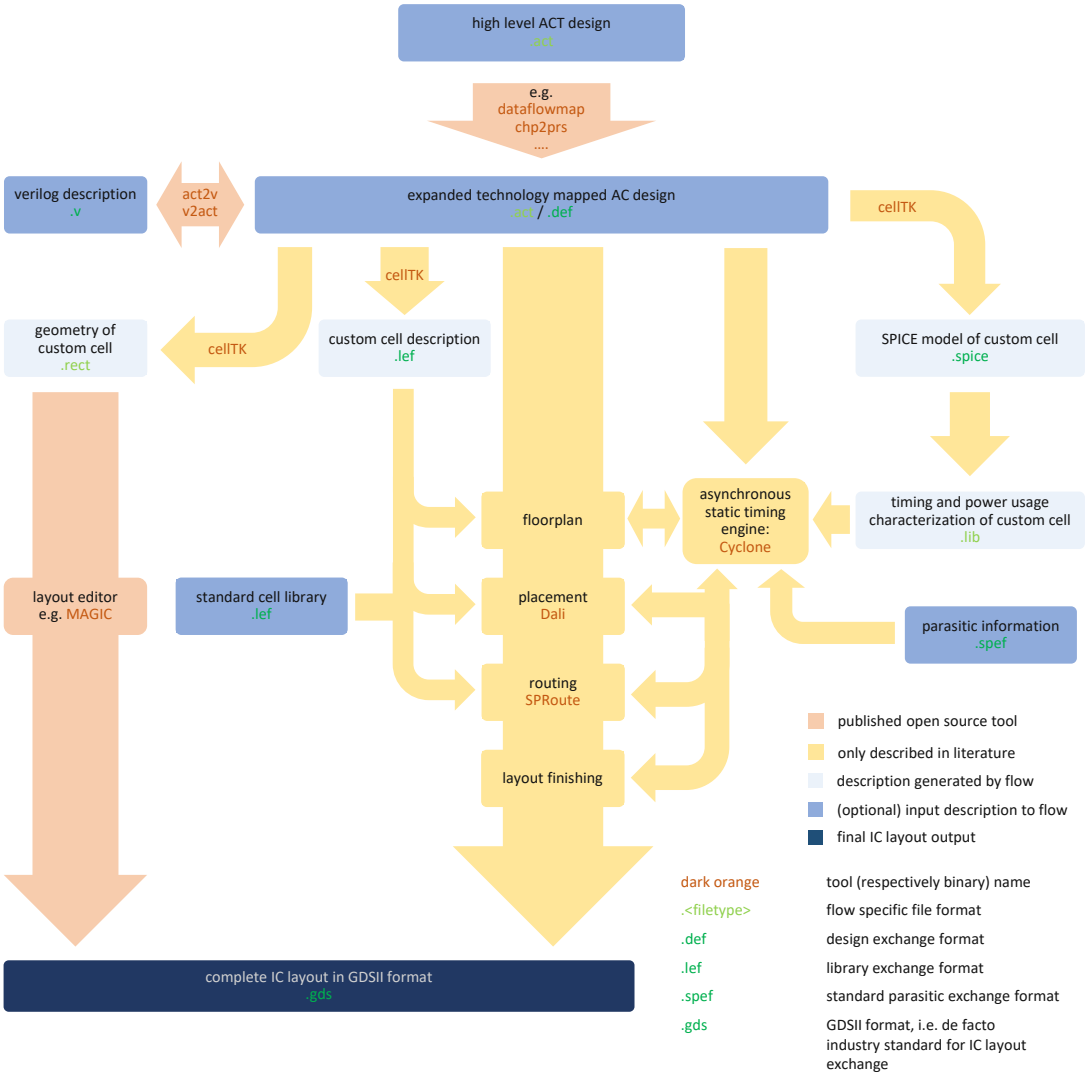


Figure 4.7: Physical design flow to generate layout corresponding to an ACT description.

In summary it can be said that section 4.1 describes the flow of a custom design approach, where focus is put on presenting the different design entries as well as different simulation levels, while the flow described above features more automated design, so that after describing a circuit in ACT (or alternatively Verilog or DEF) a physical implementation of that circuit is generated automatically. However, inconveniently all the tools needed for this automated flow are still not published, therefore it is reasonable that e.g. the official documentation [91] and some other sources declare currently the custom design flow with MAGIC as the main (or only) flow.

CHAPTER 5

Integration of Design Flows and Comparison

After the separate presentation of the design flow developed by the Embedded Computing Systems (ECS) group at TU Wien in chapter 3 and the design flow developed by the asynchronous VLSI and architecture (asyncVLSI) group at Yale University in chapter 4, now the centerpiece of this work, the integration of these two flows will be presented. Alternatively the integration of both can also be framed as integrating (some of) the tools from the Yale University to the fault-injection experiment flow of the TU Wien. Therefore various new tools/scripts, extensive complements and alternations to the existing design flow of TU Wien have been added. In particular a translation script from the production rule set (PRS) format, usual at the ECS group at TU Wien, to the Asynchronous Circuit Toolkit (ACT) language developed by asyncVLSI group at Yale University was added. Furthermore the gate level simulation tool Prsim has been incorporated to the engine for fault-injection experiments developed at TU Wien. It should also be noted that alongside this thesis a substantial refactoring (especially of the Python scripts) of various parts of the flow for fault-injection experiments was accomplished. In this chapter now at first an overview over the integrated flow and therefore all new paths for designing and simulating an asynchronous circuit (AC), which can now be taken, will be presented. Then the integration of Prsim as an alternative to Modelsim is examined. Parameter set and performance issues are discussed in section 5.3. In section 5.4 the results of (partial) reruns of the fault-injection experiments presented in [7], [63], [62] and [6] with the new integrated flow, which features besides a performance upgrade also a more strict fault model, follow as a proof of concept. A comparison of the capabilities of the the two original and the integrated flow follows.

5.1 Overview

Figure 5.1 shows the integrated design flow. It can be considered as a rough combination of Figure 3.1 from section 3.1 and Figure 4.1 from section 4.1, where some finer details shown in the original illustrations are omitted to keep the overall view compact. Blueish arrows and boxes indicate a derivation from the design flow of TU Wien, greenish too, but it specially marks an import relation of one Python script into another, and yellowish a derivation from the design flow of the Yale University. The different shades of blue are purely for aesthetics. The light yellow distinguishes the merely unused content of the Yale design flow regarding fault-injection experiments, but which is now also accessible to ACs described by Python scripts as usual to the work of TU Wien due to the translation tool `prs2act`. The dark red lines and boxes mark the revisions and complements alongside this thesis. In particular the tool `prs2act` has been developed to translate code described by the Python production rule package (pypr) to ACT. The respective Python libraries providing all necessities to describe a circuit and translate it to Very High Speed Integrated Circuit Hardware Description Language (VHDL) have been complemented to also support a translation to ACT. For practical use the concrete Python script `prs2act.py` is provided to take a PRS description of TU Wien as an input and output a corresponding ACT description. Given that it is already possible to utilize the *full* capabilities of the design flow developed at Yale University, but with pypr as design entry point standard at TU Wien. Note that in theory this does not only apply to the incomplete light yellow paths shown in Figure 5.1, but also to the physical layout design tools like e.g. MAGIC (as shown in Figure 4.1 in section 4.1) or the still not fully published (but specially developed at Yale University) physical design flow illustrated by Figure 4.7 in section 4.2. Furthermore the dark red arrow from the box labeled *low level ACT* to the `autosetup.py`/`dbworker.py` complex indicates that circuits described in ACT are now also suitable for fault-injection experiments. No translation to VHDL is needed. Prsim is used instead of Modelsim to simulate ACT descriptions. A distinct `tbgen.py` has been written to generate a corresponding ACT testbench. `dbworker.py` got an extensive refinement to enable the capability to manage fault-injection experiments with circuits described in ACT. The logging of Prsim is obviously different, so a `prsim.log` file is introduced. `autosetup.py` does currently not entirely support simulations with ACT code, however great gratitude should be expressed to Robert Najvirt, who made the `autosetup.py` script compatible to the heavily refactored `dbworker.py`. Compatibility to preceding tools (like e.g. `prscom.py`) as well as to the Structured Query Language (SQL) database still persists. Nevertheless one noteworthy change is that Prsim simulations do model metastable behaviour, therefore the fault model now also features a metastability fault. This resulted in an additional columns in one table in the SQL database.

Table 5.1 finally gives a synoptic overview over all the revisions and complements made to integrate the two design flows into one. Also recall that the changes to `autosetup.py` and `dbworker.py` were immense also for pure refactoring purposes.

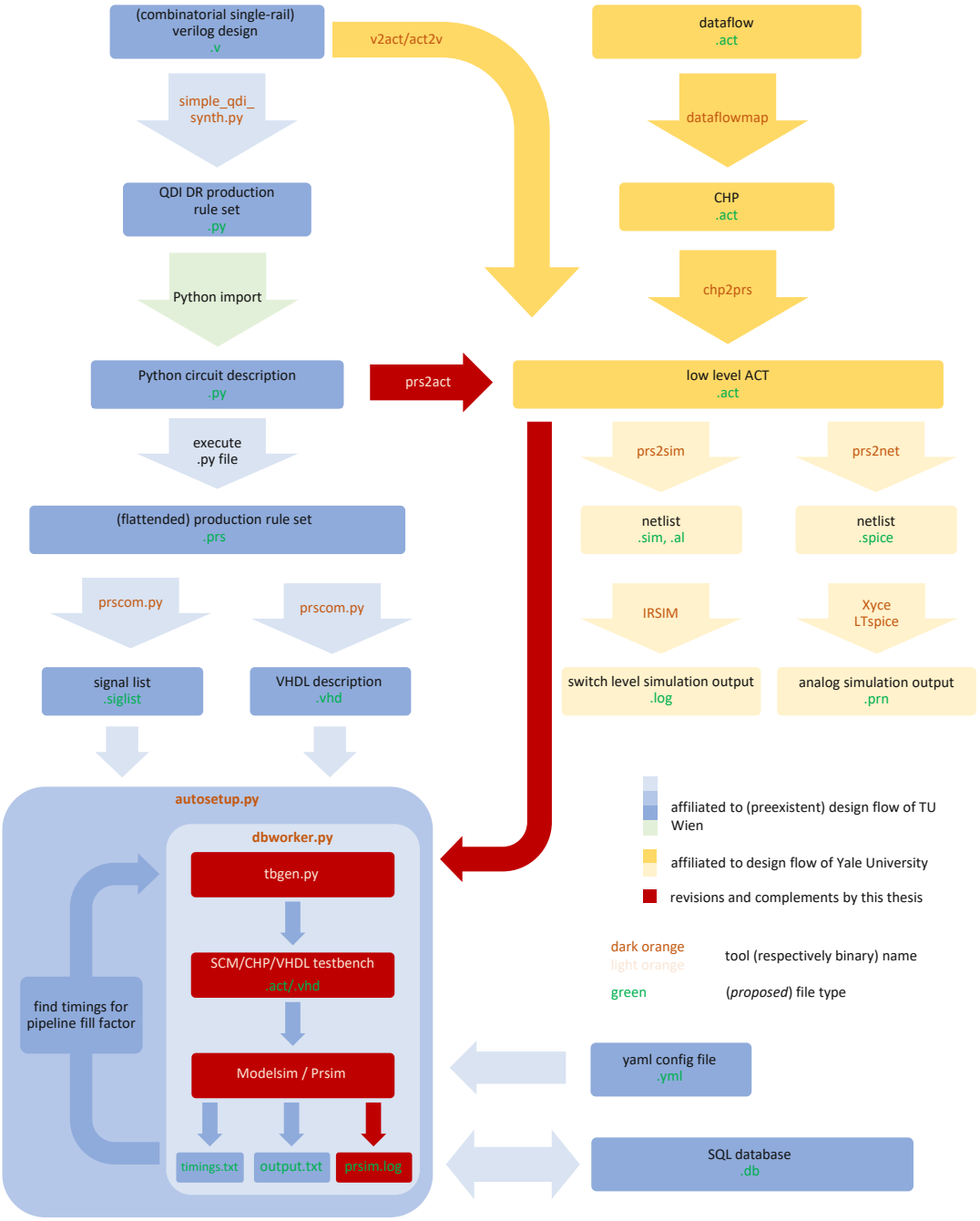


Figure 5.1: Integrated flow.

New or revised component	Description
pypr library	The pypr library was extended to support the translation from PRS format to ACT.
`prs2act.py`	Python script utilizing the translation capabilities of pypr from PRS to ACT.
`dbworker.py`	Has been heavily refactored. Now also supports fault-injection experiments with circuits described in ACT.
`autosetup.py`	Still not entirely compatible with ACT, but at least compatible with the refactored `dbworker.py` thanks to Robert Najvirt.
Prsim	The new simulation tool as it is provided by the ACT repository [89].
`tbgen.py`	Distinct Python script to generate testbench for ACT circuits.
`prsim.log`	Prsim outputs a different log file format.
YAML config file	It is now configurable which simulation tool (Modelsim or Prsim) should be used. Also various Prsim specific configurations have been added.
SQL database	New column to indicate fault due to metastability has been added to the according results table.
`prscom.py`	Some minor changes were made so that e.g. the signal list of signals appropriate for fault-injections is usable for Prsim.

Table 5.1: Refinements and complements done to integrate the design flow of TU Wien and the one of Yale University to one.

5.2 Integration of Prsim

Prsim is the gate level simulation software specially developed for the Yale flow. It is as the rest of the flow open-source. Here its incorporation into the fault-injection engine of the TU Wien flow as an alternative to Modelsim is discussed. Hence, to make the `dbworker.py` script and related scripts compatible for the usage of Prsim an extensive overhaul of the existing Python code was needed. Now individual classes with standardized interface orchestrate the usage of each simulation software. Nevertheless the `dbworker.py` does still mainly prepare folders and files, which are specified by YAML configuration files, and then executes the according simulation software within the corresponding context. For the preparation of the new testbench files for ACT a new `tbgen.py` script has been written, as well as many code fragments for dealing with the new simulation log format of Prsim, but especially interesting is the generation of ACT testbenches and their interactions with Prsim. Currently there are two different simulation approaches or testbench generation approaches, but beforehand a few clarifications how Prsim interacts with a given testbench shall be provided. A testbench for Prsim is a

regular circuit designed the usual way with e.g. dataflow, Communicating Hardware Processes (CHP), PRS etc. The original target circuit described by a process in ACT is instantiated in this testbench circuit and eventually connected to some helper circuits like a sink. Prsim also takes a script file as an input, which specifies how to interact with the testbench circuit. Prsim takes here two different input files, where in Modelsim maybe everything is specified in just one VHDL testbench file. The reason is that Prsim circuit descriptions can not (or it is very cumbersome to) wait for a specified amount of time and pull down or up signals in between in a sequential order. So delays need to be coded into components connected to the target circuit in the testbench. Two different ways of accomplishing this are presented by the testbench code for Prsim in Listing 5.1, which is illustrated by Figure 5.2, and the alternative shown in Listing 5.2 and Figure 5.3.

For both illustrations the target circuit is represented by a box labeled with UUT (unit-under-test) the amount of input ports and output ports is not important in that example as just the delay management should be demonstrated and not the actual operation the target circuit performs. However, notice that the quasi-delay-insensitive (QDI) dual-rail (DR) standard is used here. Hence, each logic data bit is represented by two rails, the *true* and *false* rail. For the fault-injection experiments common at TU Wien each individual rail, as well as signals of control logic need to feature a different and changing signal delay per token. One way of achieving this is by setting the delay of the input buffer per rail/signal accordingly, simulate, stop when the delay is supposed to change, change it, continue simulation and repeat this cycle. Here the actual testbench ACT code shown in Listing 5.1 does not do much besides attaching input buffers to all input signals, for which delays can be repeatedly changed during runtime of Prsim, and determining the phase (DATA- or NULL-phase) of each QDI DR port by completion detectors (CDs) so that the CYCLE TRIGGER component can trigger each time a transition from one phase to another has occurred and therefore the simulation should be halted and delays adjusted. Remember that Prsim takes two files as an input to run a simulation. The testbench file of this approach is shown in Listing 5.1 and illustrated by Figure 5.2. The other contains a list of commands including start and stop commands for simulation similar to a tcl script when using Modelsim. However, the real magic of this approach does not occur in this command file itself, but in the mini-scheme file included into this file. Mini-Scheme is a minimal version of the functional programming language Scheme [86]. For specific mini-scheme reference see the README in the `miniscm`[1] folder in the ACT core repository. New commands for Prsim can be defined in mini-scheme or in other terms the mini-scheme language can be used to augment the command(-line) interface to Prsim. So the procedure for one simulation works conceptually as following:

1. Run Prsim with the according ACT testbench and prescripted command file.
2. The first few commands in the command file will categorize input and output signals of the target circuit and probably apply the reset signal.
3. Then delays will be configured for the first token and the simulation is started within a special environment set up by the mini-scheme language.

[1] https://github.com/asyncvlsi/act/tree/master/miniscm

4. As soon as one of the CDs detects a phase change the CYCLE TRIGGER component will indicate this to the environment and certain mini-scheme procedures will halt the simulation and update the delays accordingly.
5. The simulation then continues with adjusted delays before it is halted again, when the CYCLE TRIGGER component triggers again.
6. This continues until certain end conditions for the simulation are reached.

The above approach may sound simple and light weighted, but the simulation performance is very bad due to the recurring halting and continuing of the simulation. Furthermore the procedures written in mini-scheme needed for this approach are quite lengthy, complex and difficult to adapt. Hence, the approach shown in Figure 5.3 with the corresponding testbench ACT code shown in Listing 5.2 is much more performant and easier to maintain. The target circuit is shown in Figure 5.3 and is connected to the testbench. Again the count of input and output ports are not important here, because it is again neglected here what actual operation the target circuit may perform. The deciding difference is that for this approach every more complex procedure has been implemented in CHP instead of relying on mini-scheme. So all test logic is directly implemented in the ACT testbench including the target circuit and everything can be simulated in one run without halt in between. The SOURCE and SINK components in Figure 5.3 are implemented in CHP. The reason why there are multiple sources and sinks is that each source produces one token. So the schematic view in Figure 5.3 actually shows a simulation, where three tokens (i.e. data values) are sent to the target circuit. There is one source per token, because for each token the distinct delays need to be hard coded into a delay component beforehand. The delay components for each data rail and acknowledge signal are labeled with Δ_+/Δ_- to hint that the delay for pull up and pull down is obviously different. All other Muller C-elements (MCEs) and OR gates basically form a merge logic for the input port of the target circuit and a fork logic for the output port. These two logics are quite specific, because they need to comply to the CHP described sources and sinks. Each token is really distributed to its own source, which is then deciding only once, because only this way each rail of each token could get its specific preconfigured delay element. In particular one simulation works as follows:

1. Run Prsim with the according ACT testbench and the much more light weighted command file.
2. The first few commands in the command file will categorize input and output signals of the target circuit and probably apply the reset signal (as before).
3. Then the simulation is started to run. In this approach everything is done without ever halting the simulation again.
4. Initially all output rails/signals of one SOURCE block (of Figure 5.3) are '0'.
5. The data rails of the first SOURCE block get their values assigned (DATA-phase). Through the delay components and OR gates the signals progress to the target circuit.
6. When the acknowledge signal of the target circuit rises, the SOURCE 1 block applies again '0' to each rail (NULL-phase).

7. When the acknowledge signal of the target circuit then falls again, the SOURCE 2 block starts its transaction. This continues until the last source block is done. It is coded in CHP that the SOURCE 2 block starts its transaction only after the transition with the SOURCE 1 block is done.
8. When for the first time the output rails of the target circuit indicate completion, the SINK 1 block raises its acknowledge signal.
9. The acknowledge signal of SINK 1 progresses through the delay component and the OR gate to the target circuit.
10. The target circuit will again apply '0' to each output rail. The acknowledge signal of SINK 1 will fall again and the cycle continues with SINK 2.

The second approach is by far more stable, performant and easier to maintain than the first one, even if there seems to be more redundant logic. Building a larger testbench circuit around the target circuit can be done more generic in CHP than any mini-scheme coding. Mini-scheme coding is in general cumbersome. Prsim is much faster, if simulations are not halted.

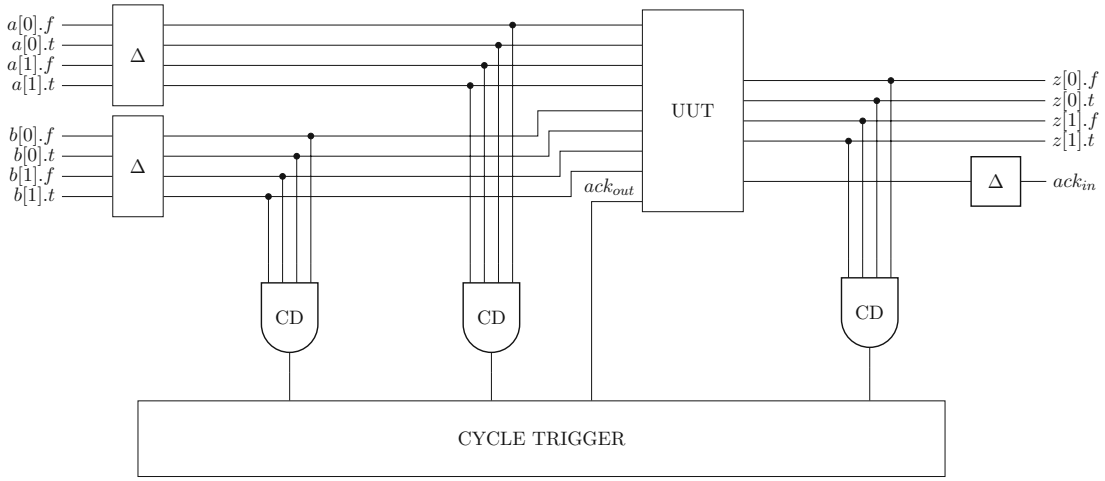


Figure 5.2: Testbench design for fault injection experiments using Prsim, where inputs are orchestrated by the mini-scheme command line augmenting language.

```
1  import "alu_flat.act";
2  import "auxiliary.act";
3
4  uut uut_inst;
5
6  completion_detector<2> completion_detector_a(.z=uut_inst.a);
7  completion_detector<2> completion_detector_b(.z=uut_inst.b);
8  completion_detector<2> completion_detector_z(.z=uut_inst.z);
9  // buffers include the delay components
10 dualrail_buffer<2> dualrail_buffer_a(.o=uut_inst.a);
11 dualrail_buffer<2> dualrail_buffer_b(.o=uut_inst.b);
12 bit_buffer bit_buffer_ack_in(.o=uut_inst.ack_in);
13 bit_buffer bit_buffer_reset(.o=uut_inst.reset);
14
15 bit_buffer after_reset_delay;
16
17 // for each change on an input (.i) the simulation is halted.
18 cycle_trigger<6> cycle_trigger_inst(.i={completion_detector_a.done,
       completion_detector_b.done,alu_inst.ack_in,alu_inst.reset,alu_inst.ack_out,
       completion_detector_z.done});
```

Listing 5.1: Testbench ACT code for fault-injection experiments using Prsim. Inputs are orchestrated by the mini-scheme command line augmenting language.

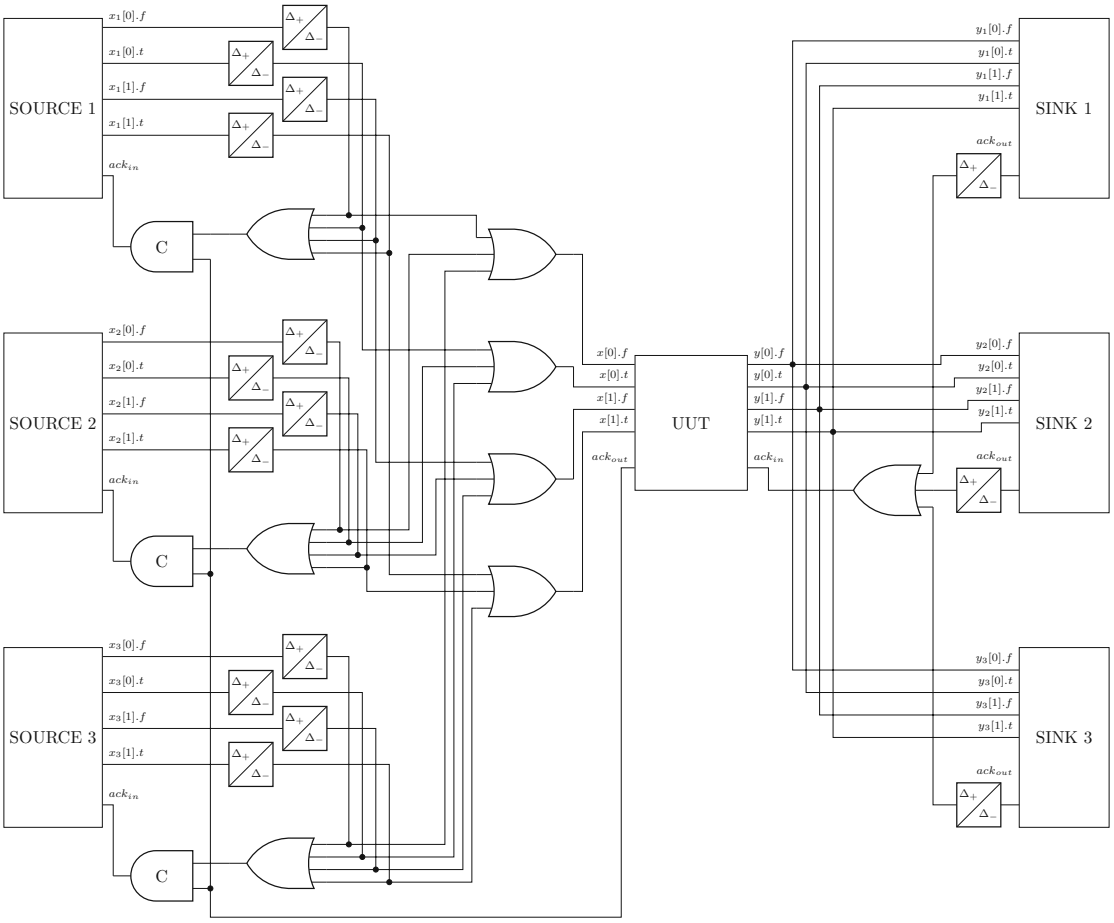


Figure 5.3: Testbench design for fault-injection experiments using Prsim with sources and sinks described via CHP.

```
1  import "chp_tb.act";
2  import "alu_flat.act";
3
4  // aMx1of2<2> is the type for dual-rail encoding of a two bit vector with
       acknowledge signal
5  aMx1of2<2> x;
6  aMx1of2<2> y;
7  uut uut_inst;
8  uut_inst.x=x.d.d; /* x.d.d refers to the raw dual-rail data vector */
9  uut_inst.y=y.d.d;
10 uut_inst.ack_in=y.a;
11 uut_inst.ack_out=x.a;
12 sdt_chp_tb sdt_chp_tb_inst(.x=x, .y=y);
13 // omitted in the schematic view
14 uut_inst.reset=Reset;
15 reset_delay reset_delay_inst(.in=Reset);
```

Listing 5.2: Testbench ACT code for fault-injection experiments using Prsim with sources and sinks described via CHP.

5.3 Covered Parameter Space with Resulting Performance

Section 3.2 has already shown in Figure 3.6 the parameter span of interest for previous fault-injection experiments done at TU Wien. The respective section alongside with the presented papers there is advised for comprehensibility of the offered parameter span. The target parameter span for fault-injection experiments performed with the integrated flow is given by the Cartesian product shown in Equation 5.1.

$$\begin{gathered}\text{CIRCUIT} \times \text{DATA\_WIDTH} \times \text{BUFFER} \times \text{LOGIC} \times \text{GATE\_DELAY} \times \text{PLF} \\ = \\ \begin{bmatrix}\text{ADDER}\\ \text{MULT}\\ \text{ALU}\end{bmatrix} \times \begin{bmatrix}4\\ 8\end{bmatrix} \times \begin{bmatrix}\text{WCHB}\\ \text{DeadlockingWCHB}\\ \text{InterlockingWCHB}\\ \text{DualCDWCHB}\end{bmatrix} \times \begin{bmatrix}\text{DIMS}\\ \text{NCL}\\ \text{NCLX}\end{bmatrix} \times [0.9, ..., 1.1] \times \begin{bmatrix}0.25\\ 0.5\\ 1\\ 2\\ 4\end{bmatrix}\end{gathered} \tag{5.1}$$

The three different circuits in the target parameter span can be further characterized as following:

- ADDER: Simple Ripple-Carry-Adder circuit with a buffer before and after the according logic.
- MULT: Pipelined multiplier with the number of stages related to DATA_WIDTH.
- ALU: Very small arithmetic logic unit featuring an adder, OR and NOT operation. Shown in Figure 5.4.

For the different weak-conditioned-half-buffer (WCHB) types again see section 3.2. The logic styles are explained in section 2.3. The gate delays are randomized between 0.9 and 1.1 nanoseconds. The pipeline-load-factor (PLF) works as described in [7]. The fault parameter span is the same as in section 3.2. Hence, the fault location (i.e. the signal chosen for a single-event-upset (SEU)) is determined randomly, as well as the fault polarity (force to '0' or '1') and the time of injection. In fact the `autosetup.py` script, which conceptionally still works like in [7] is used to calibrate the circuits for a given parameter set.

Obviously the concrete chosen parameter set influences the simulation performance, as the choice of the simulation software does. Table 5.2 presents a few examples of reachable simulations per second (SPS) with the Intel Core i7-4790K CPU at 4-4.6GHz speed on 8 threads. The chosen simulation software, the token count, the concrete circuit and the data width mainly influence the performance.

TOKENS	Circuit	DW	Modelsim SPS	Prsim SPS
5	ALU	4	≈43	≈57
5	ALU	8	≈43	≈47
5	MULT	4	≈44	≈58
10	ALU	4	≈40	≈39

Table 5.2: Performance comparison of Modelsim and Prsim.

Table 5.2 is definitely no comprehensive performance evaluation. Currently the flow is not optimized and presumably bottlenecks, because of implementation details. Hence, the main intent here is just to show quickly that there is definitely a performance gain for the average case of up to 24% by using Prsim instead of Modelsim. The average case uses 5 tokens, because two are used for initial pipeline filling to then inject at the desired PLF, which happens within the next two tokens. Then with the final token all possible effects of the SEU should be observed. The examples with a token count of 10 are just there to estimate the impact of this parameter. The multiplier is also there to provide simply a bit of variation. In particular the following conclusions can be drawn from Table 5.2:

- Prsim is more performant than Modelsim in the average setup used for fault-injection experiments.
- A higher token count, which per definition demands a longer simulation time, leads obviously to less SPS.
- The different circuit types in the parameter space seem to be all similar enough to not have a significant influence on the SPS.
- The data width has a huge impact on the SPS for Prsim simulations. Probably this results from the testbench construction for Prsim circuits as described in section 5.2. For Modelsim this is not the case.
- Given that the SPS for Modelsim simulations are not influenced by the other parameters as strong as Prsim simulations are, Modelsim simulations presumably just face a bottleneck for reasons outside of the parameter choice.

The results of the simulations of the ALU will be presented and interpreted in section 5.4. Therefore everything is quite centered about this circuit. Many more simulations in the parameter span of Equation 5.1 have been run mainly for testing the integrity of Prsim. Hence, the main focus of these runs was always to check the comparability of the results yielded by Prsim to those of Modelsim regardless of performance.

5.4 Comparison of Fault-Injection Experiment Results

This section compares the results of fault-injection experiments performed by Modelsim with those performed by Prsim. The concrete parameter set given by Equation 5.2 is used for the example. So basically only the PLF is varied here, because the calibration for it is the main feature of the `autosetup.py`. The representative aspects that are suitable for discussion are already plenty enough in this example.

$$\begin{gathered}\text{CIRCUIT} \times \text{DATA\_WIDTH} \times \text{BUFFER} \times \text{LOGIC} \times \text{GATE\_DELAY} \times \text{PLF} \\ = \\ \Big[\text{ALU}\Big] \times \Big[4\Big] \times \Big[\text{WCHB}\Big] \times \Big[\text{DIMS}\Big] \times [0.9, ..., 1.1] \times \begin{bmatrix} 0.25 \\ 0.5 \\ 1 \\ 2 \\ 4 \end{bmatrix}\end{gathered} \tag{5.2}$$

The two tools process the equivalent simulation tasks with the given parameter sets quite differently. Great effort was put into equalizing the outputs, nevertheless several fixed constraints contribute to an inevitable differentiation in the results. The most significant are the following:

- The SEU mechanic is different. The injection duration differs necessarily for Prsim with equal configuration. In Modelsim the force command is used to inject an upset into a target signal. The original signal value is instantaneously overridden at the configured timestamp. If the configured injection duration has expired, the signal value again instantaneously returns to its original value. Prsim features a special SEU command, which also immediately applies the upset to the target signal. However, the signal does not necessarily return to its original value after the configured injection duration. It depends on the individual gate delay, which drives the signal, and the time offset to the nearest regular signal transition. Hence, the actual injection duration for Prsim deviates from the configured one and can not be determined easily in advance, but the configured injection duration is a minimum.
- Prsim simulations are sensible to metastability. It is introduced as a new error type. A signal is considered metastable, if it is undecidable from its voltage level, whether it should be interpreted as '1' or '0'. A signal can enter this state, if, e.g. by a SEU, its regular charging is interrupted at a critical instant. Metastability can then spread to other signals until eventually the whole circuit is infected. Various other fault types may be originally caused by (internal) metastability. A metastability error is only reported, if an output signal becomes metastable.
- The deadlock and tokencount error type are equivalent in some sense. The deadlock error is reported, if Modelsim runs in a timeout, because presumably the circuit is stuck. Hence, not all expected tokens have arrived at the sink. Tokencount

errors are used by Prsim, because Prsim never runs in a timeout, it just stops early, probably skipping remaining tokens. So basically both cases indicate that a genuine deadlock has very likely occurred, deadlock is exclusive to Modelsim and tokencount to Prsim.

The results of hundreds of thousands of simulations of the QDI 4-phase DR AC shown in Figure 5.4 over five different PLFs are shown in Figure 5.5 (simulations performed by Modelsim) and Figure 5.6 (Prsim). The error counts on the vertical axis are given as percentage of the corresponding injection counts. The injection counts are different per PLF, because to keep an equal density of injections, the injection count must be adapted, if the injection time frame changes caused by a different PLF.

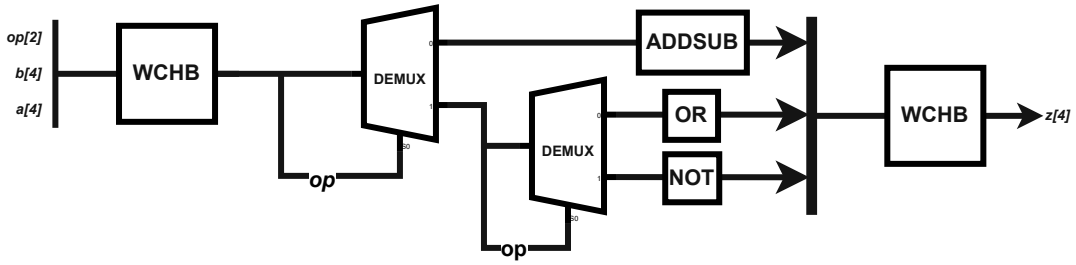


Figure 5.4: QDI 4-phase DR AC implementing a minimal arithmetic logic unit.

The tasks delegated to Modelsim and Prsim were equivalent. Therefore an overall similarity of the results is clearly present. However, there are differences urging for reasoning. The injection count is usually between 80,000 and 350,000 per PLF. The timing error results show quite different patterns. In a narrow sense the reason for that is still unknown. In a broader sense it is reasonable by the differences already explained above. Probably longer injection durations or internal metastability are causing it. Furthermore note that the different error types shown in Figure 5.5 and Figure 5.6 also overlap regularly.

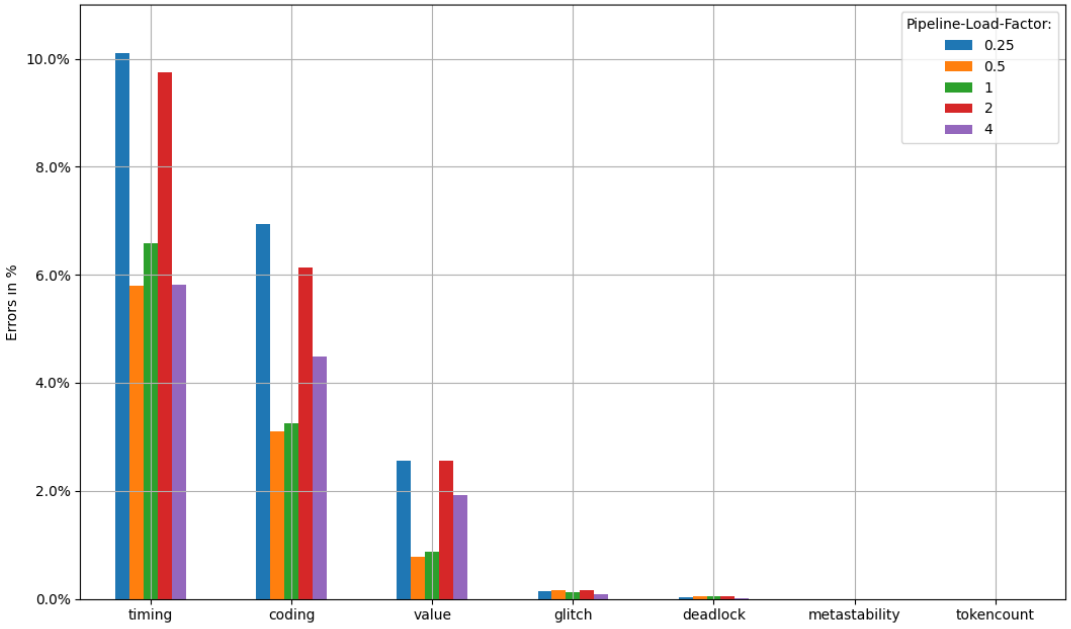


Figure 5.5: Percentage of different error types in Modelsim simulations.

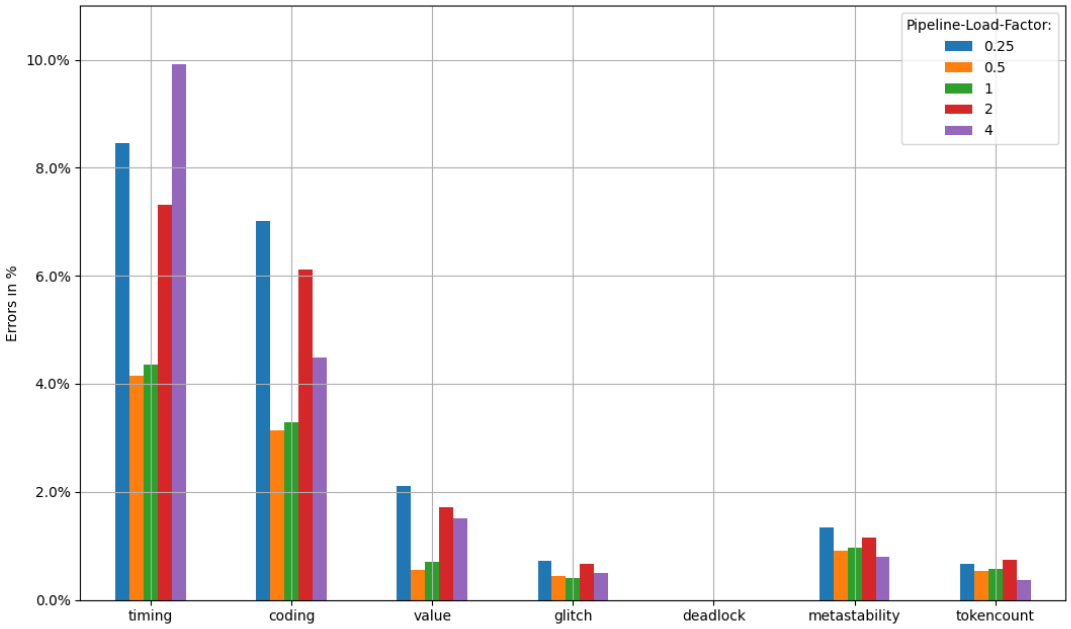


Figure 5.6: Percentage of different error types in Prsim simulations.

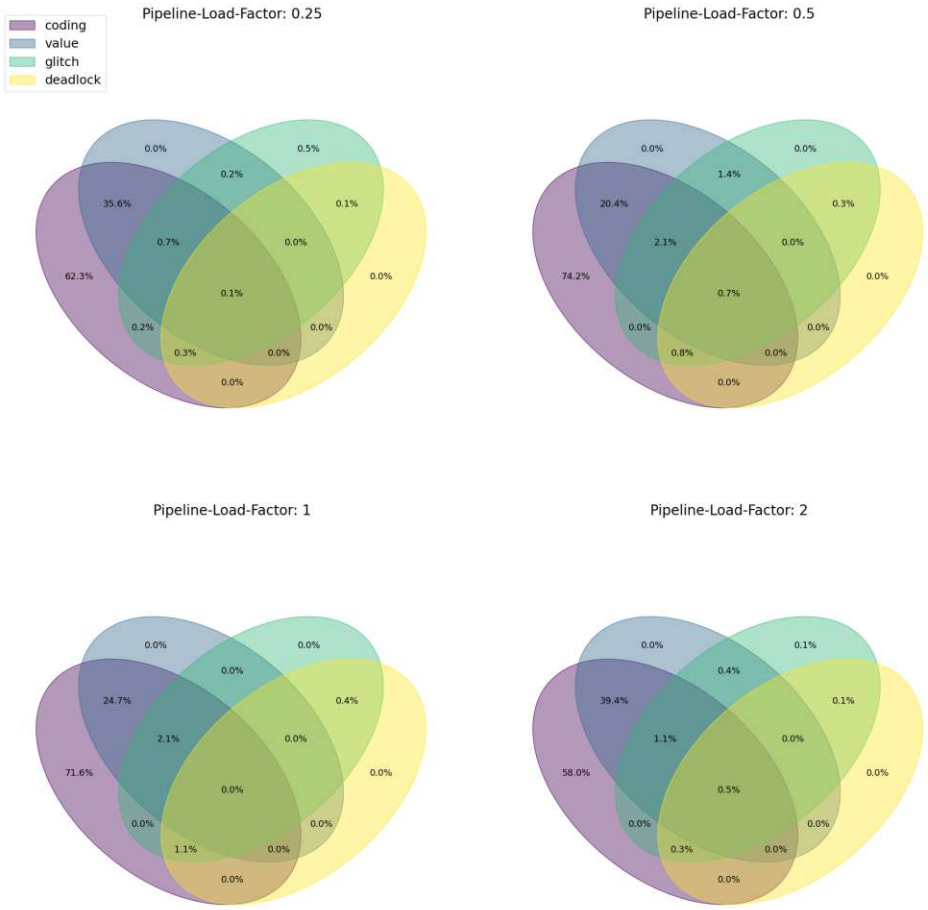


Figure 5.7: Overlaps of different error types in Modelsim simulations.

Figure 5.7 shows the overlapping of different error types for the simulations performed with Modelsim. The underlying data is the same as before, but the plot for the PLF of 4 is omitted for reasons of space. Also the timing errors are omitted, because usually more than 95% of all errors are also timing errors. The following points are considered especially noteworthy:

- The most common error type beside the timing is the coding error, because each illegitimate drive to '1' at a signal influencing the data output rails results quite likely in a coding error.
- Exclusive value errors do not happen. Mostly they occur together with coding errors. This makes sense by cycling through the different scenarios. A '0' can be illegitimately driven to '1' on one of the data output rails and usually this alone already indicates a coding error. If not, then probably this will not result in an error at all. If it does, it will result in a glitch, as well as a drive to '0' will do at a critical instant.
- Glitches are rare. This is reasonable, because in most cases therefore a CD needs to

be *tricked*. This is not easy with SEUs, because the probability to hit exactly the deciding signal at a critical time frame is low.
- Deadlock errors only occur together with glitches, because it needs at least a glitch to deadlock a standard WCHB.
- The PLF seems not to influence the error type overlapping significantly. The seen deviations can be reasoned by the randomness involved in each simulation.

Figure 5.8 shows the overlapping for the simulations performed with Prsim. Conclusions originally concerning Figure 5.7 obviously also apply here, if applicable. E.g. value errors sill do not occur exclusively. So complements special to error type overlapping for Prsim simulations are the following:

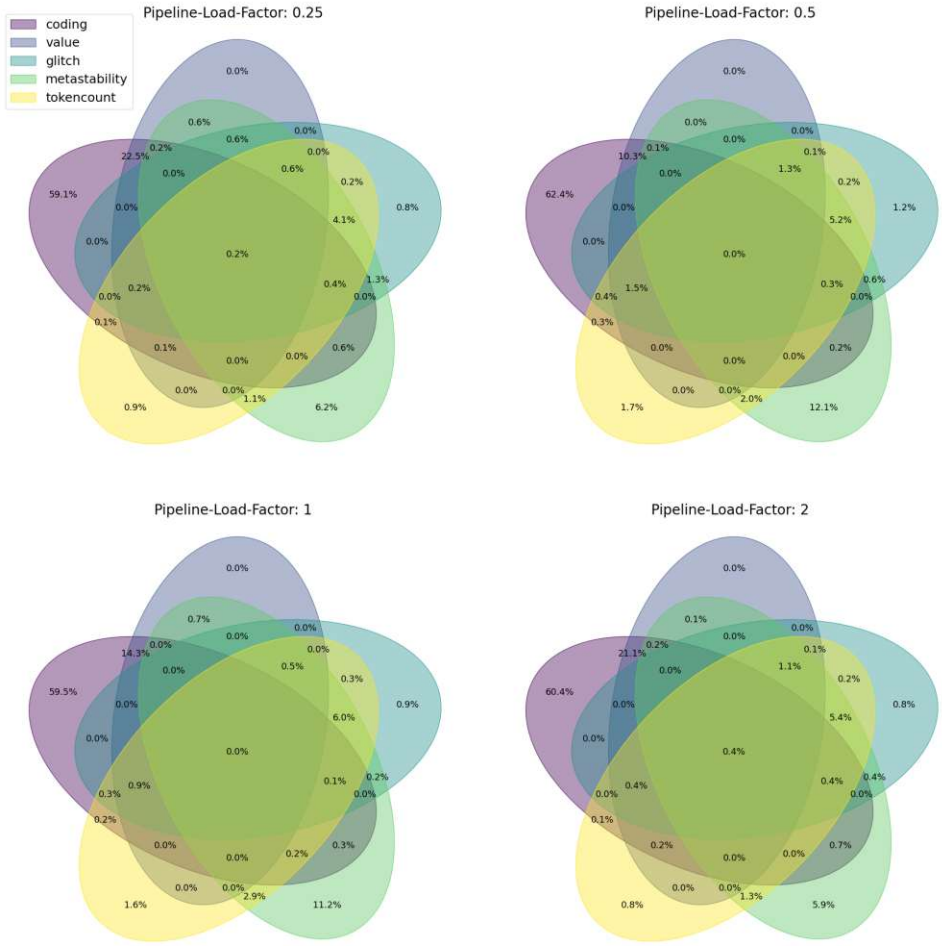


Figure 5.8: Overlaps of different error types in Prsim simulations.

- The deadlock error type has been switched for the tokencount error type and the new metastability error type is featured as well.
- Interestingly metastability occurs mainly exclusive, i.e. the circuit can continue

operation even with (short) metastability occurrence at the output in most cases. This is reasonable, because in many circumstances an metastable signal at e.g. an output data rail will just stabilize again before the sink finally catches it. If it does spread, glitches and eventually tokencount errors are likely to follow.
- Most glitches and tokencount errors are caused by metastability.
- Tokencount errors occur also exclusively. This occurs, if the last NULL-phase is missing. The last data-phase was maybe processed correctly, but without the final NULL-phase one token is considered as missing. Due to slightly different operation of the testbenches this is not reported, when using Modelsim, and for sure does not count as a deadlock.

To further characterize the similarities and differences between Prsim and Modelsim simulations with equal configurations, Figure 5.9 provides a scatter plot over injection timestamps (horizontal axis), error types (vertical axis) and simulation tool choice (color). The Modelsim dots are a bit lifted and the Prsim dots lowered, so each dot can be well seen. Hence, the following conclusion can be drawn from Figure 5.9:

- The injection time frame with PLF one is the smallest, because, if source and sink are equally fast, the tokens traverse the pipeline in the fastest way possible.
- Characteristic gaps are shared among Modelsim and Prsim injection time lines for timing, value and coding errors with a few outliers.
- Metastability and tokencount errors do obviously only appear for Prsim simulations, while deadlock errors are exclusive to Modelsim.

Finally Figure 5.10 shows the overlaps of timing, coding, value and glitch errors of Modelsim and Prsim simulations. The PLF one has been picked for this example. Plots with other PLFss show no significant differences. The following conclusions can be drawn:

- The proportion of timing errors that are exclusive to their simulation tool, are maybe also caused by different conceptional views about when a timing variation is actually illegitimate in the implementation. There is some ambiguity.
- Coding errors overlap are as expected.
- Value errors were expected to also overlap a bit more. However, reasons for exclusivity need to be investigated further.
- It is good to see that Prsim recognizes every glitch that Modelsim does too. Additional glitches are most likely caused by (internal) metastability as Figure 5.8 and the corresponding discussion have shown.

Synoptically it can be said that Modelsim and Prsim report mostly the same errors, where they are expected to do so. In a wide sense deviations can be generally blamed on the differences pointed out at the beginning of this section. However, in a narrow sense there are still investigations required beyond the scope of this thesis. The investigated QDI 4-phase DR AC shown in Figure 5.4 was just exemplary, nevertheless other ACs show similar characteristics. In particular a plain adder and multiplier circuit with varying buffer and logic styles have also been examined behind the curtain. The specific AC

from Figure 5.4 has been chosen, because it is compact, easy to understand, nonetheless contains multiple combinational and control logic elements.

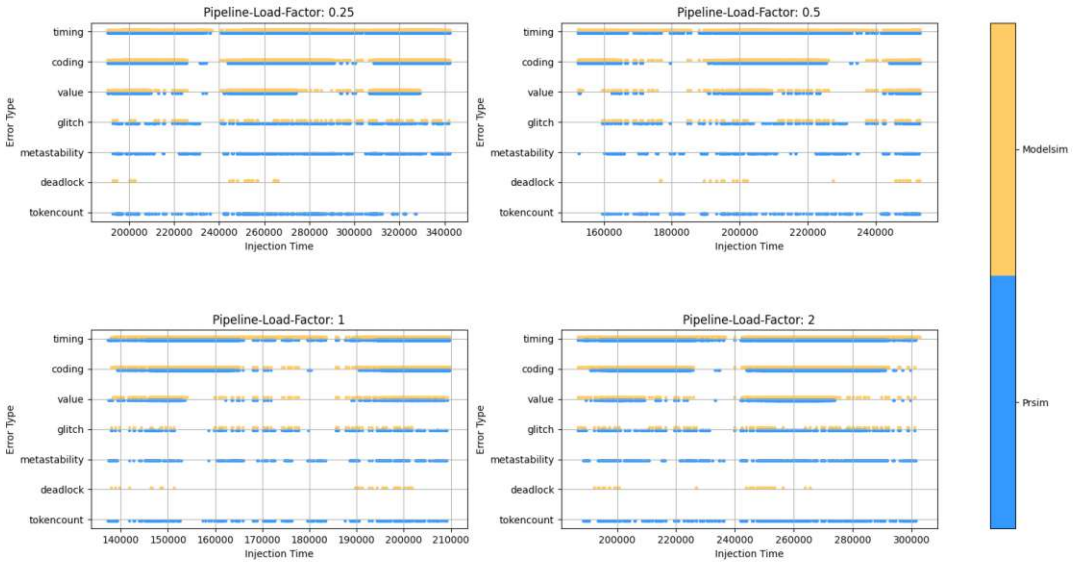


Figure 5.9: Comparison of Modelsim and Prsim injection timestamps and their impact.

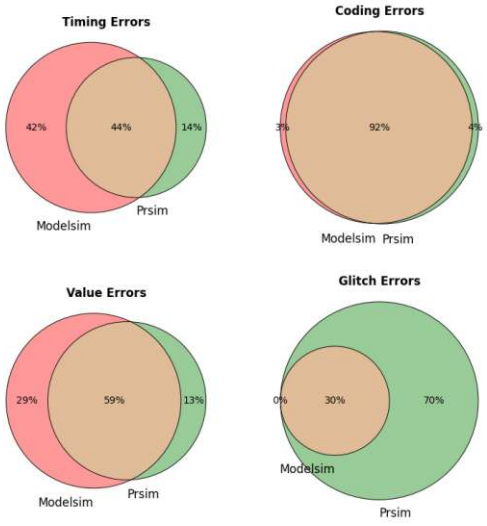


Figure 5.10: Overlapping of error types of Modelsim and Prsim simulations for PLF 1.

5.5 Capability Comparison

As already mentioned multiple times in this thesis, the motivation of the design flow of TU Wien and the flow of Yale University are quite different. The former is focused on gate level fault-injection experiments due to current research interests at TU Wien. The later aims for a complete coverage of AC design from high level description to fabricable GDSII format. Hence, the Yale flow is definitely more elaborated, especially when it comes to (industrial) chip design. The asyncVLSI group participated in the development of impressive industrial showcases like e.g. the TrueNorth [3]. Thereby massive industrial experience influenced the currently published flow of the Yale University, i.e. ACT. However, the flow of TU Wien does feature competitive components in the author's opinion. The high level design in Python provided by pypr is an easy entrance, overall handy and versatile. I.e., multiple design levels are covered by pypr and code can be as generic as Python allows. Also an extensive and easily expandable standard library for AC buffers and logic is provided. ACT does feature multiple design levels and a standard library as well, but the scope Python itself provides is often a clear advantage to the standalone languages bundled in ACT. Furthermore, the fault-injection engine composed of `autosetup.py` and `dbworker.py` adds substantial to the TU Wien flow, if fault-injection is an aspect of interest. The Yale flow by itself, in particular Prsim, provides here just a simple SEU command similar to the force command of Modelsim.

The two original flows from TU Wien and Yale University are also compared against their integration to one flow as described in section 5.1. Thereby the enhancements by integration are apparent alongside the original differences. Table 5.3 shows a first clear comparison of unambiguous properties. A cross (×) indicates that a property applies, a dash (−) that it does not. The classification (very) low, medium or high is frequently used to indicate the level of satisfaction provided by the respective flow for the property. Otherwise simple adjectives are used to classify the appliance of a property. Mostly the integrated flow inherits the superior classification. Showcase examples are the standard library support and the design entry points.

Additionally to Table 5.3 each individual property will be discussed below. Note that some more detailed aspects will be mentioned without a special explanation. Please refer to chapter 3 for a comprehensive explanation of the design flow of TU Wien and to chapter 4 for the Yale flow.

- **VHDL support**: The flow of TU Wien necessarily supports VHDL, because simulations are performed with Modelsim. Therefore the PRS representation has to be translated to VHDL by `prscom.py`. The flow of Yale University currently does not support VHDL at all to the author's awareness. The integrated flow preserves the conversion from PRS to VHDL, but does not complement it.
- **Verilog support**: Verilog is supported by both original flows. `simple_qdi_synth.py` of the TU Wien flow features the synthesis of a combinational single-rail Verilog design to a QDI DR PRS design. Hence, this feature is contributed to the integrated flow. The Yale flow offers the tools `v2act` and `act2v`, which provide a *structural*

Property	Design flow of TU Wien	Design flow of Yale University	Integrated flow
VHDL support	×	–	×
Verilog support	×	×	×
bundled-data support	×	×	×
dual-rail support	×	×	×
standard library support	medium	high	very high
fault-injection support	high	minimal	high
physical design support	–	×	×
footprint on development system	high	low	high
publication status	planned	published	considered
Circuit Description			
design entry points	few	many	plenty
high level support	medium	high	very high
parameterization	high	medium	high
low level support	medium	high	high
Simulation			
gate level simulation	×	×	×
switch level simulation	–	×	×
analog level simulation	–	×	×
metastability modeling	–	×	×
simulation performance	low	high	high

Table 5.3: Comparison of various properties of the three different flows.

translation to and from Verilog. *Structural* translation means that only the module structure of a hierarchical design is translated. So the relation of a top level module to its included modules stays intact. Probably arbitrary logic (e.g. given by production rules) at lower levels is not translated. For e.g. `act2v` this means that the hierarchy of ACT processes is reconstructed with Verilog modules, but statements inside `prs {...}` are ignored. The integrated flow inherits the tools `v2act` and `act2v` as they are.

- **bundled-data support**: The flow of TU Wien and Yale University both support bundled-data (BD), but DR is default for both. So does the integrated flow.
- **dual-rail support**: The default AC type all flows operate with.
- **standard library support**: pypr of TU Wien offers an extensive and easily expandable standard library featuring different buffers and logic for ACs. The Yale flow does so as well. At the moment the standard library coming with pypr is quite competitive. However, it is almost certain that Yale will surpass the TU Wien in time. The integrated flow shines here, because both libraries can complement each

other.

- **fault-injection support**: Fault-injections are the main focus of the TU Wien flow. Hence, it features a sophisticated fault-injection engine. The simulation tools affiliated to the Yale flow just provide usual SEU/force commands one would expect from every serious simulation software. It is one of the main contributions of this thesis to incorporate Prsim as an alternative to Modelsim into the fault-injection engine for the integrated flow.
- **physical design support**: The flow of TU Wien does not support physical design at all. The flow of Yale University does offer extensive physical design capabilities. Currently there are two paths supported, a full custom layout by the open-source tool MAGIC and a special path designed at Yale with self-made tools like `cellTK` (custom cell generator), `Dali` (physical placement tool), `SPRoute` (routing tool), `Cyclone` (asynchronous static timing engine) etc. It is another main contribution of this thesis that the physical design tools of the Yale flow are now accessible to pypr designs in the integrated flow.
- **footprint on development system**: Modelsim as part of the TU Wien flow occupies several gigabytes on the hard drive. The occupation by the Yale flow is clearly below one gigabyte, depending a bit on the number of installed modules/repositories of it. The integrated flow inherits the large footprint of Modelsim.
- **publication status**: The publication of the flow of TU Wien is planned mid- to long-term. The flow of the Yale University is open-source and published under [89]. Probably the integrated flow will be published one day as enhancement to the TU Wien flow, but at the moment there are no concrete plans.
- **design entry points**: The main entry point of the TU Wien flow is pypr, which provides by itself multiple levels of AC description. A pure dataflow description with almost automatic choice of the according standard library elements is possible, as well as a specification of each individual gate and its connections. For the Yale flow ACs are usually described by data flows, with CHP or by plain ACT with production rules specified inside `prs {...}`. Furthermore Fluid [74] is a tool capable of translating C code to asynchronous data flows. Here the integrated flow shines again, because the possible design entries to chose from increase.
- **high level support**: Both original flows provide AC synthesis from a data flow description. This is already quite high level. However, the Yale flow is additionally compatible to Fluid [74] and its custom languages are also a bit less verbose than Python with pypr. pypr functionality could be packed in even more compact Python functions, but theoretically this is also possible with ACT code. The integrated flow offers here for sure an enhancement, because of more choice and combination possibilities.
- **parameterization**: The capability to describe ACs in Python allows immense parameterization. Hence, this is one strength of the TU Wien flow. The Yale flow offers some basics like templates for processes, but nothing extraordinary. The integrated flow inherits the capabilities of the original flows here, but provides no enhancements.

- **low level support**: The flow of TU Wien does not aim for physical design, so no annotations designated to it are available. Nevertheless, pypr allows the specification of custom attributes per port, entity etc. This could be used to specify lower level details. In contrast the Yale flow provides even the possibility to annotate type, size and leakage of individual transistors. The integrated flow does not add to this.
- **gate level simulation**: The gate level simulation tool used by the TU Wien flow is Modelsim. The Yale flow provides its own special gate level simulation tool, Prsim. The integrated flow enhances the fault-injection engine by allowing Prsim as an alternative to Modelsim.
- **switch level simulation**: Switch level simulation is a term used in the self description of the open-source tool IRSIM. Basically IRSIM takes a netlist as an input and performs a transistor level simulation. Finally the waveform view looks very similar to a gate level simulation. The flow of the Yale University supports the translation of ACT to a IRSIM compatible netlist. The integrated flow inherits this.
- **analog level simulation**: The flow of Yale University also provides an analog level simulation (i.e. a SPICE simulation) using the open-source version of Xyce. Therefore ACT code is translated to a SPICE compatible netlist. The integrated flow again inherits this functionality.
- **metastability modeling**: While metastability modeling is neglected in the current Modelsim setup for the TU Wien flow, Prsim of the Yale flow features unavoidably metastability modeling. Hence, fault-injection experiments with integrated flow yield different results depending on the chosen simulation software.
- **simulation performance**: Prsim of the Yale flow is more performant in the average case than Modelsim of the TU Wien flow. See also section 5.3.

Synoptically it can be said that the integrated flow successfully enhances the variety of design paths significantly. The two standard libraries are practically combined, because a component designed by pypr can always be translated to ACT. The synthesis of single-rail Verilog logic can now also be utilized for ACT designs. Many more (high level) design entries are there to chose. The low level annotations missing in pypr can be substituted by attributes. These can then be easily applied as the corresponding annotations in ACT by e.g. a few simple helper scripts. All the other simulation tools/levels are accessible to ACs designed by pypr now as well. Especially the analog simulation with Xyce might be interesting. Finally also the physical design flow of the Yale University can be fueled by ACs (partially) described by pypr.

CHAPTER 6

Conclusion and Outlook

The contribution of this thesis is split into two aspects. First the related work coverage is extensive and therefore hopefully enlightens the reader about the historical path of asynchronous circuits (ACs) and their prospects today. Chapter 2 started with a history about ACs, which is not complete, but probably one of the rarer approaches to outline one, oriented on multiple source branches. The state of the art section then continued with presenting examples mainly from the field of brain-inspired hardware. The Spiking Neural Network architecture (SpiNNaker) [1], the neuromorphic system for simulating large-scale neural models Neurogrid [2] and the neurosynaptic processor TrueNorth [3] have been presented. The examples show that, despite being aced out by synchronous design, ACs still produce prominent innovations. Also there is a rising awareness in the async community that the lack of design tools especially dedicated to ACs is one of the major obstacles for them to shine. Nevertheless some tools usually from University origin have been developed over the years, which were featured at the end of the state of the art section. The last section of chapter 2 covered fundamentals of AC design. However, probably an even more extensive covering of related work was done by section 3.2 and section 4.2, which covered the creation process of the two design flows serving as a basis for this thesis. The stories behind the two flows, as derivable from published literature, were covered there. The flow of TU Wien (chapter 3) is strongly focused on fault-injection experiments. Alongside [7] and [6] this thesis contributes to describing the flow. Also a massive overhaul of many old functionalities was done. Publication of the flow is planned for mid- to long-term. The flow of the Yale University aims for a complete coverage of all design steps needed from conceptual high level descriptions to fabricable GDSII format. Additionally to the official documentation page [91] this thesis offered an overview with easy examples to comprehend the flow and its versatility. Physical design was not covered, because it is no real concern at TU Wien with the focus on fault-injections at gate level. Explanations and examples to the Yale flow are of course unofficially, because there is no affiliation of the author to the group behind the Yale flow.

Now secondly this thesis came up with its own integration approach of the flows from TU Wien and Yale. This was covered in chapter 5. The intention behind this was to achieve a connection of the TU Wien flow to all capabilities the Yale flow offers. The flow of Yale University is by far more elaborated and covers many fields like e.g. physical design, which the TU Wien flow does not. So for the research of TU Wien it is for sure a progress to access this functionality with circuits still designed with the Python production rule package (pypr) as common at TU Wien. However, not only a translation from AC descriptions used at TU Wien to the Yale flow was contributed, but also the fault-injection engine as part of the TU Wien flow was made fit for experiments with Asynchronous Circuit Toolkit (ACT) code from the Yale flow. Especially this complement has been utilized to provide a proof of concept by running about a million fault-injections with a circuit originally designed with pypr, translated to ACT and then simulated with Prsim (i.e. the gate level simulation tool of the Yale flow) orchestrated by the fault-injection engine of the TU Wien flow. As a reference the simulations have also been performed by Modelsim as usual at TU Wien and were then compared. The simulation tools, as well as other details are different and so the results were necessarily not exactly equal, but overlapped to a certain extent, see again section 5.4. Finally a comprehensive comparison section completed the chapter. There the capabilities of the two original and the integrated flow were compared. Hence, for the integrated flow it was discussed what complements to the overall capabilities have been contributed by the integration. It is remarkable that comprehensive fault-injection experiments for ACT circuits are now possible thanks to the connection to the fault-injection engine. Also the connection the TU Wien flow now has to the physical design capabilities of the Yale flow is enriching. In general the high level AC design approaches using pypr by TU Wien are quite competitive, but the overall coverage of aspects needed for chip design of the Yale flow is not even slightly contested by it.

Points the thesis considered out of scope should be addressed here as an outlook. Theoretically the integration of the two flows could be even more tight. A translation from higher levels of ACT/Communicating Hardware Processes (CHP) to the production rule set (PRS) format as implemented at TU Wien could be added. It was spared, because the fault-injection engine could utilize Prsim for ACT code anyways. Then much optimization potential and refactoring would be applicable for large parts of especially the Python code of the fault-injection engine before an eventual open-source publication. Then already section 5.3 and section 5.4 themselves stated that a more extensive performance and result analysis would probably dig up some enlightening outcomes. However, this thesis mainly focused on the extensive discussion of the two original flows and the creation of the integrated flow. Therefore the million fault-injections should only present a small proof of concept example.

The future research interests of the Embedded Computing Systems (ECS) group at TU Wien are currently re-evaluated, because the project, which initiated the fault-injection interest, is ending. Nevertheless there is a strong interest in ACs in general and especially the design flow of the Yale University (ACT) is seen as very promising. This thesis can

be interpreted as a first approach to it.

List of Figures

List of Tables

Listings

Glossary

2-phase handshake protocol

Asynchronous handshake protocol by transition indication. It needs 1 transition for the request and for the acknowledge signal per data transition. See section 2.3. 9–12, 77

4-phase handshake protocol

Asynchronous handshake protocol by value indication. It needs 2 transitions for the request and for the acknowledge signal for one data transition. See section 2.3. 8–14, 20, 21, 23, 27, 29–31, 36, 63, 67, 77, 81, 86

Asynchronous Circuit (AC)

Sequential digital logic circuit, which does not use a global clock for synchronization. Instead, individual components use handshaking for synchronization. See section 2.3 ix, 1–10, 17, 18, 20, 23, 27–31, 33, 34, 43–48, 51, 52, 63, 67, 69–74, 77, 84–86

Asynchronous Circuit Toolkit (ACT)

See chapter 4, https://github.com/asyncvlsi/act or https://avlsi.csl.yale.edu/act/doku.php. ix, 2, 4, 5, 33–38, 40, 42–45, 47–49, 51, 52, 54–56, 58, 59, 69–72, 74, 77, 79, 81, 83

Asynchronous VLSI and architecture (asyncVLSI) group

Research group at Yale University in New Haven (Connecticut), which published the Asynchronous Circuit Toolkit (ACT). See http://avlsi.csl.yale.edu/, ix, 1, 33, 40, 51, 69

Bundled-data (BD)

Asynchronous handshake protocol using a request and an acknowledge signal. See section 2.3. 9, 36, 44–46, 70

California Institute of Technology (Caltech)

A prestigious private research University in Pasadena, California, which developed (predecessors of) various tools discussed in this thesis. See https://www.caltech.edu/. 4–6, 19, 42, 44, 84, 85

Caltech Asynchronous Synthesis Tools (CAST)
A language for hierarchical production rules developed at California Institute of Technology (Caltech). See [67]. 4, 42–44

Caltech Intermediate Form (CIF)
"Low level graphics language for specifying the geometry of integrated circuits" [87]. 33

Communicating Hardware Processes (CHP)
Programming notion for description of asynchronous circuits (ACs). See https://www.cs.yale.edu/flint/cs428/notes/chp_introduction.pdf, http://vlsi.cornell.edu/~fang/hackt/hac/CHP.html or [70]. 33, 34, 36, 37, 42–45, 55–57, 59, 71, 74, 77, 81, 84

Communicating Sequential Processes (CSP)
Basis of Communicating Hardware Processes (CHP). A formal language first described by C. A. R. Hoare in [69]. 7, 8, 42, 44

Completion detector (CD)
Circuit component to detect if a dual-rail (DR) circuit is in NULL- or data-phase. See section 2.3. 12, 14, 55, 56, 65

Computer-aided-design (CAD)
Use of computers to aid in design tasks. In this thesis often synonymously used with Electronic Design Automation (EDA). 6, 7, 42, 44, 84

Delay-insensitive (DI)
Timing model where arbitrary gate and wire delays are allowed, as long as they are positive and finite. See section 2.3. 15, 27, 28, 30

Delay-insensitive-minterm-synthesis (DIMS)
Synthesis strategy for quasi-delay-insensitive (QDI) dual-rail (DR) logic. See section 2.3. 12–14, 21, 77, 85

Dual-rail (DR)
Encoding of one bit in two rails, the true- and false-rail. See section 2.3. 8, 9, 12–14, 18, 20–23, 29–31, 36, 55, 63, 67, 69, 70, 77, 81, 84, 86

Electronic Design Automation (EDA)
Category for Computer-aided-design (CAD) tools specially designated to the design of electronic systems. 43, 45, 84, 85

Embedded Computing Systems (ECS) group
Research group at TU Wien (Austria), which developed the the design flow in chapter 3. ix, 1, 17, 27, 31, 51, 74

Field Programmable Gate Array (FPGA)
An integrated circuit designed to be reconfigurable in the field after manufacturing. 1, 5, 43, 47

Graphic Design System II (GDSII)

Industry standard for Electronic Design Automation (EDA) of integrated circuits. ix, 2, 33, 69, 73

Hardware description language (HDL)

E.g. VHDL or Verilog. 18, 22

Muller C-element (MCE)

State holding gate, which changes its state only, if all inputs agree. See section 2.3 for a further characterization. 4, 10, 12–15, 31, 36–41, 56, 77, 81

NCL with Explicit Completeness (NCLX)

Optimization to top delay-insensitive-minterm-synthesis (DIMS) and NULL-Convention-Logic (NCL). See section 2.3. 14, 15, 77

NULL-Convention-Logic (NCL)

Logic design style using threshold gates. See section 2.3 or e.g. [48]. 5, 14, 85

Pipeline-load-factor (PLF)

A PLF of e.g. 2 indicates that the source is twice as fast as the sink. Vice versa for a PLF of 0.5. See [7] for a further explanation. 27–29, 60–63, 65–68, 78

Production rule set (PRS)

Description concept for (asynchronous) circuits originally established at California Institute of Technology (Caltech). However, in this thesis it usually refers to a concrete implemented format to represent asynchronous circuits (ACs) of the design flow of TU Wien. ix, 18–24, 27, 28, 30, 51, 52, 54, 55, 69, 74, 81

Python production rule package (pypr)

Python package, which provides various classes and functions for description of an asynchronous circuit (AC) in Python. Part of the design flow of TU Wien (chapter 3). See also [6]. vii, 1, 2, 18, 20, 23, 27–31, 52, 54, 69–72, 74

Quasi-delay-insensitive (QDI)

The QDI model allows arbitrary gate delays, but for wires the *isochronic fork* [55] condition must apply. See section 2.3. 8, 15, 18, 20–23, 25, 27–31, 43–46, 55, 63, 67, 69, 77, 81, 84, 86

Register-transfer level (RTL)

Used by hardware description languages to present a high level view of a digital circuit. 1, 8, 18, 44, 47

Simulation Program with Integrated Circuit Emphasis (SPICE)

SPICE is an open-souce analog level simulator for electrical circuits. 34

Simulations per second (SPS)

The SPS metric is used to compare the performance of different simulation tools. 60, 61

Single-event-upset (SEU)

Here an upset (i.e. illegitimate value change) of a single signal for a limited amount of time. 28, 60–62, 66, 69, 71

Structured Query Language (SQL)

Standard language for the communication with databases. 17, 18, 20, 23, 25–27, 29, 31, 52, 54, 79

Synchronous Circuit (SC)

Sequential digital logic circuit, where state changes of memory elements are synchronized by a (global) clock signal. ix, 1, 8, 47

Verilog

IEEE standard hardware description language. 7, 18, 20–22, 28, 30, 34, 45–47, 49, 69, 70, 72, 81

Very High Speed Integrated Circuit Hardware Description Language (VHDL)

IEEE standard hardware description language. 18, 20, 23–26, 30, 52, 55, 69, 81

Very large-scale integration (VLSI)

The process of combining millions of MOSFETs onto a single chip to create an integrated circuit. 33, 34

Weak-conditioned-half-buffer (WCHB)

Most common buffer for QDI 4-phase DR ACs. See e.g. [88]. 12–14, 17, 20, 21, 23, 27, 28, 31, 32, 60, 77, 81

YAML

Originally an acronym for *Yet Another Markup Language*. It is the human-readable language for configuration files. Used by the flow of TU Wien (chapter 3). 18, 20, 23–25, 30, 54, 81

Bibliography

Literature

[1] E. Painkras, L. A. Plana, J. Garside, *et al.*, „Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation“, *IEEE Journal of Solid-State Circuits*, vol. 48, no. 8, pp. 1943–1953, 2013. DOI: 10.1109/JSSC.2013.2259038.

[2] B. V. Benjamin, P. Gao, E. McQuinn, *et al.*, „Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations“, *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014. DOI: 10.1109/JPROC.2014.2313565.

[3] F. Akopyan, J. Sawada, A. Cassidy, *et al.*, „Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip“, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015. DOI: 10.1109/TCAD.2015.2474396.

[4] K.-S. Chong, B.-H. Gwee, and J. S. Chang, „Energy-efficient synchronous-logic and asynchronous-logic fft/ifft processors“, *IEEE Journal of Solid-State Circuits*, vol. 42, no. 9, pp. 2034–2045, 2007. DOI: 10.1109/JSSC.2007.903039.

[5] O. C. Akgun and Y. Leblebici, „Energy efficiency comparison of asynchronous and synchronous circuits operating in the sub-threshold regime“, *Journal of Low Power Electronics*, vol. 4, no. 3, pp. 320–336, Dec. 2008. DOI: 10.1166/jolpe.2008.185. [Online]. Available: https://doi.org/10.1166/jolpe.2008.185.

[6] F. Huemer, „Contributions to efficiency and robustness of quasi delay-insensitive circuits“, Ph.D. dissertation, TU Wien, 2022.

[7] P. Behal, „Quantitativer vergleich der empfindlichkeit von delay-insensitiven design templates gegenüber transienten störungen“, M.S. thesis, TU Wien, 2021.

[8] D. E. Muller, „Theory of asynchronous circuits, report no. 66“, *Digital Computer Laboratory, University of Illinois at Urbana-Champaign*, 1955.

[9] H. C. Brearley, „Illiac ii-a short description and annotated bibliography“, *IEEE Transactions on Electronic Computers*, vol. EC-14, no. 3, pp. 399–403, 1965. DOI: 10.1109/PGEC.1965.264146.

[10] R. E. Miller, *Switching Theory*. New York: John Wiley, Aug. 1979, ISBN: 9780882757599.

[11] C. J. Myers, *Asynchronous Circuit Design*. Wiley-Interscience, Jul. 2001, ISBN: 9780471415435.

[12] S. H. Lavington, *A history of Manchester computers*, 2nd ed. British Computer Society, 1998.

[13] R. Ibbett, „The university of manchester mu5 project“, *IEEE Annals of the History of Computing*, vol. 21, no. 1, pp. 24–33, 1999. DOI: 10.1109/85.759366.

[14] J. Woods, P. Day, S. Furber, J. Garside, N. Paver, and S. Temple, „Amulet1: An asynchronous arm microprocessor“, *IEEE Transactions on Computers*, vol. 46, no. 4, pp. 385–398, 1997. DOI: 10.1109/12.588033.

[15] S. Furber, P. Day, J. Garside, N. Paver, and J. Woods, „Amulet1: A micropipelined arm“, in *Proceedings of COMPCON '94*, 1994, pp. 476–485. DOI: 10.1109/CMPCON.1994.282880.

[16] S. Furber, J. Garside, P. Riocreux, *et al.*, „Amulet2e: An asynchronous embedded controller“, *Proceedings of the IEEE*, vol. 87, no. 2, pp. 243–256, 1999. DOI: 10.1109/5.740018.

[17] S. Furber, J. Garside, S. Temple, J. Liu, P. Day, and N. Paver, „Amulet2e: An asynchronous embedded controller“, in *Proceedings Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997, pp. 290–299. DOI: 10.1109/ASYNC.1997.587182.

[18] J. Garside, S. Temple, and R. Mehra, „The amulet2e cache system“, in *Proceedings Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996, pp. 208–217. DOI: 10.1109/ASYNC.1996.494452.

[19] S. Furber, D. Edwards, and J. Garside, „Amulet3: A 100 mips asynchronous embedded processor“, in *Proceedings 2000 International Conference on Computer Design*, 2000, pp. 329–334. DOI: 10.1109/ICCD.2000.878304.

[20] J. Garside, S. Furber, and S.-H. Chung, „Amulet3 revealed“, in *Proceedings. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1999, pp. 51–59. DOI: 10.1109/ASYNC.1999.761522.

[21] S. Furber, J. Garside, and D. Gilbert, „Amulet3: A high-performance self-timed arm microprocessor“, in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*, 1998, pp. 247–252. DOI: 10.1109/ICCD.1998.727058.

[22] I. E. Sutherland, „Micropipelines“, *Commun. ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989, ISSN: 0001-0782. DOI: 10.1145/63526.63532.

[23] A. Martin, A. Lines, R. Manohar, *et al.*, „The design of an asynchronous mips r3000 microprocessor“, in *Proceedings Seventeenth Conference on Advanced Research in VLSI*, 1997, pp. 164–181. DOI: 10.1109/ARVLSI.1997.634853.

[24] J. Teife and R. Manohar, „Programmable asynchronous pipeline arrays“, vol. 2778, Sep. 2003, pp. 345–354, ISBN: 978-3-540-40822-2. DOI: 10.1007/978-3-540-45234-8_34.

[25] C. IV, V. Ekanayake, and R. Manohar, „Snap: A sensor-network asynchronous processor“, Jun. 2003, pp. 24–33, ISBN: 0-7695-1898-2. DOI: 10.1109/ASYNC.2003.1199163.

[26] A. Martin, M. Nystroem, K. Papadantonakis, *et al.*, „The lutonium: A sub-nanojoule asynchronous 8051 microcontroller“, Jun. 2003, pp. 14–23, ISBN: 0-7695-1898-2. DOI: 10.1109/ASYNC.2003.1199162.

[27] A. Martin, M. Nystrom, and C. Wong, „Three generations of asynchronous microprocessors“, *IEEE Design & Test of Computers*, vol. 20, no. 6, pp. 9–17, 2003. DOI: 10.1109/MDT.2003.1246159.

[28] K. M. Fant, *Logically Determined Design*. Jan. 2005, ISBN: 9780471684787.

[29] E. Izhikevich, „Simple model of spiking neurons“, *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003. DOI: 10.1109/TNN.2003.820440.

[30] A. L. Hodgkin and A. F. Huxley, „A quantitative description of membrane current and its application to conduction and excitation in nerve“, *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, Aug. 1952. DOI: 10.1113/jphysiol.1952.sp004764.

[31] A. Martin, „Programming in vlsi: From communicating processes to delay-insensitive circuits“, *Developments in Concurrency and Communication*, Mar. 1991.

[32] A. Martin and M. Nystrom, „Asynchronous techniques for system-on-chip design“, *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1089–1120, 2006. DOI: 10.1109/JPROC.2006.875789.

[33] Z. Tabassam, S. R. Naqvi, T. Akram, M. Alhussein, K. Aurangzeb, and S. A. Haider, „Towards designing asynchronous microprocessors: From specification to tape-out“, *IEEE Access*, vol. 7, pp. 33 978–34 003, 2019. DOI: 10.1109/ACCESS.2019.2903126.

[34] D. Bhadra and K. S. Stevens, „Design of a low power, relative timing based asynchronous msp430 microprocessor“, in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 794–799. DOI: 10.23919/DATE.2017.7927097.

[35] S. Ataei, W. Hua, Y. Yang, *et al.*, „An open-source eda flow for asynchronous logic“, *IEEE Design and Test*, vol. 38, pp. 1–10, Jan. 2021. DOI: 10.1109/MDAT.2021.3051334.

[36] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, and N. England, „Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers“, Jan. 2000.

[37] K. Christensen, P. Jensen, P. Korger, and J. Sparso, „The design of an asynchronous tinyrisc/sup tm/ tr4101 microprocessor core“, in *Proceedings Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 108–119. DOI: 10.1109/ASYNC.1998.666498.

[38] V. Khomenko, D. Sokolov, A. Yakovlev, and D. Lloyd, „Handshake verification in workcraft“, in *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2020, pp. 63–64. DOI: 10.1109/ASYNC49171.2020.00016.

[39] D. Sokolov, V. Khomenko, A. Yakovlev, and D. Lloyd, „Design and verification of speed-independent circuits with arbitration in workcraft“, in *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2018, pp. 30–31. DOI: 10.1109/ASYNC.2018.00017.

[40] K. van Berkel, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. USA: Cambridge University Press, 1993, ISBN: 0521452546.

[41] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen, „A behavioral synthesis frontend to the haste/tide design flow“, in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, 2009, pp. 185–194. DOI: 10.1109/ASYNC.2009.10.

[42] A. Bink and R. York, „Arm996hs: The first licensable, clockless 32-bit processor core“, *IEEE Micro*, vol. 27, no. 2, pp. 58–68, 2007. DOI: 10.1109/MM.2007.28.

[43] A. Yakovlev, P. Vivet, and M. Renaudin, „Advances in asynchronous logic: From principles to gals & noc, recent industry applications, and commercial cad tools“, in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 1715–1724. DOI: 10.7873/DATE.2013.346.

[44] D. Edwards and A. Bardsley, „Balsa: An asynchronous hardware synthesis language“, *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002. DOI: 10.1093/comjnl/45.1.12.

[45] Q. Zhang and G. Theodoropoulos, „Modelling samips: A synthesisable asynchronous mips processor“, in *37th Annual Simulation Symposium, 2004. Proceedings.*, 2004, pp. 205–212. DOI: 10.1109/SIMSYM.2004.1299484.

[46] P. A. Beerel, G. D. Dimou, and A. M. Lines, „Proteus: An asic flow for ghz asynchronous designs“, *IEEE Design Test of Computers*, vol. 28, no. 5, pp. 36–51, 2011. DOI: 10.1109/MDT.2011.114.

[47] R. Diamant, R. Ginosar, and C. Sotiriou, „Asynchronous sub-threshold ultra-low power processor“, in *2015 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2015, pp. 89–96. DOI: 10.1109/PATMOS.2015.7347592.

[48] J. Sparsø, *Introduction to Asynchronous Circuit Design*, English. DTU Compute, Technical University of Denmark, 2020, Paperback edition available here: https://www.amazon.com/dp/B08BF2PFLN.

[49] S. M. Nowick and M. Singh, „Asynchronous design—part 1: Overview and recent advances“, *IEEE Design Test*, vol. 32, no. 3, pp. 5–18, 2015. DOI: 10.1109/MDAT.2015.2413759.

[50] ——, „Asynchronous design—part 2: Systems and methodologies“, *IEEE Design Test*, vol. 32, no. 3, pp. 19–28, 2015. DOI: 10.1109/MDAT.2015.2413757.

[51] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A Designer's Guide to Asynchronous VLSI*. Cambridge: Cambridge University Press, 2010, ISBN: 978-1-139-48528-9.

[52] I. David, R. Ginosar, and M. Yoeli, „An efficient implementation of boolean functions as self-timed circuits“, *IEEE Transactions on Computers*, vol. 41, no. 1, pp. 2–11, 1992. DOI: 10.1109/12.123377.

[53] A. Kondratyev and K. Lwin, „Design of asynchronous circuits by synchronous cad tools“, in *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, 2002, pp. 411–414. DOI: 10.1109/DAC.2002.1012660.

[54] A. J. Martin, „The limitations to delay-insensitivity in asynchronous circuits“, in *Beauty Is Our Business*, Springer New York, 1990, pp. 302–311. DOI: 10.1007/978-1-4612-4476-9_35. [Online]. Available: https://doi.org/10.1007/978-1-4612-4476-9_35.

[55] K. van Berkel, „Beware the isochronic fork“, *Integration*, vol. 13, no. 2, pp. 103–128, 1992, ISSN: 0167-9260. DOI: https://doi.org/10.1016/0167-9260(92)90001-F.

[56] F. Huemer and A. Steininger, „Advanced delay-insensitive 4-phase protocols“, in *2018 Austrochip Workshop on Microelectronics (Austrochip)*, 2018, pp. 50–55. DOI: 10.1109/Austrochip.2018.8520702.

[57] ——, „Partially systematic constant-weight codes for delay-insensitive communication“, in *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2018, pp. 17–25. DOI: 10.1109/ASYNC.2018.00014.

[58] ——, „Novel approaches for efficient delay-insensitive communication“, *Journal of Low Power Electronics and Applications*, vol. 9, Jun. 2019. DOI: 10.3390/jlpea9020016.

[59] ——, „Sorting network based full adders for qdi circuits“, in *2020 Austrochip Workshop on Microelectronics (Austrochip)*, 2020, pp. 21–28. DOI: 10.1109/Austrochip51129.2020.9232987.

[60] F. Huemer, R. Najvirt, and A. Steininger, „Identification and confinement of fault sensitivity windows in qdi logic“, in *2020 Austrochip Workshop on Microelectronics (Austrochip)*, 2020, pp. 29–36. DOI: 10.1109/Austrochip51129.2020.9232985.

[61] ——, „On sat-based model checking of speed-independent circuits“, in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2022, pp. 100–105. DOI: 10.1109/DDECS54261.2022.9770165.

[62] P. Behal, F. Huemer, R. Najvirt, A. Steininger, and Z. Tabassam, „Towards explaining the fault sensitivity of different qdi pipeline styles“, in *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2021, pp. 25–33. DOI: 10.1109/ASYNC48570.2021.00012.

[63] P. Behal, F. Huemer, R. Najvirt, and A. Steininger, „An automated setup for large-scale simulation-based fault-injection experiments on asynchronous digital circuits“, in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 541–548. DOI: 10.1109/DSD53832.2021.00087.

[64] F. Huemer and A. Steininger, „Timing domain crossing using muller pipelines“, in *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2020, pp. 44–53. DOI: 10.1109/ASYNC49171.2020.00014.

[65] M. Singh and S. M. Nowick, „Mousetrap: High-speed transition-signaling asynchronous pipelines“, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, 2007. DOI: 10.1109/TVLSI.2007.898732.

[66] R. Dashkin and R. Manohar, „General approach to asynchronous circuits simulation using synchronous fpgas“, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021. DOI: 10.1109/TCAD.2021.3131546.

[67] A. Martin, M. Nystroem, and C. Wong, „Design tools for integrated asynchronous electronic circuits“, p. 17, Jun. 2003.

[68] A. J. Martin, „Synthesis of asynchronous vlsi circuits“, USA, Tech. Rep., 1991.

[69] C. A. R. Hoare, „Communicating sequential processes“, *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. DOI: 10.1145/359576.359585.

[70] A. J. Martin and C. D. Moore, *Chp and chpsim: A language and simulator for fine-grain distributed computation*, 2011.

[71] D. Fang, „Profiling infrastructure for the performance evaluation of asynchronous systems“, Ph.D. dissertation, Cornell University, Aug. 2008.

[72] R. Manohar, „An open-source design flow for asynchronous circuits“, *Government Microcircuit Applications and Critical Technology Conference*, Mar. 2019.

[73] C. T. O. Otero, J. Tse, R. Karmazin, B. Hill, and R. Manohar, „Ulsnap: An ultra-low power event-driven microcontroller for sensor network nodes“, in *Fifteenth International Symposium on Quality Electronic Design*, 2014, pp. 667–674. DOI: 10.1109/ISQED.2014.6783391.

[74] R. Li, L. Berkley, Y. Yang, and R. Manohar, „Fluid: An asynchronous high-level synthesis tool for complex program structures“, in *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2021, pp. 1–8. DOI: 10.1109/ASYNC48570.2021.00009.

[75] Y. Yang, J. He, and R. Manohar, „Dali: A gridded cell placement flow“, in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[76] W. Hua, Y.-S. Lu, K. Pingali, and R. Manohar, „Cyclone: A static timing and power engine for asynchronous circuits“, in *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2020, pp. 11–19. DOI: 10.1109/ASYNC49171.2020.00010.

[77] M. Pan, Y. Xu, Y. Zhang, and C. Chu, „Fastroute: An efficient and high-quality global router“, *VLSI Design*, vol. 2012, Aug. 2012. DOI: 10.1155/2012/608362.

[78] J. He, U. Agarwal, Y. Yang, R. Manohar, and K. Pingali, „Sproute 2.0: A detailed-routability-driven deterministic parallel global router with soft capacity“, in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 586–591. DOI: 10.1109/ASP-DAC52403.2022.9712557.

[79] J. He, M. Burtscher, R. Manohar, and K. Pingali, „Sproute: A scalable parallel negotiation-based global router“, in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942105.

[80] S. Ataei, J. He, W. Hua, *et al.*, „Toward a digital flow for asynchronous vlsi systems“, in *2nd Workshop on Open-Source EDA Technology (WOSET)*, Westminster, CO, Nov. 2019.

[81] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, „Openram: An open-source memory compiler“, in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–6. DOI: 10.1145/2966986.2980098.

[82] S. Ataei and R. Manohar, „Amc: An asynchronous memory compiler“, in *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2019, pp. 1–8. DOI: 10.1109/ASYNC.2019.00009.

[83] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, „Ntuplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints“, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, pp. 1228–1240, Aug. 2008. DOI: 10.1109/TCAD.2008.923063.

[84] R. Karmazin, S. Longfield, C. T. O. Otero, and R. Manohar, „Timing driven placement for quasi delay-insensitive circuits“, in *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, 2015, pp. 45–52. DOI: 10.1109/ASYNC.2015.16.

[85] R. Karmazin, C. T. O. Otero, and R. Manohar, „Celltk: Automated layout for asynchronous circuits with nonstandard cells“, in *2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems*, 2013, pp. 58–66. DOI: 10.1109/ASYNC.2013.27.

[86] R. K. Dybvig, *The Scheme Programming Language*, 4th ed. The MIT Press, Jul. 2009, ISBN: 9780262512985.

[87] R. W. Hon and C. H. Séquin, *A Guide to LSI Implementation*, 2nd ed. Palo Alto, California: Xerox Palo Alto Research Center, 1980.

[88] C.-C. Chuang, Y.-H. Lai, and J.-H. R. Jiang, „Synthesis of pchb-wchb hybrid quasi-delay insensitive circuits“, in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

Online References

[89] R. Manohar. „Github - asyncvlsi/act: The act language and core tools“. (2021), [Online]. Available: https://github.com/asyncvlsi/act (visited on May 7, 2022).

[90] J. Bohn. „Ems - history - illiac i“. (), [Online]. Available: https://music.illinois.edu/ems-history-illiac-i (visited on Apr. 21, 2022).

[91] R. Manohar. „The act vlsi design tools - documentation“. (2021), [Online]. Available: https://avlsi.csl.yale.edu/act/doku.php (visited on Apr. 21, 2022).

[92] U. P. de Catalunya. „Petrify: A tool for synthesis of petri nets and asynchronous circuits“. (1999), [Online]. Available: https://www.cs.upc.edu/~jordicf/petrify/ (visited on Aug. 16, 2022).

[93] „Tam16: 16-bit microcontroller ip core“. (2008), [Online]. Available: http://www.tiempo-ic.com/uploads/Docs/TAM16_Datasheet.pdf (visited on Apr. 21, 2022).

[94] T. Wien. „Embedded computing systems group“. (), [Online]. Available: https://ti.tuwien.ac.at/ecs (visited on May 31, 2022).

[95] M. Research. „Github - z3prover/z3: Z3“. (2022), [Online]. Available: https://github.com/Z3Prover/z3 (visited on Jun. 16, 2022).

[96] R. Manohar. „Cast“. (1997), [Online]. Available: https://avlsi.csl.yale.edu/act/lib/exe/fetch.php?media=history:cast.pdf (visited on May 8, 2022).

[97] D. Fang. „Github - fangism/hackt: Hackt (hierarchical asynchronous circuit kompiler toolkit) a compiler suite for asynchronous system design“. (2018), [Online]. Available: https://github.com/fangism/hackt (visited on Apr. 21, 2022).

[98] J. He, Y. Yang, and R. Manohar. „Github - asyncvlsi/sproute: A scalable parallel global router“. (2022), [Online]. Available: https://github.com/asyncvlsi/SPRoute (visited on Apr. 21, 2022).

[99] S. Ataei and R. Manohar. „Github - asyncvlsi/amc: Asynchronous memory compiler“. (2020), [Online]. Available: https://github.com/asyncvlsi/AMC (visited on Apr. 21, 2022).

[100] U. of Texas at Austin. „Galois“. (2020), [Online]. Available: https://iss.oden.utexas.edu/?p=projects/galois (visited on May 9, 2022).

[101] R. Dashkin and R. Manohar. „Github - asyncvlsi/prs2fpga: Translate production rules into verilog for accelerated simulation on fpgas“. (2021), [Online]. Available: https://github.com/asyncvlsi/prs2fpga (visited on Apr. 21, 2022).