

DISSERTATION

Design of an Asynchronous Processor Based on Code Alternation Logic – Exploration of Delay Insensitivity

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften
unter der Leitung von

A.O.UNIV.-PROF. DIPL.-ING. DR. ANDREAS STEININGER

Inst.-Nr. E182/2

Institut für Technische Informatik
Embedded Computing Systems Group

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

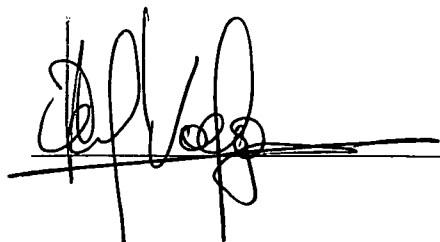
MAG. DIPL.-ING. WOLFGANG HUBER

Matr.-Nr. 9425221

Edla 9

3261 Steinakirchen/Forst

Wien, im Mai 2005



Acknowledgements

First, I would like to thank Andreas Steininger for his patience and excellent supervision on this thesis.

The positive working atmosphere at our institute made this work possible. Thanks to all of my colleagues, especially Martin Delvai for working together the last four years and Wilfried Elmenreich for the helpful suggestions, even while having lunch.

I also wish to thank my brother Bernhard and Susanne Pind-Rosnagl for faithful reading and correcting my work.

Many thanks to Traude Sommer, Gottfried Fuchs, Thomas Handl, and Peter Tumeltshammer for constructive comments and suggestions.

Last but not least, a big part on the successful completion of this work goes to Waltraud and Thomas for supporting me during writing this thesis.

Abstract

The synchronous design technique increasingly reaches its limits: More and more functionalities are integrated on one chip causing the chip size to grow, thus effectively countervailing the reduction of feature size allowed by improved manufacturing technologies. Higher clock frequency not only requires steeper clock edges, hence increasing power consumption, but also destroys the fiction of simultaneous events on the whole chip. This abstraction of simultaneity is an essential basis for how the synchronous design paradigm solves the fundamental design problems, e.g. the formal incompleteness of the Boolean Logic. Synchronous designs tackle these fundamental problems in the time domain. However, solutions in the information domain and the so-called hybrid solution – a conjunction of both methods – are also possible. Code Alternation Logic (CAL) is a representative of a hybrid solution. Up to a certain abstraction level, CAL is delay insensitive, however, for the implementation of the basic gates, temporal restrictions apply.

CAL is based on the utilization of two representations of "HIGH" and "LOW" in two different phases φ_0 and φ_1 . The representations are used alternatively, so within a sequence of data words each bit can uniquely be assigned to the corresponding data word. These four possible values are either described by a four-value single-rail signal of type `cal_logic`, which is utilized for the behavioral description of the design, or by a two-value dual-rail `cal_rail_logic` signal. The latter is used for the hardware implementation. In contrast to the conventional synchronous design flow, our CAL design flow comprises two synthesis steps. Furthermore, the designer is supported by simulation models for different abstraction levels. Altera Apex FPGAs are used as target technology, thus LUTs are the smallest units which can be addressed.

A main goal of this thesis is to analyze the delay insensitivity of a circuit implementation with CAL. For this purpose pipeline stages as well as basic gates are transformed to timed automata and analyzed with the model-checker Uppaal. In this thesis the delay insensitive behavior of pipeline structures and the correctness of the combinational logic between these stages is proven. Up to this point hardware independent models of the basic gates are used for constructing these combinational logic functions, which operate according to the CAL rules. As a next step the implementation of the basic gates in the target technology is investigated. The limitations with respect to delay insensitivity are pinpointed and appropriate design constraints are derived.

The impacts of the findings are used to improve the design flow. Furthermore, the results have allowed us designing our asynchronous processor ASPEAR. The development of an improved pipeline concept, the application of pre-compiled basic gates using Quartus, and the new library providing these gates to the synthesis tools are verified with the successful implementation of the processor.

Kurzfassung

Die synchrone Hardware-Designmethodik stößt zunehmend an ihre Grenzen. Immer mehr Funktionalitäten werden auf einen Chip gepackt, wodurch die Chipfläche trotz sinkender Transistorgrößen, die Dank besserer Herstellungstechnologien erreicht werden, steigt. Steigende Taktfrequenz erfordert nicht nur steilere Taktflanken, wodurch der Leistungsverbrauch negativ beeinflusst wird, sondern zerstört die Fiktion der Gleichzeitigkeit der Ereignisse am Chip. Diese Abstraktion der Gleichzeitigkeit ist aber die Grundlage für den Einsatz des synchronen Designparadigmas um die fundamentalen Designprobleme, wie zum Beispiel die formale Unvollständigkeit der Bool'schen Logik in den Griff zu bekommen. Der synchrone Ansatz löst das Problem der Gültigkeit von Information vollständig im Zeitbereich, weiters wären aber auch Lösungen im Informationsbereich, oder auch als Mischung beider Varianten, so genannte hybride Lösungen, möglich. Die beiden letztgenannten sind Vertreter der asynchronen Logik. Als Beispiel für einen hybriden Ansatz sei die in unserer Arbeitsgruppe entwickelte Code Alternation Logic (CAL) genannt. Bis zu einer gewissen Abstraktionsschicht ist CAL delay insensitive, die darunter liegende Implementierung der Basisgatter mit Zellen der Zieltechnologie ist nicht ganz frei von zeitlichen Bedingungen an die Implementierung.

CAL basiert auf der Tatsache, dass "HIGH" und "LOW" jeweils in zwei sich abwechselnden Phasen φ_0 und φ_1 dargestellt werden. Durch die Alternierung der Phasen kann die Zuordnung der Bits zu Datenwellen garantiert werden. Dargestellt werden diese vier Zustände entweder als vierwertige single-rails `cal_logic`, die in der Verhaltensbeschreibung der Designs verwendet werden, oder als jeweils zweiwertige dual-rail Logik – `cal_rail_logic` – für die Implementierung in Hardware. Der Designflow für CAL umfasst im Gegensatz zum synchronen Design zwei Syntheseschritte. Weiters wird die Designentwicklung durch Simulationsmodelle auf verschiedenen Abstraktionsstufen unterstützt. Als Zieltechnologie kommen Altera Apex FPGAs mit LUTs als kleinste Einheiten zum Einsatz.

Um die Eigenschaften in Bezug auf delay insensitivity zu untersuchen, werden Hardwarestrukturen wie Pipelinestufen oder auch Basisgatter in Timed Automata transferiert, die mit dem Modelchecker Uppaal analysiert werden. Der Nachweis des delay insensitiven Verhaltens der Pipelinestufen und der Logik zwischen diesen Stufen wird in dieser Arbeit erbracht. Dieser stützt sich auf hardwareunabhängige Modelle der Basisgatter, die nach den CAL-spezifischen Regeln operieren. Der nächste Schritt in dieser Arbeit ist die Untersuchung der Implementierung dieser Basisgatter in der Zieltechnologie. Der Nachweis ebenso wie die Ermittlung der notwendigen Vorgaben für die Implementierung werden mittels Modelchecker für alle im Designflow verwendeten Basisgatter erbracht.

Die gewonnen Erkenntnisse finden im überarbeiteten Designflow zur Implementierung des asynchronen Prozessors ASPEAR Anwendung. Der Funktionsnachweis der verbesserten Pipelines, der in Quartus vorcompilierten Gatter und der dadurch notwendige Einsatz einer neuen Bibliothek wird durch die erfolgreiche Implementierung des Prozessors erbracht.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Contribution and Objectives	4
1.3	Structure of the Thesis	5
2	State of the Art	6
2.1	System Model	8
2.1.1	Terminology	8
2.1.2	Data Flow	9
2.1.3	Timed Data Flow Relation	10
2.2	The Fundamental Design Problem	10
2.2.1	Formal Incompleteness of Boolean Logic	11
2.2.2	Signal Delay	12
2.2.3	Signal Skew	12
2.3	Strategic Options	13
2.3.1	Time Domain	13
2.3.2	Information Domain	15
2.3.3	Hybrid Solutions	19
2.4	Design Techniques	21
2.4.1	Synchronous Approach	21
2.4.2	Bundled-Data Approach	23
2.4.3	Huffman Approach	25
2.4.4	Design Techniques Using Signal Coding – The NCL Example	27
2.4.5	Transition Signaling Approach	29
2.4.6	Handshake Protocols: The Micropipeline Approach	30
2.4.7	High Level Description Approaches	32
2.5	Comparison	34
3	Code Alternation Logic – CAL	38
3.1	Background of CAL	39
3.2	Coding Scheme	40
3.3	Control Flow	42
3.4	Levels of Abstraction	43

3.4.1	Behavioral Description – cal_logic	43
3.4.2	Functional Description – cal_rail_logic	46
3.5	Basic Gates	47
3.5.1	AND Gate	47
3.5.2	Phase Detector	47
3.5.3	φ -Converter	48
3.5.4	CAL Register	49
3.6	CAL Design-Flow	50
3.7	Simulation Concept	52
3.8	Summary	54
4	Prototyping Environment	55
4.1	The SPEAR Processor	56
4.1.1	Core Architecture	56
4.1.2	Extension Modules	57
4.1.3	Implementation Results	58
4.2	The Hardware Platform	58
4.2.1	APEX FPGA Family	59
4.2.2	Limitations	60
5	Delay-Insensitivity of Circuits Built with CAL	62
5.1	The Uppaal Tool Suite	63
5.2	Delay Insensitivity Analysis of CAL-Registers and the Pipeline Structure	65
5.2.1	Schematic Pipeline	65
5.2.2	Pipeline Implementation	69
5.2.3	Pipeline Model with Synchronized Capture Done	73
5.2.4	Pipeline Implementation with Latched Capture Done	74
5.3	The Combinational Functions ($f(x)$)	78
5.3.1	Circuits built with CAL-Gates	78
5.4	Summary	81
6	Delay-Insensitivity of CAL Basic Gates	82
6.1	Modeling Altera FPGA Designs	83
6.2	The AND Gate	88
6.2.1	Synopsys-edif-Version	88
6.2.2	Quartus-only-Version	93
6.3	The OR Gate	95
6.4	The INV Gate	97
6.5	N-Signal φ -Detector	98
6.5.1	Two Signal φ -Detector	98
6.5.2	Four Signal φ -Detector	100
6.5.3	Generic N-Rail φ -Detector	101
6.6	Latch	102

6.6.1	One Signal wide Latch	102
6.6.2	Latch with Enable Logic	104
6.7	Summary	106
7	Hardware Implementation: Asynchronous SPEAR	107
7.1	General Description	108
7.1.1	Design Migration Issues	108
7.2	Pipeline Improvements	109
7.2.1	Providing Latches with Different Initialization Values	110
7.2.2	"Capture Done Latches" in the Feedback Path	111
7.3	Adapting the Design-Flow	112
7.3.1	Pre-compiled Quartus Gates	113
7.3.2	Additional Target Library CALRAILLIB	114
7.3.3	Simulation Support	114
7.4	Implementation Results	115
8	Conclusion	117
	Bibliography	122

List of Figures

1.1	Gate vs. Interconnect Delay [107]	3
2.1	Terminology	9
2.2	Circuit Model	9
2.3	Timed Circuit Model	11
2.4	Fundamental Design Problem	12
2.5	Transition between Consistent Data Words	13
2.6	Fundamental Solutions in the Time Domain	14
2.7	Validity vs. Consistency	17
2.8	Communication Process	18
2.9	Communication Protocols	19
2.10	Circuit Fragment with Gates and Delays	19
2.11	Synchronous Design Approach	21
2.12	Bundled-Data Design Approach	24
2.13	Huffman Circuit [85]	25
2.14	Sequence of DATA and NULL Waves	27
2.15	Micropipeline	31
3.1	Flow of Data Waves in CAL	40
3.2	Possible Phase Transition	41
3.3	CAL Pipeline Structure	42
3.4	Library Dependencies	43
3.5	Schematic and Truth Table of the AND-gate	48
3.6	The φ -Detector	48
3.7	Implementation of a φ -Converter	49
3.8	Implementation of a CAL Register	49
3.9	CAL-Design Flow	51
3.10	Simulation Concept	53
3.11	Post-layout Simulation Example	54
4.1	SPEAR Architecture	57
4.2	Generic Extension Module Interface	58
4.3	Logic Element Structure [2]	60

5.1	Uppaal Template Example: P	63
5.2	Uppaal Template Example: Two Instances of P: p1 and p2	65
5.3	Schematic CAL Pipeline Structure	65
5.4	Schematic Pipeline	66
5.5	Completing the Schematic Pipeline System	67
5.6	Uppaal Simulator (Schematic Pipeline System)	68
5.7	CAL Pipeline Structure	70
5.8	Components of Pipeline Implementation	71
5.9	Pipeline Implementation	71
5.10	Critical Delay Paths of the Basic Pipeline Implementation	73
5.11	Pipeline Model with Synchronized Capture Done	74
5.12	CAL Register with latched Capture Done	75
5.13	CAL Register with latched Capture Done signal	77
5.14	Uppaal-Model of an AND-Gate	78
5.15	Uppaal-Models	79
5.16	Input Generation Function proving the $f(x)$ Model	80
6.1	Altera Apex LUT Model	84
6.2	Altera Apex Wire Model	86
6.3	Feedback Model	86
6.4	Stimuli-Generation	87
6.5	AND-LUT Schematic generated by Synopsys	88
6.6	Simulation of the AND2-model	91
6.7	AND-LUT Schematic	93
6.8	OR2-LUT EDIF-Schematic	95
6.9	Symmetric OR-Gate	96
6.10	INV-LUT Schematic	97
6.11	Two Signal φ -Detector LUT Schematic	98
6.12	Four Signal φ -Detector LUT Schematic	100
6.13	Two Signal wide Instance of a Generic φ -Detector	101
6.14	Four Signal wide Instance of a Generic φ -Detector	102
6.15	One Signal wide Latch Cell	102
6.16	Stimuli Generation for an one Signal wide Latch Cell	103
6.17	One Signal wide Latch Cell with Enable Logic	104
6.18	Stimuli Generation for a Latch Cell with Enable Logic	105
7.1	Implementation Result of a Four Signal wide Register	112
7.2	Improved CAL Design Flow	113
7.3	Implementation Results of ASPEAR	115
8.1	CAL-Register Delay Summary	119

List of Tables

2.1	Comparison wrt. the Fundamental Design Problem	34
2.2	Comparison wrt. Area and Energy Efficiency	36
3.1	CAL Coding Scheme	41
3.2	cal_logic Coding Scheme	44
3.3	cal_rail_logic Coding Scheme and the VHDL Definition	46
3.4	Truth Table of a 2-input AND in CAL	47
5.1	Results of Uppaal checking the Schematic Pipeline	69
5.2	Results of Uppaal checking the Real Pipeline	72
5.3	Results of Uppaal checking the Pipeline Model with Synchronization	74
5.4	Results of Uppaal checking the "Latched Capture Done" Version	77
5.5	Results of Uppaal checking the $f(x)$ simulation	81
6.1	Gate Delay and the AND-gate without Feedback	90
6.2	LUT Simulation with Feedback (AND-gate)	92
6.3	Gate Delays of an AND-gate	94
6.4	LUT Simulation with Feedback (AND-gate)	94
6.5	Gate Delays of an OR-Gate	96
6.6	Gate Delays of an OR-Gate (full-Quartus) with Feedback	97
6.7	Gate Delays of an Inverter	97
6.8	Gate Delays of a Four Rail φ -Detector	99
6.9	Gate Delays of a Four Rail φ -Detector with Feedback	99
6.10	Gate Delays of a Eight Rail φ -Detector	100
6.11	Gate Delays of a Eight Rail φ -Detector with Feedback	101
6.12	Gate Delays of a 4-Signal Instance of a Generic φ -Detector with Feedback	102
6.13	Gate Delays of one Signal wide Latch with Feedback	104
6.14	Latch with Enable Cell and Feedback	105

List of Sources

3.4.1 <code>cal_logic</code> VHDL Definition	44
3.4.2 Register Implementation in <code>std_logic</code> and CAL	45
5.2.1 Uppaal-Query for the Schematic Pipeline	66
5.2.2 Definition of the Schematic Pipeline	67
5.2.3 Uppaal-Query for the First real Pipeline Implementation	70
6.1.1 Example of a Quartus Equation-File	83
6.2.1 Uppaal-Queries	89
6.6.1 Uppaal-Query for the Latch Models	103
7.2.1 Register Implementation in CAL	109
7.2.2 Register Cell Implementation in CAL	110
7.2.3 Capture Done Latch Implementation	111
7.3.1 Library Definition of an AND-Gate	114
7.3.2 Result of Synopsys Library Report: CALRAILLIB-Members	115

Chapter 1

Introduction

Asynchronous logic design has been a topic in the scientific community for a long time. Asynchronous design methods date back to the 1950s and two scientists mainly investigated by David A. Huffman [51] and David E. Muller [84]. Nevertheless, clocked circuits dominate the market of digital circuits today, while a small segment is only reserved for asynchronous chips [117]. The triumphal procession of the synchronous approach is based on its discretization of time: This facilitates the description of the circuit behavior – the designer hypothesizes that all operations within the circuit finish in time in order to be sampled with the next clock edge. Hence, neither glitches, signal delay, skew nor physical properties, such as driver power or the real duration of a logical operation have to be considered during functional description. This circumstance yielded to shorter design cycles and paved the way for *Hardware Description Languages* (HDL's) such as Verilog [121] and VHDL [62], boosting the productivity of chip designers again. A big advantage of synchronous designs is that the order in which the data arrives doesn't matter. The data can arrive at different times, but the registers delay capturing until the next active clock. As long as all signals arrive before this tick, the design will work properly. So the designer must not worry about wire delays while designing a chip.

In addition, design verification is reduced to checking the delays in the combinational logic functions between the registers [42], which can be automated.

The synchronous design paradigm in conjunction with high level hardware description languages, elaborated tools, and technological advances concerning integration density has enabled great strides to be taken into the design and performance of computers. In 1965 Gordon Moore predicted that chip density (and performance) doubles every eighteen months [82]. *“In 24 years the number of transistors on processor chips has increased by a factor of almost 2400, from 2300 on the Intel 4004 in 1971 to 5.5 million on the Pentium Pro in 1995 (doubling roughly every two years)”* [27]. Moore's observation remains valid until today (2005) and at the International Solid-States Circuits Conference (ISSCC 2003), Moore has predicted that this trend will proceed in the next decade [81]. Nowadays, processor cores clocked with several GHz and built out of more than 400 millions transistors [52][26] are typically for use in personal computers.

However, during the last decade there has been a revival in research on asynchronous circuits [95][42]. Clockless chips have long been a subject of research at facilities such as the California Institute of Technology's Asynchronous VLSI Group and the Manchester's Amulet project [129].[43]

1.1 Motivation

Clocked processors have dominated the industry since the 1960s. The question is now, what is the motivation behind moving away from a well established and approved design methodology? With the improvements made in the last decades several already existing problems concerning the chip design style became increasingly critical and will be further aggravated by each new technology step. Many problems can be ascribed to the limitations of the speed of the light¹: As soon as the signal propagation delay becomes a significant part of the clock period – clock frequencies in a range of several GHz imply clock periods under one nanosecond – circuit designers have to pay a heavy price to keep up the illusion, so that all components receive the rising edge of the clock signal at the same time [106].

Another critical issue is concerned with power consumption [18]. The clock signal always triggers the components, regardless of whether they have to do useful work or not – increase unnecessarily the energy consumption. The circumstance that the gates start functioning with the same clock edge leads to a peak in the power distribution. Furthermore, the miniaturization aggravates this situation by escalating the heat density inside high performance chips.

In addition, the combination of chips with higher functionality and faster transistors caused a fundamental change in the relation between gate and wire delay: In today's sub-micron designs, wire delays and not gate delays are the dominant factors for circuit timing (see figure 1.1).

Furthermore, a reliable verification of a circuit can be performed after place & route only, and so it is performed at a very late point in time in the design process. In practice, however, timing problems often necessitate changes in the functional design. In this way the separation of functional design and timing analysis causes unnecessary long iteration cycles.

The asynchronous approach seems to solve most of the problems in a natural way: Being event-driven, asynchronous circuits (i) perform operations only when required and thus reduce the power consumption, (ii) do not require a global time reference, when disarming the problems concerning clock distribution and signal skew.

As a fetch-ahead to the following chapters, important properties of asynchronous circuits, which can be advantages in some areas [108], are listed in the following:

¹To be more precise electrical signals travel on chips with 2/3 of the speed of light.

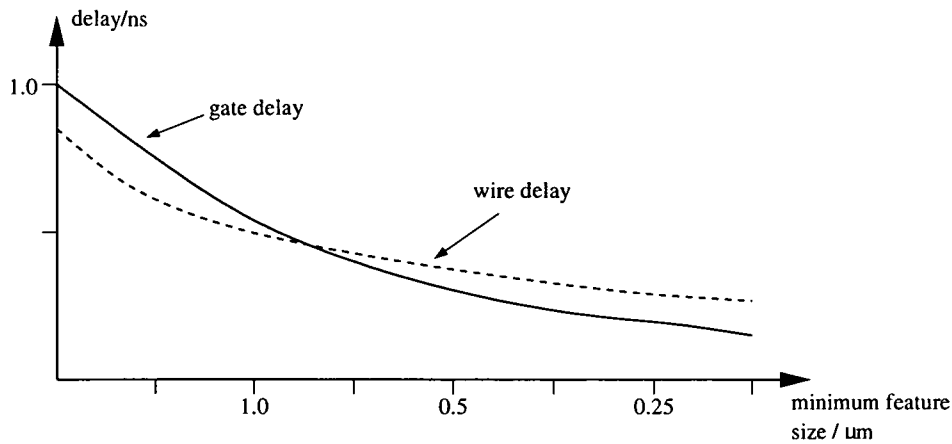


Figure 1.1: Gate vs. Interconnect Delay [107]

- Achievement average case performance, [74][73][127][128]
...operating speed is determined by actual local delays rather than the global worst-case latency.
- Low power consumption [40][39][89][11][10]
...consume power only when needed.
- Provision of easy modular composition [83][68][111][8]
...asynchronous components are combined with simple handshake protocols.
- Avoid clock distribution and clock skew problems
...because there is no global clock.
- Lower electro magnetic emission and noise [78][93][10]
...locally triggered registers tend to be active at any point in time.
- Variations in fabrication process parameters, temperature, and supply voltage are not as crucial as in synchronous designs [88][87][72]
...because the timing is based on the relationship between the delays instead of being based on absolute values.

Convinced by the potential of the asynchronous design style our department started its research activity in this field four years ago. The aim was not to invent a new method, but to provide an in-deep analysis of one existing design style. We have chosen the four phase logic approach [77][20], due to the fact that it allows us building completely delay-insensitive circuits on gate level and it does not require a neutral state between valid data words such as the Null Convention Logic [34]. On account of the four phase logic alternates the data encoding style within consecutive data words. We termed this logic CAL (Code Alternation Logic).

To perform our analysis, we have first developed a reference object, namely a synchronous processor core called *SPEAR* (Scalable Processor for Embedded Applications in

Real-time Environments). In the second step we re-designed it using the CAL approach. This not only opened the way to perform conventional analyses but it also allows comparing the asynchronous processor with the synchronous one concerning, e.g., speed, fault-tolerance, testability, etc.

1.2 Contribution and Objectives

Encoding data with alternating phases is the basic principle of CAL. Thus, all components in a CAL circuit are able to judge consistency of their input data by themselves. In addition, it is possible to decide whether a new output has to be generated or not. Therefore, we claim that CAL is delay insensitive on a high abstraction level. As mentioned before, wire delays become more and more important, so a delay insensitive approach for building circuits is particularly interesting.

However, the class of circuits that are entirely delay insensitive is limited: Circuits in this class may contain only Muller-C elements, as multiple-input gates with single output. [45, 70].

The hardware implementation of CAL uses dual-rail gates for constructing the circuits. Thus, every basic CAL gate used in these circuits has a dual output, which circumvents the above restrictions and allows the use of basic functions such as AND and OR. This is, however, a macroscopic view. On a microscopic level these basic CAL-gates are composed of standard gates from the target technology, for which the restrictions do apply. This leads to two main aspects to be considered in this thesis:

- I The delay insensitivity of CAL on a high level using CAL gates to build the circuits is to be examined. Pipeline structures with the corresponding handshake signals shall be investigated and their delay insensitivity property shall be proven. In this case limitations shall be identified as well. Furthermore, the behavior of combinational logic functions, which are constructed exclusively from CALgates shall be explored.
- II The internal structure of the basic gates has to be analyzed. The limitations mentioned above have to be pinpointed and also appropriate design rules have to be derived. The aim is to extract the bounds which must be abided by the implementation of these basic gates such that they behave as expected on the higher level.

A design flow has to be developed to facilitate a designer to use the potential of CAL. The flow from the high level of description to the target technology has to be generated. Furthermore, basic gates for the target library must be constructed that fulfill the constraints determined for the implementation.

Finally, the impacts which evolve of these findings as well as the functionality of the design flow and the basic gates shall be demonstrated by implementing an asynchronous processor core with CAL.

1.3 Structure of the Thesis

This short introduction is followed by Chapter 2, which identifies the fundamental problem designing digital circuits. The validity and consistency of data is considered as well as the possibilities in which the fundamental problem can be tackled. These two methods, namely the time domain and the information domain, are presented and the existing asynchronous design methods are classified with regard to the method applied. The section concludes with a comparison of presented methods. Chapter 3 introduces the design methodology *Code Alternation Logic* (CAL). The description of the applied coding scheme is followed by the description of the control flow. The construction of pipeline structures with the combinational function between two stages is presented, as well as the basic gates that are used to construct these function blocks. This is followed by an introduction to the CAL design flow and a comparison with the standard design flow. This chapter is rounded off with the description of the simulation concept. Our prototyping technology and the reference design are the focus of Chapter 4: The Altera Apex FPGA family is presented as prototyping hardware. Furthermore, the synchronous processor SPEAR and its implementation details are presented. The examination of delay insensitivity starts in Chapter 5: We have decided in favor of a top-down approach in order to analyze the behavior of CAL. In this chapter the delay insensitive behavior of circuits built with CAL gates is examined. Thus, pipeline structures and the handshake signals as well as the combinational functions between the stages are considered. The gates forming the basis for building the combinational functions are our focal point in Chapter 6. The principle of modeling Altera FPGAs is the starting point, afterwards the behavior of several basic gates is investigated. The limitations to delay insensitivity and the resulting restrictions are defined exactly for every basic gate. The impact of these findings yields to a modification to the design flow, as illustrated in Chapter 7. Furthermore, the implementation of the asynchronous processor ASPEAR is described. A conclusion of this work is given in Chapter 8.

CAL has been developed by Professor Andreas Steininger and his PhD students Martin Delvai and myself. Furthermore, the hardware realization of ASPEAR has been performed in cooperation by Martin and myself. Due to these facts, there are common parts in both theses. A. Steininger suggested to split up these common parts to enable his students to invest more time in the work itself instead of finding new verbalization for the same content. Thus, Chapter 2 "State of the Art" has been written by Martin Delvai and is part of my thesis too. My parts of the collaborate work – Chapter 3 "Code Alternation Logic" and the small Chapter 4 "Prototyping Environment" – are also parts of Martin's thesis[22].

Chapter 2

State of the Art

Martin Delvai¹

Contents

2.1	System Model	8
2.1.1	Terminology	8
2.1.2	Data Flow	9
2.1.3	Timed Data Flow Relation	10
2.2	The Fundamental Design Problem	10
2.2.1	Formal Incompleteness of Boolean Logic	11
2.2.2	Signal Delay	12
2.2.3	Signal Skew	12
2.3	Strategic Options	13
2.3.1	Time Domain	13
2.3.2	Information Domain	15
2.3.3	Hybrid Solutions	19
2.4	Design Techniques	21
2.4.1	Synchronous Approach	21
2.4.2	Bundled-Data Approach	23
2.4.3	Huffman Approach	25
2.4.4	Design Techniques Using Signal Coding – The NCL Example	27
2.4.5	Transition Signaling Approach	29
2.4.6	Handshake Protocols: The Micropipeline Approach	30
2.4.7	High Level Description Approaches	32
2.5	Comparison	34

¹As outlined in Section 1.3, this is the part of the common work written by Martin Delvai.

Circuit design styles can be classified into two major categories, namely synchronous and asynchronous. The first approach is based on one or more globally distributed periodic timing signals, called clocks, which sequence the circuit [19]. The asynchronous design style is an event-driven circuit design technique where, instead of the components sharing a common clock and exchanging data on clock edges, data is passed on as soon as it is available [35]. Although the asynchronous design methods have been studied for many decades, today the clocked circuits dominate the market of digital circuits. However, the synchronous design style faces some fundamental limits: Propagation of electrical signals on chips is bounded by the speed of light: As the chips get bigger and the clocks run faster, this physical restriction becomes more and more a crucial factor in the design process of synchronous chips [75]. Another critical aspect constitutes the power consumption: In CMOS circuits the dissipated energy is proportional to the switch activity – in synchronous circuits the gate activity is driven by clock signals, independent from the fact if useful work has to be done or not. Possible options to solve these problems are *clock gating* [63], where unused parts of the circuit are temporally disabled, or to *speed down the clock frequency* during idle states [53][25]. However, these are compromise solutions which aim to compensate the weak points of the synchronous design principle and which have to be paid in terms of recovery time and circuit overhead.

Being an event-driven method, the asynchronous design style promises to solve the mentioned difficulties by its nature. Motivated by this circumstance a lot of asynchronous design techniques have been developed [45]. Though all approaches have the same underlying principle, namely being event driven, their concrete effectuations look completely different.

Furthermore, many approaches deal with only one particular design aspect. Hence, implementing a complete chip often requires a combination of methods. This makes it difficult to classify asynchronous circuits and to compare them with the synchronous design style.

For completeness we will mention a third design style, namely the *Globally-Asynchronous Locally-Synchronous* (GALS) approach. The fundamental idea of this approach is applied successfully to compose systems on higher abstraction levels, connect a printer with a PC, e.g. As more and more components can be integrated on a single silicon die, this method becomes attractive even for VLSI designs [28][59]. However, this design style can be traced back to the previously mentioned styles and therefore it will not be considered separately.

All methods and design styles have one point in common: If we take a look from a more abstract point of view, we could recognize that all methods, including the synchronous approach, aim to solve the same problem, namely to ensure that all data is correctly processed by the circuit. We call this problem the *fundamental design problem*.

To be able to depict this problem in greater detail, we will first provide a system model in Section 2.1. Based on this definition we will figure out the fundamental design

problem and deduce its root in Section 2.2. In Section 2.3 we will distinguish between two basic strategies, which deal with the fundamental design, namely the use of time or the use of information. With this theoretical background we are able to analyze and classify characteristic types of design approaches in Section 2.4. This chapter concludes with a comparison of the presented design styles.

2.1 System Model

2.1.1 Terminology

Terms such as *signal*, *vectors*, *bits*, *e.g.*, are used in many different fields of applications. As a consequence these terms are interpreted in a slightly different manner depending on its context. Due to this common usage a discussion inside our department flared up about the exact meaning and interpretation of several expressions. Also the literature could not help to clarify the situation due to the fact that some terms are defined differently. It is for this reason that we devote a section to define the used *terminology*.

We call the input of a Boolean logic function an *input vector*. It is constituted by a number of *signals* – one for each input. The Boolean logic function defines a specific mapping from the input vector to an output signal. This mapping is implemented by a *logic function unit*. Often several Boolean logic functions are applied to the same input vector in parallel, creating several output signals with a common semantic context (data path elements like adder, *e.g.*). The term logic function units is used in a broader sense to describe the implementation of this set of Boolean functions as well.

We call the smallest unit of information conveyed on a signal a *bit*, and the (consistent) vector of bits conveyed on an input vector a *data word*. A signal can be physically represented by one or more *rails*, whose *logic levels* define the signal's *logic state*. The two mandatory logic states of a signal are "high (HI)" and "low (LO)", but states such as "NULL", "illegal" or "in transition" are conceivable and sometimes used as well. A *signal-level code* relates the logic levels of the rails – viewed as a vector that represents a signal – to the logic state of the corresponding signal. For the digital rails we consider that the logic level may either be "0" or "1". In the conventional single-rail encoding a signal is represented by only one rail whose logic level is directly mapped to the signal state.

We refer to an input vector as *consistent* at instant t_i , if the state of all its signals belong to the same context at instant t_i , *i.e.* if they represent one single valid data word, and inconsistent otherwise. We also call the involved signals consistent under this condition. We call a signal *valid* at instant t_i , if its state at instant t_i is the stable result of a logic operation performed on a consistent input vector, and invalid otherwise.

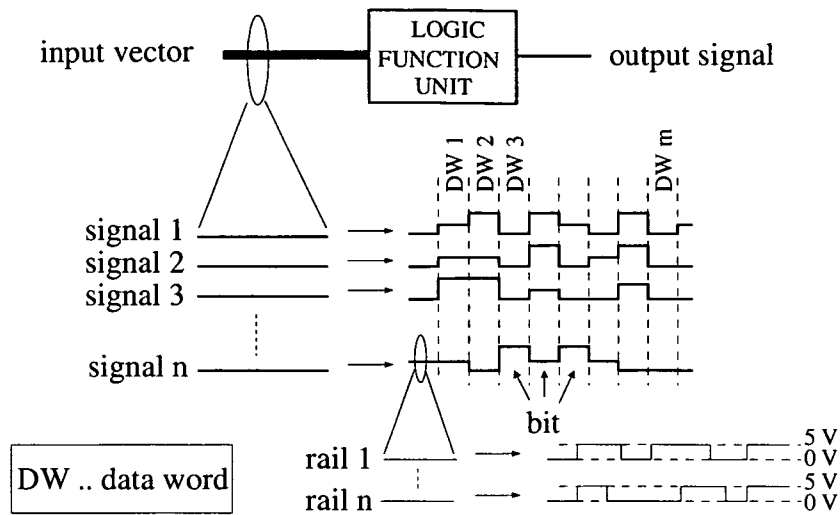


Figure 2.1: Terminology

2.1.2 Data Flow

From the point of view of information flow every function unit FU is preceded by a data source SRC that provides the input vector for FU , and followed by a data sink SNK that further processes the output signal or vector of FU (maybe in context with the outputs of other function units). Both, data sink SNK and source SRC represent an abstraction of the remaining circuit and may internally consist of further function units. We call an output bit b_y of FU consumed by the sink SNK at t_i , if b_y is still properly considered in the flow of information in SNK , regardless of whether b_y is overwritten by a subsequent bit b_z after t_i or not. Usually, consumption implies the transfer of the information to some storage element.

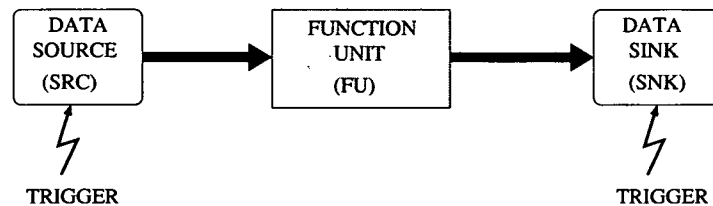


Figure 2.2: Circuit Model

An information flow is termed *lossless*, if all pertaining bits are properly consumed at all instances t_i . Also a signal path P_k is called lossless, if the information flow along P_k is lossless. To guarantee *losslessness*, SRC and SNK have to be appropriately coordinated.

2.1.3 Timed Data Flow Relation

Considering the temporal relations and delays involved in the data transfer between SRC and SNK , we have to extend our model by timing issues. Figure 2.3 illustrates this model. Thus, a source trigger $TRG_{SRC,x}$ is employed to determine the instant $t_{issue,x}$ for a data word $DW_{SRC,x}$ to be issued by the data source SRC ². As soon as SRC is ready to accept a trigger, it will react by issuing the requested data word $DW_{SRC,x}$, which will – after some delay – become visible and consistent at the output of SRC at instant $t_{issue,x}$. The interval between trigger event ($t_{TRG,SRC,x}$) and actual visibility of the consistent data word $DW_{SRC,x}$ at the output ($t_{issue,x}$) is named the *issue delay* Δ_{issue} . Next, $DW_{SRC,x}$ propagates to the function unit FU where it is processed. The corresponding result, $DW_{FU,x}$, propagates from the output of FU to the data sink SNK , passing SNK 's input logic, until it is finally available as a consistent data word $DW_{SNK,x}$ within the sink and hence ready for consumption at instant $t_{SNKrdy,x}$. The interval between $t_{issue,x}$ and $t_{SNKrdy,x}$ is termed as *processing delay* $\Delta_{process}$. At some point in time $t_{TRG,SNK,x} > t_{SNKrdy,x}$ the sink trigger $TRG_{SNK,x}$ is activated, which will – after some inherent delay – cause $DW_{SNK,x}$ to be actually consumed at instant $t_{consume,x}$. We call the interval between $t_{SNKrdy,x}$ and $t_{consume,x}$ the *consumption delay* $\Delta_{consume}$ and the interval between $t_{SNKtrg,x}$ and $t_{consume,x}$ the *sink trigger delay* $\Delta_{SNKtrig}$.

At instant $t_{issue,x+1} > t_{consume,x}$ it is safe to trigger the next data word $DW_{SRC,x+1}$ to be issued by the source. We call the delay until this actually occurs (i.e. the interval between $t_{consume,x}$ and $t_{issue,x+1}$) the *cycle delay* Δ_{cycle} . Notice that $DW_{SNK,x}$ does not necessarily become invalid immediately at $t_{issue,x+1}$ but only after $DW_{SRC,x+1}$ has propagated through FU to SNK . We describe this conservation of the previous data word by an *invalidity delay* $\Delta_{invalid}$. Consequently, the system designers have the opportunity to choose a negative Δ_{cycle} thus increasing throughput by issuing the next data word $DW_{SRC,x+1}$ already before the current data word $DW_{SNK,x}$ has been consumed. Note that all delays may vary and hence some margins have to be considered in the timing.

2.2 The Fundamental Design Problem

Based on the aforementioned definitions, the fundamental problem of digital logic design can be subsumed as follows: *Ensure a lossless information flow in the system.* Under this fundamental constraint systems are typically optimized for maximum information throughput. In order to achieve these aims the designer has to coordinate the triggers of source and sink appropriately. In context with the timed data flow model presented above, this implies the following:

- The *trigger of the sink* $TRG_{SNK,x}$ must not be activated before $t_{SNK,rdy,x}$ (ensure

²As we will see later, this trigger is the essential means for controlling the data flow in the signal path.

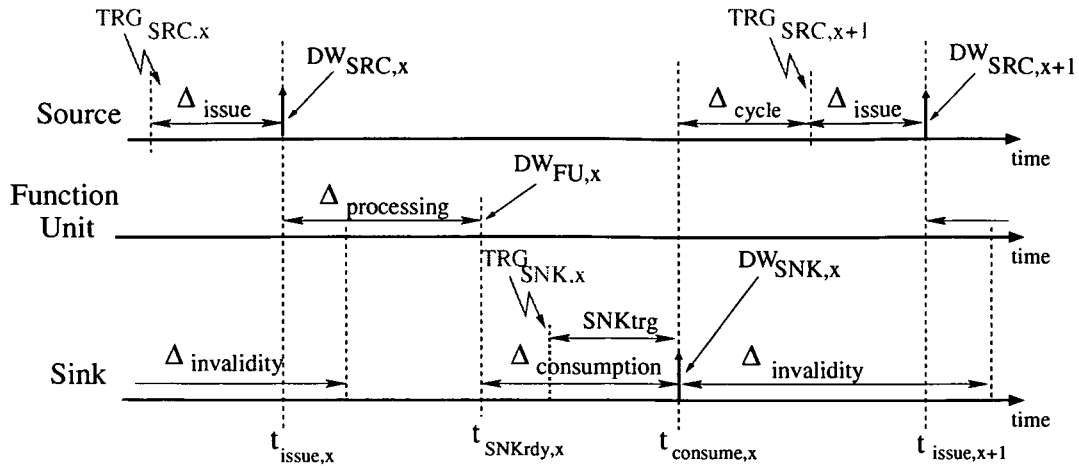


Figure 2.3: Timed Circuit Model

losslessness): $t_{SNK,trg,x} \geq t_{SNK,rdy,x}$. Less formally speaking this means that a new data word may only be captured by the sink after it has become consistent. To achieve maximum throughput capturing should, however, occur as soon after $t_{SNK,rdy,x}$ as possible. As a consequence, every design method must allow a judgement of consistency of a data word in one way or the other (**fundamental requirement 1**). Considering that validity is a prerequisite for consistency, it must be possible to judge on a signal's validity as well.

- The *trigger of the source* $TRG_{SRC,x}$ can safely be activated after $t_{consume,x}$ to guarantee losslessness, which means that the next data word may be issued only after the previous has been consumed: $t_{issue,x+1} \geq t_{consume,x}$. For maximum throughput it is desirable to place the trigger right after $t_{consume,x}$ or even prior to this instant (negative cycle delay). With respect to the design method this requires the existence of some kind of information feedback from the sink to the source (**fundamental requirement 2**).

Figure 2.4 illustrates these requirements. In practice requirement 2 has turned out to be relatively easily fulfilled by an appropriate circuit structure (micropipeline, e.g.), while the assessment of validity and consistency (fundamental requirement 1) is a notorious problem, which we will analyze more closely in the following sections.

2.2.1 Formal Incompleteness of Boolean Logic

Boolean logic defines functions on a high abstraction level. In essence, a Boolean function is a time-free mapping (truth table, e.g.) from the signals that form the input vector to an output signal. The output is reacting continuously to any change of the input word – there is no such thing as a trigger. This further implies that only consistent data words are applied to the logic function. In other words Boolean logic

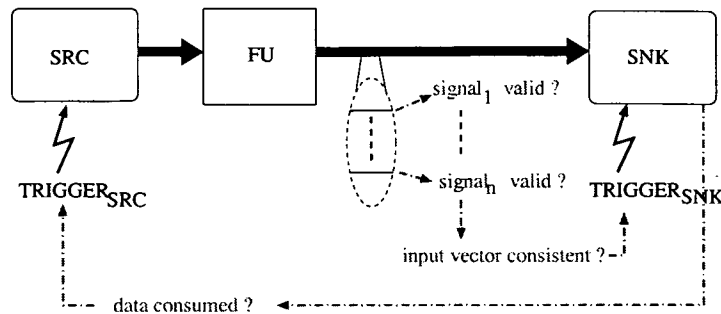


Figure 2.4: Fundamental Design Problem

does not provide any means for expressing or considering validity or consistency. Due to this fact Boolean logic is called "formally incomplete" in [34]. In fact there is even no way of expressing temporal relationships within the framework of Boolean logic – it is postulated that the input vectors are always consistent and the generated output is free of glitches. Unfortunately, due to signal delay and signal skew, none of these assumptions are fulfilled in a physical circuit implementation.

In conclusion, Boolean logic does not solve any of the fundamental requirements and so it does not contribute to solving the fundamental design problem in the first place. Still, Boolean logic is the established way of describing logic operations. All design methods have to compensate for this shortcoming in one way or another. In Section 2.3 we will analyze how different design styles solve this problem. However, before this action is performed we will analyze the roots of the problem in greater detail.

2.2.2 Signal Delay

Two constituents of signal delay are commonly distinguished, namely gate delay and interconnect delay. While gate delay is mainly determined by technology and fan-out, interconnect delay depends on many parameters that are specific to a given signal path like drive strength of the sender's output, capacitance and resistance of potential switch elements or vias along the wire, length and physical arrangement of the particular wire, and capacitance of the connected inputs. In addition, overall signal delay is a function of the operating conditions (supply voltage, temperature). As a consequence, the time it takes an output to become valid is non-zero, which is contradictory to the assumptions made by the Boolean logic.

2.2.3 Signal Skew

Due to the uncertainties with respect to signal delays no pair of signals will exhibit exactly the same delay. The difference of delays within signals of the same input vector is called *skew*. Notice that by definition skew distorts the temporal relations between signal transitions. As a result, the assumption that the transition from one data word

to the next one will occur at once (as implied by the continuous, untriggered definition of a Boolean logic function) is unrealistic. The edges on the individual rails will rather arrive sequentially, causing inconsistent intermediate signals and input vectors that (temporarily) result in invalid outputs. In this sense the skew disproves the validity and consistency assumption made by the Boolean logic. Figure 2.5 illustrates this effect.

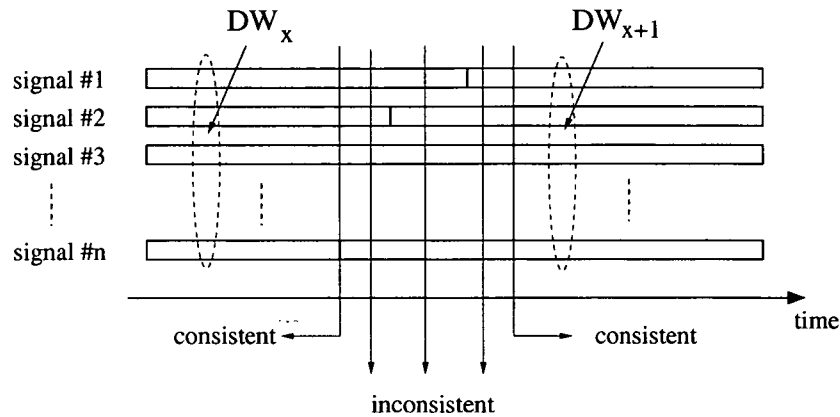


Figure 2.5: Transition between Consistent Data Words

As mentioned in Section 2.1 a signal may be represented by more than one rail. It is clear that in this case skew between the rails additionally compromises signal validity.

2.3 Strategic Options

In Section 2.2 we pointed out that it is an essential task of every digital design method to ensure that only consistent and valid data is consumed by the data sink and that the source is synchronized to the sink in such a way that no data gets lost. In this section we will identify two basic domains where this can be performed. Remember, that it is not required to solve all aspects of the fundamental design problem in one domain – mixed solutions are also possible.

2.3.1 Time Domain

Having figured out *timing* issues – namely delay and skew – as one root of the fundamental design problem, one consequent solution is to compensate for their undesired effects directly in the time domain.

Concerning the validity and consistency requirement, we can simply determine all relevant delays between source trigger $T_{SRC,x}$ at instant $t_{issue,x}$ and $t_{SNKrdy,x}$, the point in time when the data word is known to be ready for being captured at the sink. The sum of these delays constitutes the minimum time we have to wait after the source

trigger until we can apply safely the sink trigger:

$$t_{consume,x} \geq t_{issue,x} + \Delta_{issue} + \Delta_{process} \quad (2.1)$$

The determination of Δ_{issue} and $\Delta_{process}$ involves a careful analysis of the (implementation-dependent) delays. In the same way we can relate the source trigger to the sink trigger:

$$t_{issue,x+1} \geq t_{consume,x} + \Delta_{SNKtrig} \quad (2.2)$$

Like above $\Delta_{SNKtrig}$ must be determined by means of a delay analysis of a given implementation. Remember, however, that delays vary, and therefore we cannot determine exact values, but we have to make conservative estimates to be on the safe side.

Based on this strategy we can use two different approaches to implement the control of the triggers:

1. The use of coupled *timers* that – started with one trigger event (source or sink) – generate the other respective trigger event (sink or source) after the appropriate amount of time (T_{SNK} or T_{SRC}).
2. The use of a global time reference for source and sink from which periodic triggers for source and sink are derived with an appropriate phase difference, T_{Phase} .

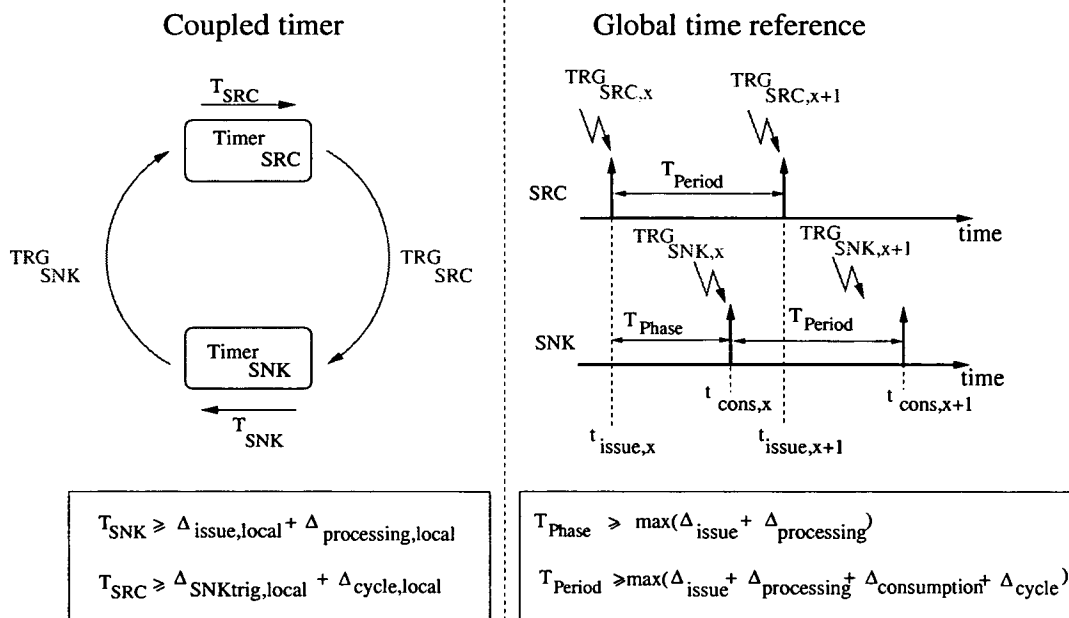


Figure 2.6: Fundamental Solutions in the Time Domain

The main difference between those methods is that the coupled timer approach only needs local delays, which are delays between the actual source-sink pair. In contrast the

global timer has to use the worst case delays of the overall circuit. Another difference is that the latter uses $\Delta_{consumption}$ while the coupled timer has to consider only $\Delta_{SNK,trig}$: Due to the fact that a timer “starts” a new trigger event only after an incoming trigger event has occurred, the difference between the point in time where data is ready to be consumed and the point in time where the trigger is recognized at the destination side does not matter. These strategies are capable of solving the fundamental design problem on all levels, since all delays have passed and the circuit is stable at the trigger instants. In some sense we have thus overcome the formal incompleteness of Boolean logic by condensing the missing information on validity and consistency into the timer settings and using dedicated control signals to convey this information between source and sink. Notice, however, that we have just *postulate* that the input vector will be consistent and valid after $\Delta_{issue} + \Delta_{process}$. In fact, we have no means to *directly* assess consistency and validity. As a result the determination of delays becomes a crucial issue. Two essentially contradicting arguments guide the choice of the timing settings:

1. *Restrictive assumptions*: It is not possible to determine any finite value for the delay without making assumptions on the implementation. Thus, the higher the delays the fewer assumptions must be made and the fewer restrictions apply to the implementation and the safer we can assume our losslessness property.
2. *Performance*: Obviously an overly conservative delay estimation has a negative impact on the throughput in terms of data words per second. In order to keep the resulting performance degradation minimal, a minimal overestimation of delays should be striven for.

So ultimately the choice of timing settings turns out to imply a tradeoff between performance and assumptions that have to be made on (and finally be met by) the implementation. Many models and techniques exist that allow determining delays for a given circuit topology and technology. However, since delay and skew depend on many parameters, an “aggressive” choice of timing settings towards maximum performance compromises the robustness of the circuit.

2.3.2 Information Domain

Alternatively we can tackle the other root of the problem, namely the formal incompleteness of Boolean logic. Different methods are available to enforce the different fundamental requirements:

Validity: Recall from Section 2.1 that a signal is termed valid if it is the stable result of a Boolean function performed on a consistent data word. There are several possibilities to judge on the validity of a signal:

- **Ensuring continuous validity:** If we can manage building the logic function unit in such a way that it produces only valid outputs, judgement of the output

signal's validity becomes trivial. A function unit of this type must change its output only in response to a consistent input word³. To this end it must (a) be able to judge on the consistency of the input word and (b) hold the last valid output signal during transient phases of inconsistent inputs. This obviously requires some kind of storage element for each logic function unit.

Even with an input perfectly changing from one consistent state to the other, skew within the function unit may cause invalid transient spikes at the output signal. Therefore, special care must be taken for the design of the function unit. This causes a trade-off with respect to the partitioning of a circuit into function blocks: A coarse-grained partitioning into few function units saves storage elements, while a fine-grained partitioning facilitates better control of skew effects.

If the signal is composed of more than one rail, continuous consistency in the rail domain is a necessary condition for continuous validity in the signal domain. This can be ensured by the employment of a grey-code on the rail level, e.g. [126]

- **Extending the signal code:** Another approach to make validity visible is to establish a more comprehensive alphabet in comparison to the binary Boolean logic (by using more than one rail per signal, e.g.) and to define a subset of all expressible codewords, which are considered as "valid". In contrast to the previous approach, direct transition from one valid codeword on the rail-level to the next is no more mandatory, (invalid) intermediate states are allowed, since they can be identified. In other words, if a valid codeword has been reached after a number of single transitions on k of n rails of a signal, there must be no other valid codeword that can be reached by transitions on the remaining $n - k$ rails. This allows us to identify unambiguously when a codeword is complete, irrespective of the order in which the transitions occur. The transition to the next codeword must include another transition on at least one of the k rails. The same condition – though in a different formulation – has been presented in [119].
- **Current sensing:** This method exploits the fact that transient effects in a circuit are associated with current flow. Unfortunately, however, the reverse is not necessarily true: The lack of dynamic current flow is indeed a reasonable indication that the inputs are stable (and hence consistent?) and the output is stable and hence valid. Without any restrictions on the delays, it may well occur that one slow rail transition arrives after the circuit has been considered stabilized. Another problem with this method is the lack of an event separating two successive identical data words, which substantially complicates consistency judgement. Finally the inclusion of analog circuitry for the current sensors causes additional technological efforts [48].

³Notice that ensuring continuous validity does not enforce continuous consistency, since the combination of valid signals pertaining to a different context does not yield a consistent data word.

Consistency: Imagine the situation depicted in Figure 2.7: SNK has an input vector composed by two signals, each of which are valid. This does not necessarily imply that the input vector is consistent, because the bits on the signal could belong to different contexts. Notice, that validity does not imply consistency, but consistency requires validity.

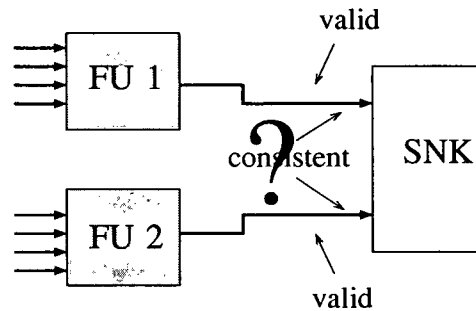


Figure 2.7: Validity vs. Consistency

To judge consistency, a circuit must be able to differentiate between two consecutive bits carried on a signal line, even if they hold the same information. This means that we have to choose a signal level code which relates information to context. So in order to be applicable for our purpose, a coding scheme must meet two conditions:

Consistency Condition 1: *Existence of transitions*

There must be at least one signal transition between any two successive code words. While this naturally happens in transition based coding schemes, it requires special efforts to ensure a transition between two successive identical data words in state based coding schemes. A usual solution is to introduce a "neutral" code word (like all zero, e.g.) between any two data words in a "return to zero" manner.

Consistency Condition 2: *Membership to contexts*

As can be viewed in Figure 2.3, two data waves (belonging to a different context) will transiently coexist between SRC and the associated SNK: There is a finite interval when the new data wave has already been issued and propagates through the FU, but the previous one is still valid at the SNK's input. This procedure is properly synchronized by trigger control. If more data waves were admitted between SRC and SNK we would lose control of them and in particular not be able to prevent one data wave from catching up with its predecessor (unless this is ensured by timing assumptions). As a consequence, if our basic requirement is to be able to distinguish data waves with different contexts, we normally come along with two disjoint code sets on the rail level, which allows us to unambiguously assign every bit to one of the two data waves.

Losslessness: As already outlined, the losslessness property requires us to provide the data source SRC with information when new data can be issued and the data sink SNK with information on when data can be consumed. The latter can be achieved by checking consistency and validity of the SNK's input vector without relying on the

time domain.

The source trigger can only be derived from information explicitly provided by the data sink such as a control signal acknowledging the consumption of the previous data word. Since there is only one single bit of information required on this backward path, there is no potential for skew effects. Nevertheless, the consumption of a data word can usually not be directly measured, which gives rise to conceptually weak compromises in this respect.

From a higher level of abstraction we can consider the function unit as part of the data source/sink and map the lossless requirement of a communication process problem (see Figure 2.8).

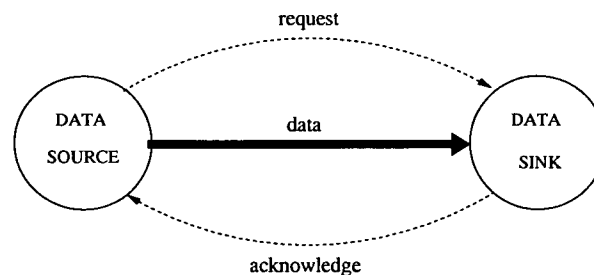


Figure 2.8: Communication Process

In fact there is a strong relation between communication channels and delay insensitive circuits [85]. However it is essential to realize that communication channels solve only a part of the fundamental design problem, namely losslessness. Consistency and validity cannot be answered by a communication channel alone, other mechanisms for this purpose are still required. Due to the fact that a lot of literature concerning communication channels in context with asynchronous logic [120] [131] [110] exists, we will give only a brief overview on it in this section. A data source and a data sink are connected over a *communication channel*. The point where a channel is connected is called a *port*. We distinguish between *unidirectional* and *bidirectional* channels. For the following we will consider only unidirectional channels which reflect the natural of communication in digital circuit. A port can be *active*, this means that such a port initializes a communication process, or *passive*, where the port reacts on incoming events.

Obviously there must be an agreement between source and sink in which way data is transmitted over the communication channel – a so-called *communication protocol*. Basically we can distinguish between a *2-phase protocol* and a *4-phase protocol*. In contrast to the 2-phase protocol, the 4-phase protocol returns back to its “neutral state” after each communication cycle. (see Figure 2.9)

Furthermore, we have to distinguish between *push channel*, where the data source is the active part, and *pull channel*, where the data source reacts on requests of the data

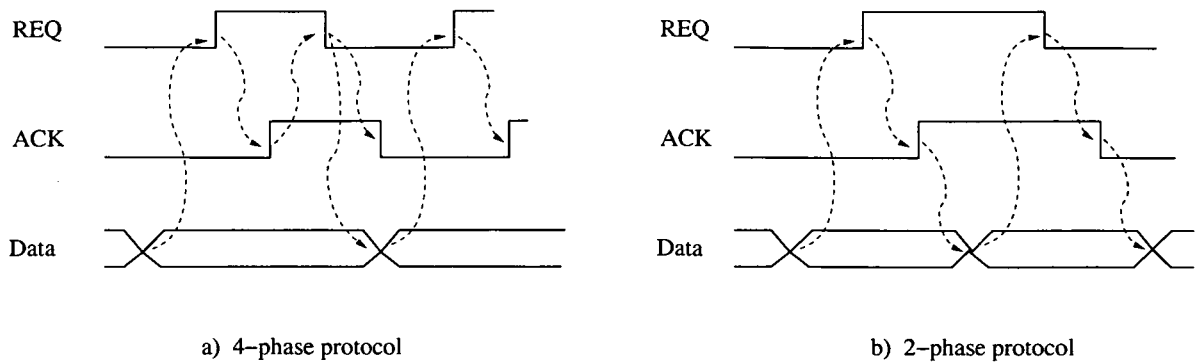


Figure 2.9: Communication Protocols

sink. A detailed description of communication mechanism with respect to asynchronous circuits can be found in [85].

2.3.3 Hybrid Solutions

It is not necessary to solve the fundamental design problem only in one domain. Quite on the contrary, many design approaches are based on a hybrid solution. Huffman codes [51] or micropipelines [111], e.g., solve only a part of the fundamental design problem and only their combination with other methods yields the desired result.

In most cases library cells, such as AND, OR, latches, etc., are implemented by making local timing assumptions e.g. isochronic fork [70] or fast local feedbacks [34][33], since it is quite easy to consider timing assumptions within such atomic cells and yield more efficient implementations in terms of speed and silicon area.

This leads to a further classification of circuits with respect to the assumptions made about timing [108]. Figure 2.10 shows a circuit fragment comprising three gates, where the output signal of gate A is connected to the inputs of gate B and C. The delays inside the gates Δ_A, Δ_B and Δ_C , represent the processing delays, while Δ_1, Δ_2 and Δ_3 , form the propagation delays of each wire segment.

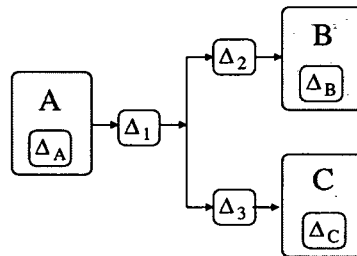


Figure 2.10: Circuit Fragment with Gates and Delays

Depending on the assumption made with respect to delays, circuits can be classified as follows [108]:

Delay-insensitive circuits (DI): We consider a circuit delay-insensitive if its correct operation depends neither on gate delays nor on wire delays. [70] shows that only circuits composed by Muller-C-gates and inverters can be delay insensitive using single output gates. This is a strong restriction, which limits the practical applicability of such type of circuit. However, this is the only class of circuits, which solves all aspects of the fundamental design problem exclusively in the information domain.

Quasi-delay-insensitive circuits (QDI): These circuits are delay-sensitive with the exception of some carefully identified wire forks. Related to Figure 2.10 this would require that $\Delta_2 = \Delta_3$. In other words, the QDI approach hypothesizes that all transitions at the end point of (carefully selected) wired forks occur at the same time. Such forks are called *isochronic forks*.

Speed-independent circuits (SI): These circuits operate correctly assuming that gate delays are bounded but unknown and that the wires are ideally zero delayed. Hence a SI implementation of the circuit depicted in Figure 2.10 would require that $\Delta_1 = \Delta_2 = \Delta_3 = 0$.

Self-timed circuits (ST): Forcing always (Q)DI or SI could result in an overkill – sometimes a tradeoff between implementation complexity and delay assumptions is reasonable. In this sense circuits whose correct operation relies on more elaborate and/or engineering assumptions are called self-timed circuits.

Timed circuits (TI): In this class of circuits all delays, gates and wire delays have to be taken into account in order to ensure a correct behavior of the circuit. In other words, such types of circuits solve the fundamental design problem entirely in the time domain.

Furthermore, different abstraction levels of a circuit implementation have to be considered. Until now we have dealt with abstract logic function blocks only, disregarding whether we are considering a simple inverter built from two transistors or a complex ALU. The distinction between abstraction levels is vital because several design approaches use speed-independent or quasi-delay-insensitive library cell implementations (on transistor level) and combine them yielding to a delay-insensitive circuit on gate level. In this way the timing analysis of arbitrary circuits is restricted to a small number of (little) library elements and hence has to be performed only one time during library compilation. This allows us to build circuits, such as an ALU, for which the fundamental design problem is entirely solved in the information domain (on this higher level of abstraction).

2.4 Design Techniques

This section is intended to give an overview about current design techniques with the aim to illustrate how they solve the fundamental design problem. Obviously not all design methods developed in the last half century can be covered. Instead, characteristic representatives of each design approach will be dealt with.

2.4.1 Synchronous Approach

Basic principle: The synchronous approach answers all subproblems concerning the fundamental design problem in the time domain using a common time reference (see Figure 2.6). It employs a unique control rail, the clock signal, to indicate validity, consistency and losslessness at the same time. At every active edge of the clock all signals have to be consistent and valid by definition and therefore ready to be consumed. Due to the fact that data sources get the same clock signal as data sinks, the active clock edge signalizes also the point in time where new data can be issued. In this way the regulation of the data flow is also strictly based on time and occurs without feedback (see Figure 2.11). By assuming that all data sources and sinks get a common global time reference from the clock signal, it is implied that all these components actually get the active edges at the same point in time. However, since skew and delay effect also affect the clock signal, this claim is not justified for deep sub-micron technologies. Quite on the contrary, [75] predicts that in the near future only a small percentage of the die will be **reachable** during a single clock cycle. Furthermore, the clock signal has no immediate relation to consistency/validity of signals or rails and the clock signal – it is just a strictly periodic and time driven control signal.

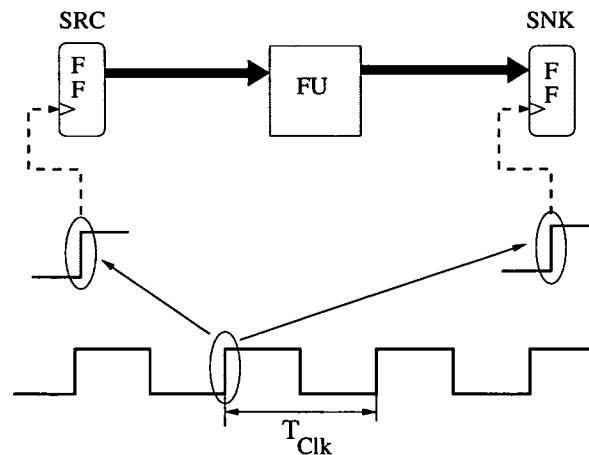


Figure 2.11: Synchronous Design Approach

The minimum distance between active clock edges T_{Clk} is derived from $\Delta_{process}$ and $\Delta_{consumption}$. Its calculation is based on worst case assumptions concerning physical properties, performable operations, applicable data and operation conditions [44].

Δ_{Cycle} and Δ_{Issue} are reflected in hold and setup time of registers. Note, that in the synchronous approach data is consumed and issued in exactly the same point in time. Further, it is assumed that both, data sources and sinks, are always ready to perform their operations on each active clock edge – flipflops have no means to signalize that they are busy at the moment.

Efficiency: The synchronous approach is extremely hardware-efficient, since it uses one single global control signal, which is easy to generate by means of a crystal oscillator. The highly efficient single-rail encoding can be used to represent all signals. If the logic state of a signal changes from one data word to the next, a signal transition is performed; if the state remains the same, no transition is required. Assuming a random distribution of state patterns on a signal, this yields to an average of 0.5 signal (=rail) edges per bit, which means that the energy consumption caused by data transitions is extremely low. Assuming a properly chosen clock frequency, no consideration of transient effects and consistency issues is required during functional design.

Through the insertion of so-called pipeline registers the signal path is often structured into smaller sub-paths. The timing of these smaller sub-paths can be more easily analyzed and in addition, pipelining yields some performance gains [47].

Problems: So apparently all problems of logic design are solved by the synchronous approach, and indeed millions of synchronous designs have been working properly and reliably over the past decades. However, substantial problems have remained unsolved on the conceptual level, and the current technology trends make these problems more and more evident:

- The indirect conclusion from time to consistency and validity of signals is the main conceptual deficiency: Time is easy to measure but not by itself an indication for consistency and validity. In fact, an artificial correlation between time and consistency and validity is extremely hard to establish and can never be guaranteed.
- The assumption of stable states during functional design does not eliminate the need for consideration of transient effects. In fact it only postpones the problem to an explicit timing analysis that is required later on. This timing analysis is often much more complicated than the functional design. With the increasing clock rates and the proceeding miniaturization this problem becomes more and more stringent.
- With its wide extension and the strong drivers required to keep delay and skew low, the clock network dissipates a significant share of the power of a chip. In order to be able to keep the clock skew within 300 picoseconds, the designers of the DEC Alpha CPU [106] developed a clock driver circuit, which dissipates over 40% of the power of the entire chip ([19]). Unfortunately, this outweighs the

advantage of low power consumption in the data path. In addition, substantial heat problems are caused by the fact that switching activities are periodic rather than demand driven.

- A solution of the delay and skew problems in the timing analysis phase is possible only if restrictions on the timing behavior are made. This, however, has severe consequences:
 - Considering that interconnect delays already dominate gate delays [104] realistic timing estimations can only be constructed after the place & route, i.e. at a very late point in the design process. In practice, however, timing problems often necessitate changes in the functional design. In this way the separation of functional design and timing analysis causes unnecessarily long iteration cycles.
 - Any change in the circuit or technology requires a complete revision of the timing analysis.
 - As already mentioned the actual delays of a given implementation still depend on the operating conditions and are affected by type variations. Hence the delay assumptions made during the timing analysis *must* be arbitrary to some extent. While assuming the worst case scenario within the specified range of operating conditions clearly leads to performance loss in the average case, there is still a residual risk of exceeding the assumed limits: *"...In order to achieve a reasonable shield against these variables, the clock period is extended by a certain margin. In current practice these margins are often 100% or more in high speed systems."* [19]. Some innovative design methods [114][92] soften this rationale by adopting the clock rate to the actual condition. However this requires an additional effort in terms of silicon die and control mechanisms.
- As a matter of fact no restrictions can be made for asynchronous inputs at synchronous/real-world borderlines and interfaces to other clock domains. Consequently these signals cannot properly be considered in the timing analysis and so metastability problems arise [36]. By use of additional synchronizer circuits metastability can be made sufficiently improbable, but no conceptual remedy to completely eliminate it has been found so far.
- Synchronous designs have a very problematic behavior with respect to EMC, since most of the energy is concentrated in one single spectral line.

2.4.2 Bundled-Data Approach

Basic principle: The basic concept of **bundled-data** [108] is to arrange several (data-) signals in a group and to use a common control signal, which serves as a trigger

to signalize validity and consistency of these (data-) signals. The control signal is generated at the same time as the related data signals by the source node and hence to operate correctly, the data path must be at least as fast as the control path. To ensure this procedure it may be necessary to insert additional delays, so-called *matching delays* in the control signal path. In this sense bundled-data solves consistency and validity in the time domain. The control signal can only be used as a trigger for data sinks and therefore the bundled data approach does not provide any means for data flow control. This requirement has to be fulfilled by other methods or on a higher system level.

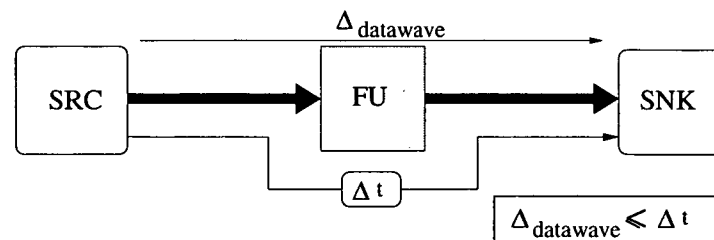


Figure 2.12: Bundled-Data Design Approach

As illustrated in Figure 2.12 consistency and validity are ensured in a similar manner as in the synchronous approach. This allows asynchronous designers to use standard (i.e. non hazard-free) implementations of logic function units [55]. The main difference between the synchronous and the bundled data approach is that the latter requires only local timing information (see coupled timer in Section 2.3.1) instead of taking into account the whole circuit to determine the temporal sequence of trigger events.

Efficiency: The most efficient representation of data is to use one single wire per bit – the higher the number of data bits, which are bundled, the closer the bundled-data approach moves to this maximal efficiency rate. Apart from the matching delays, which can be implemented using inverter chains or by duplicating the critical path of the stage between source/sink, no extra completion detection circuits are required. Assuming a random distribution of state patterns on a signal and a reasonable number of bundled signals, the bundled data leads similar to the synchronous approach to an average of 0.5 signal (=rail) edges per bit. Thus, the bundled data approach is highly efficient not only in terms of silicon area, but also in terms of energy efficiency. Due to this fact, bundled data was used in several asynchronous design implementations [41][58][90][112].

Problems: Although the major difficulty of the synchronous design style, namely providing a global time reference anywhere in the circuit, is defused by requiring only local timing information, the bundled data still faces some problems:

- Time is still used to determine consistency and validity of signals. The basic problem with this indirect conclusion is similar to those in synchronous systems, even if the locality makes it more manageable.

- The matched delays have to be calculated considering worst case scenarios. This yields to waste of performance.
- Due to the increasing dominance of wire delay over gate delay [49], matching delay can be determined reliably only after place&route. Furthermore, a validation of the final circuit is required, due to the fact that some variations during the fabrication may affect the (data-) signal path but not its related matching delay for example.
- Moving to a new technology all delay elements have to be re-calibrated.
- Bundled-data is usually used to model data busses. However, means for controlling the data flow are not provided.

2.4.3 Huffman Approach

Basic principle: D.A. Huffman [51] can be considered as one of the spiritual parents of the asynchronous logic design. Huffman developed the so-called fundamental mode circuits [86]. These circuits are intended to be used for asynchronous state machines. As depicted in Figure 2.13, Huffman circuits have primary inputs, primary outputs, and a feedback loop. The fundamental mode requires that only one input signal changes at a time. The current state is “stored” in the feedback path and thus, delay elements may be required to prevent state changes from occurring too rapidly. However, the feedback signals are inputs of the combinational logic as well – hence it is even required that by passing from one to the next state, only one bit changes. Therefore, the state encoding scheme has to be carefully chosen [108]. A further requirement of Huffman circuits is that the combinational logic is glitch-free, which can be achieved through redundant terms in the KV-map [86].

While validity is answered in the information domain (glitch-free functions) and by the environment (only one bit changes at the input side), consistency is solved by the delay element in the feedback path. The lossless property has to be guaranteed by the environment: It is assumed that a new input vector is issued only when the circuit has reached a stable state.

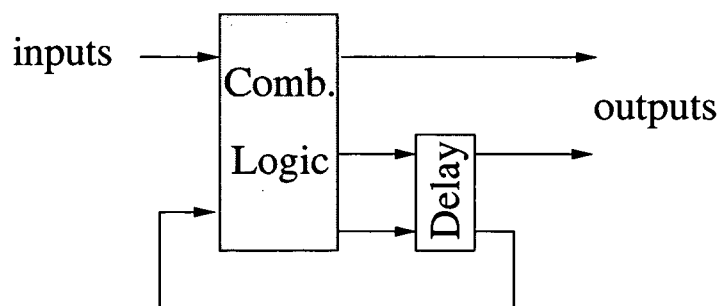


Figure 2.13: Huffman Circuit [85]

There are some enhancements of the Huffman circuit which soften the fundamental mode requirement. The *multiple input changes* (MIC) [37][66] extension is based on the assumption that the input changes happen within some tightly bounded interval of time, and hence they can be considered to have occurred simultaneously. Stevens [109] allows input changes at any time as long as they are grouped together in bursts. This yields to the so-called *burst mode* circuits. The most general mode of operation is the *unrestricted input change* mode (UIC) [115]. The UIC design method demands that an input does not change twice in a given time period.

Efficiency: Just like the approaches presented previously Huffman circuits use a single rail encoding. However, the Huffman approach does not allow glitches, albeit delay elements are used. The reason is that the delay element is not used to primarily signalize consistency, but prevents the circuit to become unstable due to the feedback signal. The demand of being glitch free limits potential optimizations during synthesis and leads to larger circuits. A lot of work has been done in this field and the interested reader can find further information about Huffman circuit synthesis approaches in [16][17] [96][123][130]. The restriction that a new input can occur only when the system has settled in a stable state, limits the throughput: A new input must be delayed at least two times the delay of the combinational logic (in the first step the next state is calculated, in the second step the output is settled according to the input and the new state information) and one time the delay of the delay element. Using a one-hot state encoding simplifies the associated logic but worsens the throughput further:

“... For a one-hot encoding, this means that a new input must be delayed long enough for three trips through the combinational logic and two trips through the delay element.” ([45] p.71)

Problems: The requirement posed by the fundamental mode but also by its extensions (MIC and UIC) lead to several limitations of the circuit design:

- One big handicap of Huffman circuits is that data paths cannot be implemented – a data bus carries information, which is arbitrary per definition and hence restrictions cannot be applied. This limits the practical applicability of the Huffman approach to control circuits only.
- The implementation of hazard free circuits requires an additional effort during system design. An in-deep discussion about *Hazard-free two-level logic synthesis* can be found in [85] on page 165 ff.
- Some boolean functions may not change monotonically during a multiple input change. Such functions are considered to have a *functional hazard*. Eichelberg [30] shows that it is impossible to build a hazard-free gate level implementation of a function, which has function hazards.

- Although glitch-free function units have to be used, delay elements are still required. The same drawbacks concerning delay elements, as mentioned in the previous sections, are true for the Huffman circuits.
- No means for data flow control are included – it is postulated that data is issued in a correct manner by the environment. The fact that the points in time where data can be issued depend not only on a straight forward delay calculation of function units, but also on the delay calculation of circuits containing loops, aggravates this weak point.

2.4.4 Design Techniques Using Signal Coding – The NCL Example

Basic principle: Many approaches exist, which use signal encoding to ensure validity of signals and make consistency of signal vectors directly visible [65]. NCL (Null Convention Logic), which is developed by *Theseus Logic* was chosen as the representant of this class of implementation approaches, due to the fact that it is the most mature one and industrial experiences have been already made [78]. This design approach extends the Boolean logic by a so-called NULL state [34]. In particular an NCL signal can assume a DATA state – which is either a valid HI or a valid LO, in NCL called "TRUE" or "FALSE", respectively – or a NULL state. For encoding these three states the single-rail approach is obviously not sufficient, and a two-rail signal representation is used instead, with NULL being represented as (0,0), TRUE as (1,0) and FALSE as (0,1). The NULL state does not convey any information, it serves only as a neutral state separating two consecutive codewords. Figure 2.14 illustrates this behavior.

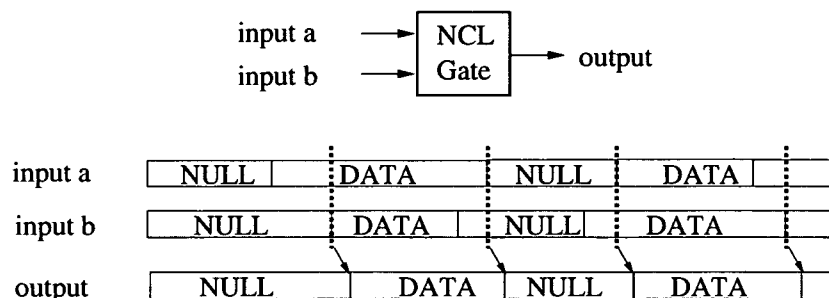


Figure 2.14: Sequence of DATA and NULL Waves

Feedback ensures that new data (DATA or NULL) can be processed only when the input vector is consistent. To realize this behavior so-called threshold gates are used [64]. These gates change their output only when the complete input vector is either DATA or NULL. This hysteresis provides a synchronization of the wavefronts on the gate level. In other words consistency and validity checks on signals are implemented at gate level. With the proposed encoding on signal level, exactly one rail changes its logic level upon the transition from NULL to DATA and vice versa, regardless of

whether DATA is TRUE or FALSE. Due to the mandatory introduction of the NULL waves a neutral state (NULL) is assumed on every signal after every single data word, which enforces the edge required to meet the consistency condition 1 (see Section 2.3.2). From this neutral state an edge on any of the two rails leads to the TRUE or FALSE state, which guarantees that the codeword itself is always valid. The NCL approach does not provide any mechanism to ensure losslessness.

Efficiency: A NULL state between each pair of DATA states regulates the data flow in onward direction and ensures consistency. From a performance point of view this convention is very expensive – in fact the maximal achievable throughput is halved by the NULL wave. However, due to the fact that this approach does not require any delay elements, the resulting circuit operates as fast as it can, which partially compensates the drawback of the NULL wave.

In contrast to single-rail encoding styles where the average of 0.5 signal (=rail) edges per bit can be assumed, NCL requires in any case two edges per bit on the rails.

The usage of two rails per bit yields by its nature to larger circuits compared to single rail implementations. Furthermore, each NCL primitive requires some kind of storage element, which increases again the price in terms of silicon area. However, Theseus Logic proposes some tricky hardware solutions, which keep this overhead within reasonable limits [33].

Problems: The NCL approach integrates data and control information in a single expression. This merger combined with the alternation of DATA and NULL waves makes validity and consistency directly visible, without making (apparently) any assumption about timing – this feature has its price:

- Higher effort in terms of gates and interconnect: the dual rail encoding doubles the number of wires and multiplies the size of logical gates: A gate with two single-rail inputs has to take in account four possible input combinations, while a two dual-rail input gate has sixteen possible input combinations.
- The convention that NCL gates start to produce a new output value only when all inputs are in the NULL/DATA state requires that the gate holds its output value in between. As a consequence, an NCL gate must contain some kind of memory element inside. Theseus Logic proposes *threshold gates* for this purpose. The functionality of these gates is basically implemented using feedback signals inside the gate. Although NCL does not require timing assumption on gate level, to operate properly the feedback signals inside the gates have to be fast enough to settle the gate before the next input vector change occurs. This is a sustainable requirement, however, due to the fact that a timing assumption has to be made, NCL circuits have to be classified as quasi-delay-insensitive circuits rather than delay-insensitive ones.

- The NULL waves reduce throughput on the one side and energy efficiency on the other side (see previous paragraph)
- NCL does not provide any means for data flow control. This means NCL has to be combined with other design techniques such as e.g. *Micropipelines*. For this purpose consistency of a signal vector has to be provided explicitly to the additional design method. This requires a further circuit, so-called *Completion Detection Circuit* (CMPD).

2.4.5 Transition Signaling Approach

Basic principle: In conventional coding techniques logic states of signals are mapped to voltage levels of physical rails. In contrast transition signaling [108] uses *edges* on rails to convey the information. Transition signaling also employs two-rail coding on the signal level. A transition on one rail indicates a HI, a transition on the other rail a LO. From a more abstract point of view transition signaling uses a one-hot encoding scheme for HI and LO and therefore fulfills the validity property on code level. The neutral state between consecutive codewords is defined by the absence of transitions on the rails. In contrast to NCL where the neutral state must be explicitly generated, transition signaling provides this state automatically and hence a new codeword is recognized even if it carries the same information as the previous one. In this sense consistency is integrated directly in the coding style. In [71] it has been shown that the only single output gates that can be used in conjunction with transition signaling circuits are Muller-C-Gate and inverter. This limits the usability of this scheme for real circuits.

Efficiency: Transition signaling can be compared to an NRZ coding style. This obviously favors the achievable throughput and hence promises higher performance for circuits using this approach. Albeit transition signaling uses a dual rail encoding, only one single transition/edge per bit is required. Note that a transition occurs in any case, even if the same bit information is transmitted consecutively by the same signal. Thus, data content itself does not influence the number of edges required to convey the information.

Compared to single rail encoding, the dual-rail approach doubles the number of wires. However, the main weak point of transition signaling with respect to area efficiency is the complexity of gates, which have to be able to operate on signal transitions instead of signal levels.

Problems: Coupling information to events is an extremely elegant method to solve the fundamental design problem concerning validity and consistency. Nevertheless, there are some (practical) problems, which inhibit the breakthrough of this design technique:

- Gates require a high implementation effort due to the fact that they operate on edges instead of signal levels. Furthermore, the set of allowed gates is limited, this restricts the practical applicability of this design style.
- The basic principle of digital design is to distinguish between two discrete signal states/levels, namely LOW and HIGH or '0' and '1'. Transition signaling is based on transitions of signals instead of levels of signals, which means that transition signaling is event-based instead of state-based. Hence, this approach requires to completely change the well established and approved way of thinking concerning digital circuit design. This radical change demands not only new tools but also a complete re-education of engineers.
- Transition signaling circuits are susceptible to interferences. Each glitch even the smallest one produces two edges, which are interpreted per definition by a transition signaling circuit as two valid bits. Muller-C gate implementations as proposed in [111] moderate this problem, since they are more robust against glitches. In spite of the risk that a small impulse generated by an electrical interference, e.g., causes a malfunction, is much higher than in other design approaches.
- The fact that transition signaling is event-based makes it extremely difficult to debug transition signaling circuits. Debug tools cannot directly derive the logical information carried by signal – instead the event sequence must be journalized to determine the information, which is currently conveyed by the signal.

2.4.6 Handshake Protocols: The Micropipeline Approach

Basic principle: There are several choices of handshake protocols, which can be used to control the communication inside a circuit [85](see Section 2.3.2). The **micropipelines** introduced by Sutherland [111] in particular use a 2-phase signaling for the handshake protocol. Basically, micropipelines are means for structuring complex logic designs in general and data path designs in particular. In contrast to synchronous pipelines, they employ local handshake signals between any two pipeline stages to interlock the inter-operation between the individual stages so that the speed of data flow can be adapted to the local situation. They provide an elastic pipeline for the handshake signals that allows buffering requests. In this way the micropipeline approach provides a straightforward solution for data flow control.

The latches inside the micropipeline have two operation modes:

- **Transparent:** input data is passed directly to the output.
- **Frozen :** the latch maintains the value of the output independently of the input data.

As illustrated in Figure 2.15, the latches have four control signals, by means of which their behavior can be controlled: *Capture*(C) and *Capture_done*(Cd) as well as *Pass*(P)

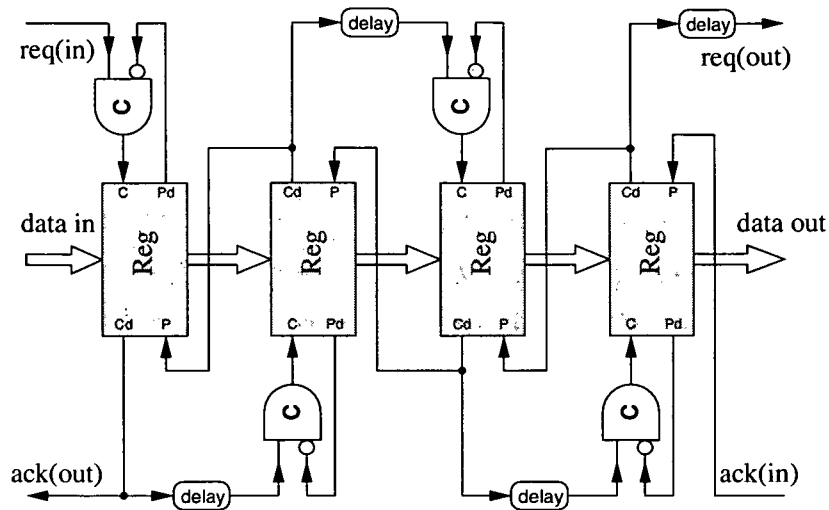


Figure 2.15: Micropipeline

and *Pass_done*(Pd).

The *Pass* input sets the register in the transparent mode. After a certain delay the register achieves this state, which is signaled by the *Pass_done* signal. Similarly, the *Capture* and the *Capture_done* signals freeze the latch and signalize that the latch is effectively frozen. The *Muller-C gate* [111], which acts as AND concatenation of events, ensures that the latch freezes only when new input data has been passed through the register. The original micropipeline approach employs *delay elements* to ensure consistency. Fundamentally, this corresponds to bundled data circuits between pipe-latches. However, it is possible to generate the completion signals by combining the micropipeline with other design approaches.

Efficiency: First of all the micropipeline approach provides a mechanism to control the dataflow. Like its synchronous counterpart, the micropipeline can be further used to enhance throughput of circuits. Especially the micropipeline introduced by Sutherland seems to be particularly suitable for this purpose due to fact that it implements a 2-phase-handshake protocol. This means that no *Return-to-Zero* is required, which shortens the cycle time. However, practical experience shows something quite different: Based on this argument the first asynchronous ARM processor *Amulet1* [129] was developed using techniques based extensively on Sutherland's Micropipelines. For the second processor generation, *Amulet2(e)* [40], a 4-phase-handshake protocol rather than a 2-phase-handshake protocol was chosen because it was discovered to be simpler and more efficient.

Examination of area efficiency is not meaningful when we only speak about communication protocols. To implement the function unit between pipe stages, micropipelines have to be combined with an other design style – bundled data was suggested by Sutherland e.g.. Therefore, the area efficiency depends strongly on the chosen method

for making consistency visible.

Problems: The Micropipeline approach is intended to solve only one part of the fundamental design problem, namely the data flow control. Weak points of this approach are:

- The original micropipeline introduced by Sutherland provides a bundled data approach to signalize consistency. This includes all problems mentioned in the section about bundled data to the micropipeline approach as well. However, the bundled data design style can be replaced by any other approach, which solves the consistency problem.
- Apart from the delay element parallel to the data path, two additional delay elements are required inside the latches: a *capture_done delay* and a *pass_done delay*. This vast use of delay elements cuts down the potential advantage of asynchronous circuits.
- Although the 2-phase-signaling used for the handshake protocol promises higher performance compared to a RTZ protocol, the practice has shown the opposite: The second generation of the AMULET processors was based on a 4-phase-signaling handshake protocol due to the fact that the 2-phase-signaling in the first processor generation permitted only a slow and complicated implementation.

2.4.7 High Level Description Approaches

Basic principle: In contrast to all methods discussed so far, *High Level Description Approaches* do not explicitly consider the effective hardware implementation of the circuit, but outsource this aspect to an (automated) synthesis process instead. Hence the main task of these high level methods is to purvey a description, which fulfills specific constraints/requirements in order to enable the synthesis tool building correctly operating circuits. However, the synthesis process on its part has to revert to one of the “low level” design approaches described previously. Therefore, related to strategic options high level description methods do not pose a new design technique, they provide a framework to develop circuits and to formally verify its behavior instead. High level descriptions fall roughly into 2 categories, namely *Graphical methods* and *Translation methods*.

Graphical methods: Due to the fact that Petri nets [105] are used to describe concurrent systems, almost all of the graph-based methods are based on this graph model or on a restricted form of it [57]. *Signal Transition Graphs* (STG) introduced by Chu [14] are such a restricted form of a Petri net, which allow only limited options to select alternative responses of the circuit. Other variants of Petri nets are *Interface nets* (I-nets) [80] *Machine nets*, (M-nets) [103] or *Change Diagrams* [118]. *Timed*

Event/Level structure (TEL) is a graphical method, which allows specifying timing information [99], in order to permit efficient circuit implementations.

Translation methods: Almost all high-level description languages for asynchronous circuits are based on the use of a language that belongs to the *Communicating Sequential Processes* (CSP) [13] [12] family, rather than to classical hardware description languages such as VHDL [62] or Verilog [121]. The characteristics of CSP are described in [85] as follows:

- Concurrent processes
- Sequential and concurrent composition of statements within a process
- Synchronous message passing over point-to-point channels (supported by the primitives send, receive and – possible – probe)

OCCAM [113][91], *LOTOS* [132] and *CCS* [79] are programming languages which are able to describe parallel processes. *Tangram* [9], *CHP* [69] and *BALSA* [5] are languages which are specially designed to model (concurrent) asynchronous circuits.

Efficiency: In general high level descriptions permit shorter development cycles due to automated processes below the abstraction level of the description. Today, global optimization techniques for asynchronous logic are difficult to utilize during the translation process and hence automated synthesis often produces inefficient results [57]. However, it is a matter of time until asynchronous synthesis tools achieve the same quality as its synchronous counterparts.

Problems: There are mainly three problems which can be identified concerning high level descriptions:

- Although the asynchronous design style has a long history, interest arose only in the last decade and thus researchers and engineers started to investigate this discipline. It is clear that existing approaches and tools are not fully developed yet.
- Only circuits with limited complexity can be modeled. This is especially true for graphical-based approaches due to their awkwardness in specifying input choices [57].
- The automated synthesis process hides the information about the implementation on gate level. Having a well approved and established tool chain this may be a desired property, but as the asynchronous design techniques are being still in the fledgling stage this circumstance limits the possibilities to investigate the implemented circuit and to find out possible improvements.

2.5 Comparison

Due to the fact that different design techniques are intended for different purposes – Huffman circuits for ASFMs, bundled data for data path modeling e.g. – and because each design style has a lot of extensions on its part, it is difficult to make a comparison. Thus we will confront the presented design techniques with respect to basic aspects and compare them only in a qualitative manner. This should still enable the reader to judge the presented design techniques and visualize their advantage and drawbacks.

Covered part of the fundamental design problem: The most characteristic features of a design technique are the aspects of the fundamental design problem it covers and the domain (time or information) in which the related problems are solved. Hence in Table 2.1 the presented methods are compared with respect to the domain, in which they solve consistency, validity and losslessness. The column *E* (Environment) is used to express that the design technique does not solve the corresponding subproblem, but moves the responsibility to the environment. Column *I* (Information) and *T* (Time) are used to express whether the problem is solved in the information or in the time domain.

	Validity			Consistency			Lossless		
	T	I	E	T	I	E	T	I	E
Sync	x	-	-	x	-	-	x	-	-
Bundled Data	x	-	-	x	-	-	-	-	x
Huffman	-	x	-	x	-	x	-	-	x
NCL	-	x	-	-	x	-	-	-	x
Trans. Sig.	-	x	-	-	x	-	-	-	x
Microp.	(x)	-	-	(x)	-	-	-	x	-
High Level Desc.	-	x	-	-	x	-	-	-	x

Table 2.1: Comparison wrt. the Fundamental Design Problem

In contrast to all other methods the synchronous approach provides a complete solution of all subproblems of the fundamental design problem in the time domain. On the one hand the clock signal guarantees consistency and validity at the instant when data is taken over and on the other hand it regulates the data flow. The bundled data approach is intended to soften the problems concerning distribution of a global time reference by using local timing assumptions only. It makes consistency and validity “visible”, but leaves data flow control issues unconsidered. Similarly, the Huffman circuits move the responsibility to provide only “allowed” inputs at the right time to the environment. In the same way the NCL approach alone does not provide any means to control the data flow. However, the alternating data waves in combination with the completion detection signal make this approach particularly suitable to be extended

by a communication protocol, which controls the data flow. Due to the event based approach and the one-hot-encoding for events, transition signaling also solves consistency and validity in the information domain. Means for controlling the data flow are not provided. In contrast, the micropipeline is a concrete implementation of a handshake circuit and thus intended for data flow control. Sutherland suggested to combine the micropipeline with the bundled data approach to build function units inside pipe stages. Therefore, consistency and validity are solved in the time domain. It is difficult to classify high level design methods due to the multitude of different techniques covered by this category. In general, these methods demand some restrictions concerning input vectors, which have to be abided by the environment. Consistency and validity are largely solved in the information domain by these methods.

Area and energy efficiency: Other important aspects are the area and energy efficiency. Basically the number of gates, which are required to implement a given functionality depends on the used design style. However, specific technologies favor certain design styles – furthermore, the degree of customization of basic gates has a crucial impact on the resulting circuit size. So to provide a quantitative expression not only the design style, but also the used technology (CMOS, NMOS, ...) and the degree in which basic (library) gates are adapted to a given design approach, has to be considered. The same is true for power consumption. As a consequence, a quantitative analysis permits a comparison of circuits with highly specific implementations as illustrated in [65]. Instead, this section claims to provide a generic overview and hence the design styles will be investigated with respect to area and energy efficiency from a qualitative point of view only. In Table 2.2 the comparison with respect to area is subdivided in three aspects: (i) *wires per bit*, which indicates the number of wires representing a bit. (ii) *gate size*: this defines the number of boolean basic gates (AND, OR, INVERTER), which are necessary to build an AND-gate of the analyzed method. It is clear that specific implementations yield to a much better solution in terms of transistor count. However, we will use standard logic basic gates as reference points, to get a suitable comparison. (iii) *add. circuits* indicate if the design technique requires additional circuits apart from the implementation of the logical function itself in order to build working circuits.

Based on the fact that (C)MOS poses the state of the art technology for circuit implementation, the energy efficiency can be roughly drawn back to the number of edges which occur within a circuit. Hence with respect to energy efficiency, we distinguish three scenarios: (i) *worst case*, where it is assumed that the signal toggles in each cycle from *TRUE* to *FALSE* and vice versa. (ii) *average case*, where a random distribution of the signal states is assumed, and (iii) *best case*, where the signal always keeps the same information.

Synchronous and bundled data approaches have similar characteristics concerning their area efficiency. The main difference lies in the method to distribute the timing information: The synchronous style uses a global time reference, which is distributed

	Area			Energy (transition per bit)		
	wire/bit	gate size	add. circuits	worst-	average-	best case
Sync	1	1	Clock tree	1	0.5	0
Bundled Data	1	1	Delay elem.	1	0.5	0
Huffman	1	1+	Delay elem.	1	0.5	0
NCL	2	8	CMPD circ.	2	2	2
Trans. Sig.	2	?	MullerC gate	1	1	1
Microp.	1	1	Delay elem.	1	0.5	0

Table 2.2: Comparison wrt. Area and Energy Efficiency

over a clock tree, while bundled data uses coupled timers, which can be implemented using delay elements. The $1+$ entry in the gate size column of Huffman codes should indicate that this approach can basically use the same gates as the previous methods, but an additional effort in terms of gates is required to ensure that the resulting function unit is glitch free. Using signal coding, the NCL style requires 2 wires to represent a bit. As a consequence, the size of the basic gates increases exponentially: From the truth table depicted in [64] it is easy to derive that an NCL-AND gate can be built using six conventional gates (four AND and two OR gates). To guarantee that the output keeps its old value having inconsistent inputs two additional gates to memorize the output value of each wire are required.⁴ Based on the bundled data approach, the micropipeline also shows its characteristics concerning area efficiency. With respect to energy efficiency, the first three approaches quoted in Table 2.2 show foreseeable behavior: If the signal state does not change, no edges occur, if the state changes, each time then an edge always occurs. The NCL approach instead shows a more surprising characteristic: in each scenario (even in the best case!!) **two** edges occur per bit: Based on an RTZ scheme, NCL has to transmit each information bit twice – in the first step the effective information is emitted and to return back to the neutral state the previous information has to be inverted and sent again. Also the transition signaling approach does not show any difference concerning the number of edges between the best and the worst case. The reason therefore is that the information itself is coupled to the signal edges and hence even if the same information is consecutively transmitted over the same signal line, one edge per information bit takes place. As expected the micropipeline shows similar to the area efficiency considerations the same characteristic as the bundled data approach.

At this point it is important to highlight the distinction between energy efficiency and power consumption. The first describes the energy which is required to transmit one single bit. The latter is the energy that a circuit dissipates over time. In general, asyn-

⁴It is clear that a memory element is much more complex than a simple AND gate for instance. Due to the fact that an NCL basic gate does require a full memory element, but a solution similar to a *transition gate* lasts out in a dynamic logic style, we have equated these memory elements with two standard gates in Table 2.2.

chronous circuits operate only when required (=event-driven), a synchronous circuit is always triggered by a periodic clock signal. Therefore, asynchronous circuits have a worse energy efficiency than the synchronous ones and they may still consume less power than their synchronous counterparts.

Chapter 3

Code Alternation Logic – CAL

Contents

3.1	Background of CAL	39
3.2	Coding Scheme	40
3.3	Control Flow	42
3.4	Levels of Abstraction	43
3.4.1	Behavioral Description – cal_logic	43
3.4.2	Functional Description – cal_rail_logic	46
3.5	Basic Gates	47
3.5.1	AND Gate	47
3.5.2	Phase Detector	47
3.5.3	φ -Converter	48
3.5.4	CAL Register	49
3.6	CAL Design-Flow	50
3.7	Simulation Concept	52
3.8	Summary	54

As implied by the name, the major part of this section will consist of the coding of signals, but CAL provides much more. The system consists of a tool-set to realize asynchronous circuits which are automatically compiled in several stages. All these steps are performed with the Synopsys design compiler by the use of synthesis-scripts. Furthermore, a simulation concept is added to be able to prove the functional description of the circuit as well as to ensure the correctness of the synthesis. This tool-set allows us designing a 16 Bit processor based on CAL, and putting it successfully into operation. This chapter will give a detailed step-by-step introduction into CAL.

3.1 Background of CAL

CAL can be classified as a *signal coding method*, which solves the fundamental design problem from Section 2.2 in the *information domain*. Let us recall these terms:

With delay-insensitive circuits a method is provided to design asynchronous circuits in a way that their behavior is independent of the speed of their components or the delay on the wires. They are correct by design. A further big advantage of such circuits is that the circuit can derive information whether the computation has finished or not. Only the time needed for this computation is used for waiting rather than the worst-case time.

Signal coding describes a coding system, which is widely used to design self-timed systems. Design methods using signal coding can be split up into several approaches by means of how data is encoded. The traditional style – the 4-phase dual-rail¹ approach – uses three logic states, which can be formed with the two rails: "1", "0" and "invalid". There is a separate spacer used between every change of the state. This spacer token is necessary to distinguish whether a new data wave had begun or not. So the throughput is reduced to the half.

This disadvantage of needing such spacers is not given by the other popular dual rail technique – the transition signaling. But this approach has also its drawback: As shown in Section 2.4.5, the actual state of rails cannot be determined by simply looking on it: Because an internal state "00" of both rails could represent a logic "1" as well as a "0". It depends on the context in which this transition has happened.

So the idea is to combine these two approaches and to try to eliminate the two drawbacks: On the one hand it should be possible to transport information every cycle, on the other hand it should be possible to determine the value without considering the history. We have designed a coding scheme that is based on the alternation of code sets as shown in detail in the rest of this chapter. There are two similar approaches from the early nineties: [20] introduces the *Level-Encoded 2-Phase Dual-Rail (LEDR)* and [77] the same coding technique *Four State Asynchronous Architecture*:

Level-Encoded 2-Phase Dual-Rail (LEDR) [20] denotes three different hardware

¹In this context, we use the term dual-rail to describe a signal consisting of two rails. The instance how data is coded is not defined so far.

implementations of the LEDR principle: The first is based on a PLA-structure, the second on a self-timed Domino logic structure with dynamic storage, and the third implementation uses series stack of transistors. There is, however, no design methodology given how to build logic with this gates. Further work in the LEDR field is done by [101, 100] where four input *Phased Logic* gates are used as computational elements. Therewith a net-list of D-Flip-Flops and combinational logic driven by a single clock can be automatically synthesized.

Four State Asynchronous Architecture This approach uses only multiplexors and the authors claim that this allows reducing complexity. Furthermore, the multiplexors have been optimized at the transistor level and have been implemented in 2 μm CMOS technology in 1991. This approach is optimized with respect to speed. Best performance is achieved using dynamic latches because they are smaller and faster. [76]

As pointed out in Section 2.2 the fundamental design problem leads to the two fundamental requirements, which are the main parts of the next sections.

3.2 Coding Scheme

The key idea of CAL is to use two disjoint code sets for representing the logic state of a signal. The additional information denoting which code set is being applied is called the *phase* of a signal, φ_0 and φ_1 respectively. The representations are used alternatively, so within a sequence of data words each bit can uniquely be assigned to the corresponding data word.

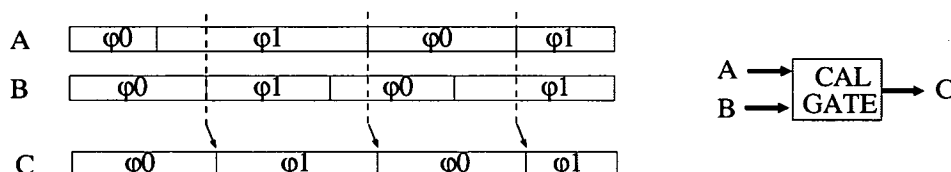


Figure 3.1: Flow of Data Waves in CAL

Figure 3.1 shows the flow of data waves in CAL: Due to the alternation of φ_0 waves and φ_1 waves it becomes easy to synchronize signals within a data word even in the case of arbitrary skew.

Two logic states in two representations lead to the need of four code words, which can be encoded with at least two rails a and b. Table 3.1 depicts the used state assignment:

Table 3.1 and Figure 3.2 show the important property of CAL: If data words are coded in alternate phases φ_0 and φ_1 , every valid transition from one phase to the other changes exactly one level of one rail:

logic state	code φ_0	code φ_1
"LOW"	(a,b)=(0,0)	(a,b)=(0,1)
"HIGH"	(a,b)=(1,1)	(a,b)=(1,0)

Table 3.1: CAL Coding Scheme

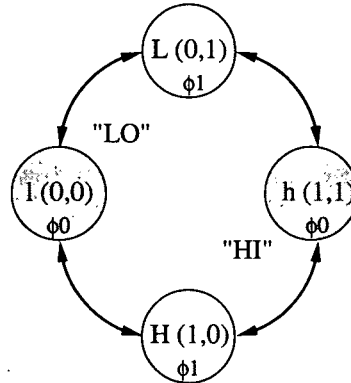


Figure 3.2: Possible Phase Transition

A logic "LOW" in phase φ_0 can only be followed by a "LOW" or a "HIGH" in phase φ_1 . In the first case the level of the rails changes from (0,0) for "LOW" in φ_0 to (1,0) for a "HIGH" in φ_1 . The second transition leads to (0,1) for the representation of "LOW" in phase φ_1 .

As seen in Table 3.1, CAL uses a dense code which means that every bit combination is used for describing a valid code word. There is no representation for the state *invalid*. Recall that one of the three requirements in the information domain (see Section 2.3.2) is validity: In the case of CAL *continuous validity* is ensured. So every gate has to guarantee a valid output signal. As described above, exactly one transition is needed to change from one valid code word to another valid one. This fulfills both conditions for *consistency* representing the second part of the fundamental requirement: The demand for the *existence of a transition* is met due to this fact as well as the *membership to context*: If there is *exact* one transition between every code word, every transition will change the context and so the membership can be derived. The impact on CAL by the fundamental design rules leads to the following important rules which are summarized here:

- I: Data values of each signal must be coded in alternating phases.
- II: The calculation is performed when all input signals are in the same phase.
- III: In the case that the input signals are in different phases the output has to remain in its last valid state.

Until now it looks like that CAL solves all problems in the information domain – in other words it is delay insensitive. In fact CAL is a hybrid solution as described in

Section 2.3.3. Concerning a design built with CAL-gates – the CAL approach is *delay insensitive*. Neither assumptions are made on gate delays nor on wire delays. However, a closer look at the CAL gates shows that there are timing assumptions, e.g. for local feedback loops in latches. The resulting constraints for the design can be solved within the basic gates. The information to build these gates in a correct manner is stored in specific libraries.

Both, validity and consistency are needed to solve the *fundamental requirement 1*. The second one will be the target of the next section.

3.3 Control Flow

The design rules of Section 3.2 must hold for the whole design, so they must be valid for pipeline structures too. Rule I defines that the code set used in CAL alternates with every data word. This means that a bit that has been part of a valid code word in φ_0 becomes invalid in φ_1 . Recall the *fundamental requirement 2* from Section 2.2, which states that some kind of feedback is needed. Figure 3.3 shows the pipeline structure where the feedback is represented in terms of `capture_done` signals to trigger the source firing. The sink can derive its trigger condition directly from the data wave: If all bits of a data word are in φ_0 , the data word is consistent and can be consumed. As soon as several bits change to φ_1 , the φ_0 bits become obsolete and the data word is inconsistent until the last bit has changed to φ_1 as well. Obviously, some kind of synchronization is required to prevent that a fast φ_0 bit, e.g., catches up with the preceding φ_0 data wave.

This is, however, easy to achieve by the inclusion of a hysteresis in the logic functions: Similar to the approach used in NCL the output of a logic gate in CAL changes only when the data word at the input is consistent as defined in rule III. In Figure 3.3 a simple linear pipeline is shown:

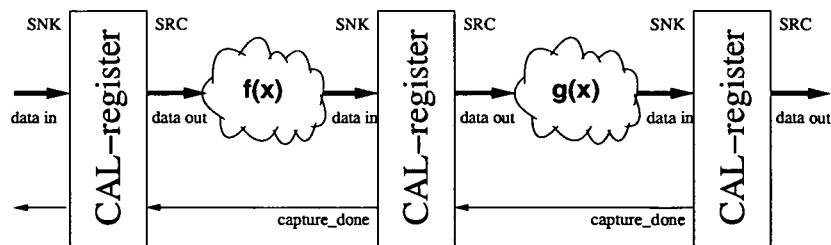


Figure 3.3: CAL Pipeline Structure

To explain the functionality of the pipeline structure the stage in the middle is used. There are two conditions causing this stage to fire:

1. The upstream logic function $f(x)$ has completed its calculation and so the data on the input of this stage is ready to be captured. This information can be retrieved directly from the data word.

2. The downstream stage has already caught the previous wave (the result of $g(x)$) and so the data of this stage is not needed any longer. The downstream stage provides this information with the `capture_done` signal.

Recall Figure 3.1 which shows the flow of data waves in CAL: Due to the alternation of φ_0 waves and φ_1 waves, it becomes easy to synchronize signals within a data word even in case of arbitrary skew. It can be verified that all three rules defined in Section 3.2 are fulfilled.

3.4 Levels of Abstraction

It is not very comfortable to design logic circuits using a rail representation as described in Table 3.1. Furthermore, it is not possible to use existing synthesis tools, because they are designed for single rail logic used in synchronous designs. This leads to the need of two different descriptions of CAL: One for the designer and another one for the tools. Both definitions are written in VHDL in our case, but it is also possible to transform the representations to Verilog or any other hardware description language.

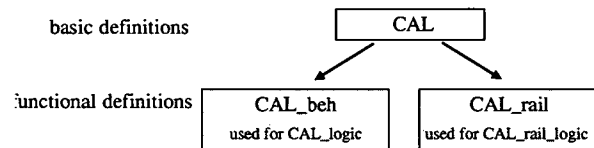


Figure 3.4: Library Dependencies

As shown in Figure 3.4, the library structure is built hierarchically: The CAL library is the root of all other libraries and provides basic type definitions for all others. All common definitions for the behavioral and the rail style of CAL are given there. Furthermore, some basic conversion functions are provided. This library will be used in every step of the asynchronous design as well as in the testbench. In addition to the CAL library, the libraries `cal_beh` and `cal_rail` contain functionality needed for the corresponding logic. Both logic systems, `cal_logic` and `cal_rail_logic` comprise for example a logic AND. In `cal_logic` this function has two single rail inputs and one single rail output, while in `cal_rail_logic` the same function requires dual-rail signals. In summary, these two libraries provide functions with the same purpose but with the logic types needed for the logic system actually used – `cal_logic` or `cal_rail_logic`.

3.4.1 Behavioral Description – `cal_logic`

The definition of `cal_logic` is the interface for the human designer. A single rail, multi value code is used to describe the four states of CAL. As shown in Table 3.2, the different states are specified with lower and uppercase letters "l" and "h". Phase

logic state	code φ_0	code φ_1
"LOW"	l	L
"HIGH"	h	H

Table 3.2: cal_logic Coding Scheme

φ_0 utilizes the lower case versions "l" and "h", for the logic states "LO" and "HI", and likewise "L" and "H" are applied for φ_1 .

To provide full simulation and synthesis support for traditional design tools it is necessary to define several VHDL types and classes. At first, a new data type has to be declared. In the case of cal_logic the four states have to be defined. Furthermore, it is not enough to build a four-value type, since a reasonable simulation tool needs to differentiate more possible values. There has to be a value which sets a signal to undefined, e.g. at the startup. Moreover, the simulation should be able to handle the situation when two outputs drive one signal and both of them want to assign a different value. This definition is very similar to the std_logic data type of the std_logic_1164 standard for the VHDL language [122]. Furthermore, the type is expanded to a vector of n such signals and so the cal_logic_vector type is created. As shown in Source 3.4.1, the VHDL definition for the cal_logic type consists of eight characters:

```

type cal_logic is ( 'U',           -- Uninitialized
                   'X',           -- Forcing Unknown
                   'l',           -- 0 type phi0
                   'h',           -- 1 type phi0
                   'L',           -- 0 type phi1
                   'H',           -- 1 type phi1
                   'Z',           -- High Impedance
                   '-');         -- Don't care

```

Source 3.4.1: cal_logic VHDL Definition

As described above, the definition and some basic conversion functions, e.g. from std_logic to cal_logic and vice versa, are part of the cal library. The definition of the data type is the starting point of the methodology to build logic devices with CAL. Furthermore, several logic functions have to be designed to support the simulation and the synthesis of CAL designs. These are special parts for the behavioral description of designs with CAL and therefore they are part of the cal_beh library.

Boolean functions: In order to build designs various functions have to be defined.

Such functions describe the relationship between the inputs and the output. They are also used by the synthesis to build, e.g., conditions of if-clauses. Considering a two-input AND-gate, the function between the two inputs and the output can be defined only, if the two input signals are in the same phase. So rule I and II

from Section 3.2 are ensured. Considering the condition in an if-clause again, it is not possible to use methods which use any kind of history or context. Therefore, it is not possible to retain the old state with simple functions, because they can purely derive the new value. Thus it must be ensured that these gates process only input signals that are in the same phase. This is done by inserting so called *stable-procedures* into the VHDL code.

Stable-procedure: In VHDL this procedure is inserted into the behavioral code to ensure that the VHDL-process continues only if all inputs of this *stable-procedure* are in the same phase. Thus, it enforces that rule III is fulfilled. The procedure is implemented with VHDL "wait until" statements to suspend the current process until the condition is met. Notice, that this function is only necessary in `cal_logic`.

Register and latches: One of the big differences between `cal_logic` and usual synchronous designs is the methodology by which storage elements are implemented. In synchronous designs this is usually done with clock edges. As shown on the left side in Source 3.4.2, the active clock edge is the point in time where the current value is accepted and frozen:

<pre> p2_SM : process (clk, reset) begin if reset = RES_ACT then Pc <= (others => '0'); elsif clk'event and clk = '1' then Pc <= PcNxt; end if; end process p2_SM; </pre>	<pre> p2_reg: cal_reg generic map (w => 108, reset_value => 01) port map (d => PcNxt, q => Pc, c_done => c_done, pass => pass, reset => reset); </pre>
--	---

Source 3.4.2: Register Implementation in `std_logic` and CAL

The right side of the source code shows the register implemented in CAL. Both implementations utilize an input signal (`PcNxt`), an output signal (`PC`), a reset signal, and the value which should be used after the reset. In the synchronous approach `others => '0'` is used to specify the value after reset. CAL `reset_value => 01` is used as a generic map. The big difference is given when the register stores the data. The big difference between the synchronous approach and a CAL implementation is the way the register stores the data. While in synchronous versions the rising clock edge is used, in CAL implementations a handshake protocol is employed.

Conversion functions: A CAL design should be able to interact with "normal" `std_logic` circuits as well as with the environment. For this purpose a set of conversion functions is needed. In the case of `cal_logic` the transformations are done by simple translation tables.

The issues above have been described in detail for the behavioral description, because they constitute the main differences between the synchronous design and the CAL logic design. The process of transforming a regular synchronous design to CAL starts with renaming the data types from `std_logic` to `cal_logic`, followed by inserting the *stable*-procedure to ensure rule III. Furthermore, the registers must be converted from the "if `clk'event`" style to the instances of the `cal-register` and the required acknowledge signals have to be generated. To interact with the environment, the appropriate conversion functions must be applied.

3.4.2 Functional Description – `cal_rail_logic`

Table 3.3 shows the `cal_rail_logic` type consisting of two rails of `std_logic` type. The two rails are bound together and have one name.

logic state	code φ_0	code φ_1
"LOW"	(0,0)	(0,1)
"HIGH"	(1,1)	(1,0)

```

type cal_rail_logic is
  record
    line1 : std_logic;
    line0 : std_logic;
  end record;

```

Table 3.3: `cal_rail_logic` Coding Scheme and the VHDL Definition

Boolean functions: All logic functions are available as pre-synthesized elements. So only existing functions are used and the design consists of instances of them. In `cal_rail_logic` the functions `AND`, `OR`, and `INV` are defined and all other logic functions are derived from them. Notice, that here the rules I – III are fulfilled inherently by the gates because each of them comprises a kind of hysteresis or a memory element as explained later.

Special gates: For the synthesis of CAL a set of specialized gates is needed. For example, the φ -detector or the components of the `cal-register` are some of them. The gates and their functionality are defined and so they are available for the rest of the design flow.

Conversion functions: The transformation from `cal_rail_logic` to `std_logic` is quite easy, because rail `a` in CAL directly represents the signal state in Boolean logic. Hence, in the inverse case only minor coding effort is required to add the adequate phase to the conventional Boolean signal.

The implementation of some selected gates is presented in the next chapter.

3.5 Basic Gates

To illustrate how logic functions can be implemented in CAL we discuss the example of a 2-input AND in this section. The derivation of the required functions is quite straightforward, and essentially the same is true for other basic functions such as OR, NAND, NOR and XOR.

3.5.1 AND Gate

Table 3.4 shows the truth table on signal level:

Z		E1			
		h	l	H	L
E2	h	h	l	hold	hold
	l	l	l	hold	hold
	H	hold	hold	H	L
	L	hold	hold	L	L

Table 3.4: Truth Table of a 2-input AND in CAL

For inputs that are within the same phase the respective AND function is simply applied and the output is represented in the same phase. For inputs in different phases the last valid output is retained ("hold"). On rail level this truth table has to be expanded to two rails per signal, yielding one separate truth table for each rail of the output, Z_a and Z_b , each with four input rails as shown in Figure 3.5:

The resulting circuit for one AND-gate consists of two RS-FF's – one for each rail (a and b) of the output signal Z. Furthermore, for each of the RS-FF's logic functions are used to derive the correct set and reset action. This results in the need of four 4-input and 1-output functional blocks for set and reset: R_a , S_a , R_b , and S_b .

The initial hardware implementation requires six logic elements (LEs) for one CAL-AND-gate. In comparison with a standard AND the gate count increases significantly, but it should be considered, that we are mapping the design to a standard FPGA library that has not been specifically optimized for CAL.

3.5.2 Phase Detector

Considering that there are two possible phases for each signal which is used to associate a bit to a data word, there is the need to detect the phase of a signal. This is very simple for a single signal: Both rails have to be combined with an XOR and the result is the phase – 0 for the phase φ_0 and 1 for φ_1 . As shown in Figure 3.6(b) this scheme can be expanded to an n-bit wide bus: The rails of each single signal are combined with an XOR-gate and the n results are tied together with an and-gate ("all-ones detector") and an or-gate ("all-zero" detector). The RS-Flip-Flop ensures that the output only

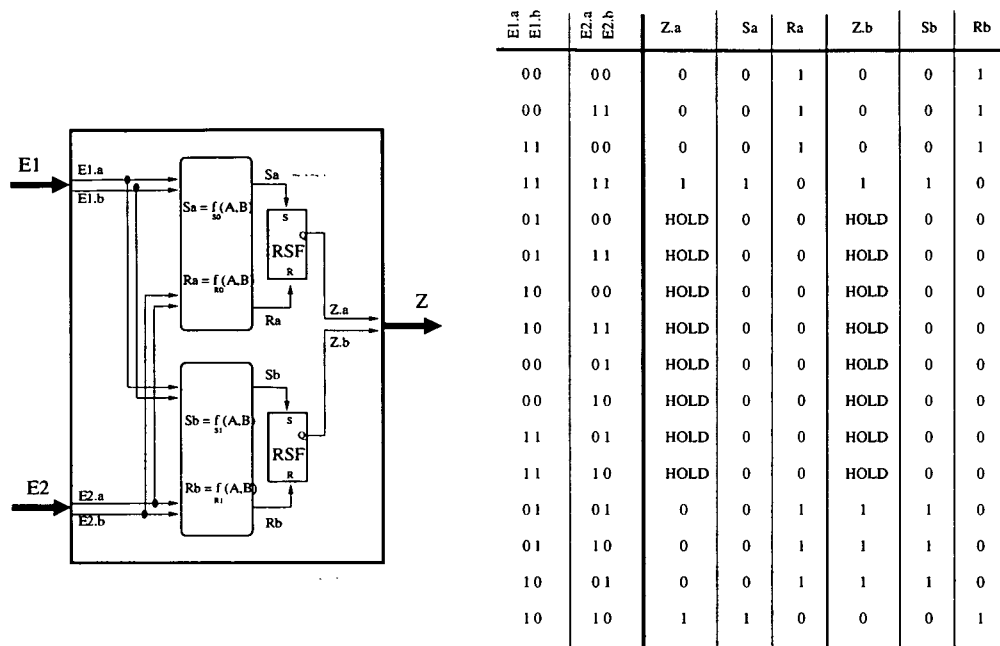


Figure 3.5: Schematic and Truth Table of the AND-gate

changes if all inputs are in the same phase as demanded by rule III. This circuit acts as a multi-input Muller-C gate.

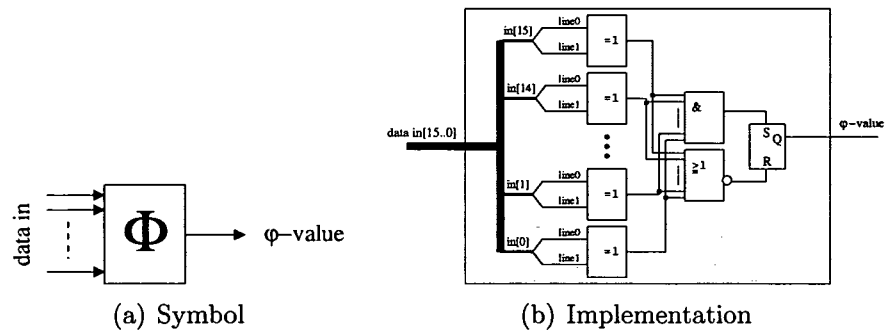


Figure 3.6: The φ -Detector

Notice that the φ -detector can also be used for completion detection, because the value at the output changes only if *all* input values are in the same phase. This is necessary, e.g., for register implementation used in pipeline structures.

3.5.3 φ -Converter

Sometimes it is necessary to convert the phase of a signal. Remember the pipeline of Figure 3.3 and consider the case that the signals from the first and second stage

are both inputs of the same gate. So the values should be used although they are in different phases. In this case a φ -converter is used to convert the phase of one of the signals so that they can be combined. Fortunately, the implementation is very simple (see Figure 3.7:

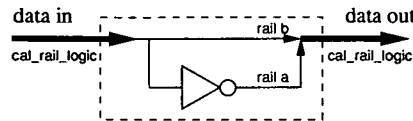


Figure 3.7: Implementation of a φ -Converter

Due to the fact, that only one bit changes by the transition from one phase to the other, the delay of this one inversion cannot cause an invalid output as a result of skew. If rail a changes due to the phase change, the result will be delayed. When rail b changes there is no impact on the circuit caused by the φ -converter.

3.5.4 CAL Register

The implementation of the registers used in the pipeline structure introduced in Section 3.3 is now discussed in detail. In Figure 3.8 the proposed implementation of such a register is shown. The chosen implementation represents a hybrid solution (see Section 2.3.3) to solve the fundamental design problem. As described further the solutions in high abstraction levels are done in the information domain and on this level there are no requirements on the design in terms of delay and skew. The claimed timing assumption on gate level must be met inside one register. If we can guarantee these requirements on this local area, the registers can be used without paying attention on their timing.

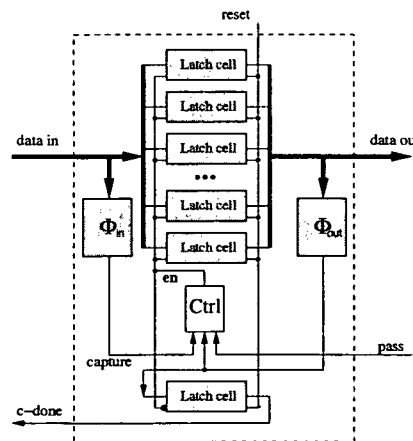


Figure 3.8: Implementation of a CAL Register

An explanation of the basic function of a register in a pipeline is given in Section 3.3. Remember that the latches get transparent if (i) the phase at the input differs from

the phase at the output and (ii) the phase of the downstream stage is equal to the phase stored in the latches (i.e. visible at the output). Condition (i) can be checked by comparing the outputs of the φ -detector, both at the input (Φ_{in}) and at the output (Φ_{out}). Condition (ii) implies that the pass signal from the downstream stage equals the output of (Φ_{out}). The Latches will lock if the phase on the output is equal with that on the input ($\Phi_{in} = \Phi_{out}$). The reset signal is used to initialize the latches with a predefined state.

There is one important detail. The capture done signal is generated by a latch with an inverse `en` input. This ensures that the `c-done` signal is not passed on to the upstream stage before the latches have actually stored their values. This works under the following timing assumptions:

- All latches must have the same *gate delay*. This can be ensured by taking all latches from the same library, so that they are built equally.
- The `en`-signal for the latches inside the registers must hold the isochronic fork assumption, this can be achieved by a well known routing process.

Further discussions on this topic will be given in Chapter 5.

3.6 CAL Design-Flow

So far we have described the basic gates. However, now we need a methodology to build hardware from a description of the design. Similar to the synchronous case there should be a behavioral description as a starting point. If the description meets the specification, it acts as the input of a tool chain which generates the associated hardware.

Therefore, as outlined in Section 3.4.2 we have defined a type to describe each signal with a single-rail 4-value data type called `cal_logic`. The basic boolean functions for this type are kept in a library. Thus, the simulation of the design on behavioral level is supported. At this state the design is described with `cal_rail_logic`. Recall that the data type used in this description consists of two rails of conventional 2-value `std_logic` signals. The required steps for generating a design in `std_logic` vectors on which standard place & route tools can be applied are described in Figure 3.9.

The difference between the conventional design flow and the approach used with CAL logic is clearly visible: Both approaches start with a behavioral description and the result of each of them is a description understood by the place & route tool. This final description is forced to use only gates of the target library, e.g. Altera APEX (see Section 4.2.1) in our case. After performing this last step the design can be downloaded to the FPGA.

In the conventional case the VHDL-code is elaborated and transformed into an intermediate language used by the synthesis tool. This functional description is the starting point for the synthesis during which the design is finally mapped to gates of the target

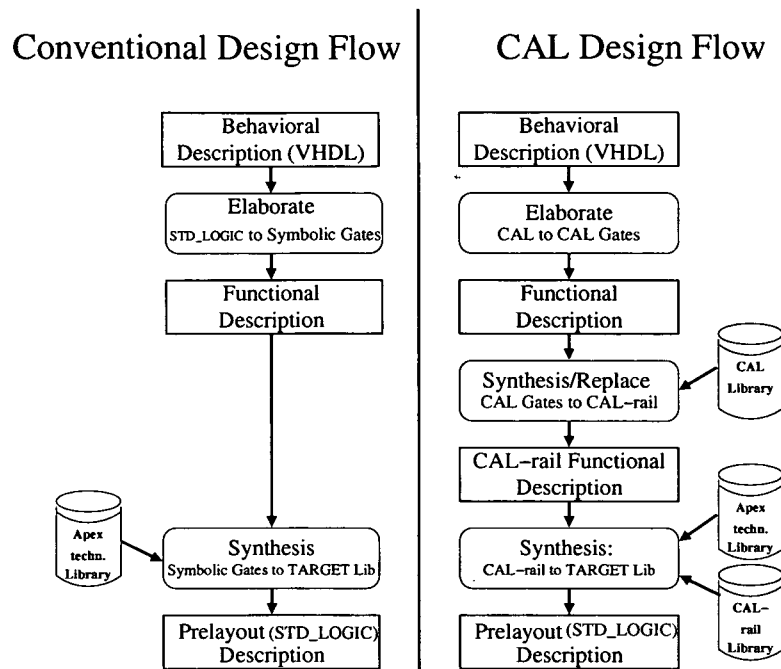


Figure 3.9: CAL-Design Flow

library. In our case this is the APEX-library. As a result we get the pre-layout representation of our design. This file is used for simulation on the one hand and as input for Quartus to perform place & route and the download to the FPGA on the other hand.

The result of the elaboration step performed in the CAL design flow is the functional description where the design is built with CAL gates. The functionality of the CAL gates is described in a special library (CAL-beh) to facilitate the simulation, which is described in detail in Section 3.7. For the synthesis another library is needed, which provides synthetic operators in order to build design specific gates. One of this operators is used to build a φ -detector with a width enforced by the design. So with the first synthesis the design is transformed from the four-value `cal_logic`-description to the dual-rail `cal_rail_logic`. As described above this representation uses pairs of `std_logic` signals and the functionality of the gates is provided by the `cal_rail`-library. This representation is used for simulation purposes as well as input for the second synthesis, which is very similar to the synthesis in the synchronous case. The APEX technology library is used as target library which results in a design constructed with APEX-gates.

It is important to note that the design flow allows us changing the actual type of pipeline register used for synthesis quite easily, because the functionality is added by the appropriate library. This led us to experiment with several implementation options that all turned out to have their particular benefits and drawbacks. A discussion of these different options will be the focus of Section 7.

3.7 Simulation Concept

In this section the simulation of a CAL design is discussed. At the beginning the four simulation steps are defined as follows (confer Figure 3.9):

behavioral simulation That means it is a simulation with `cal_logic` signals and any timing information of the resulting hardware. The input is the source code of the designer without any synthesis applied.

functional simulation The first synthesis has already transformed the `cal_logic` code into the `cal_rail_logic` format. However, no timing information has been added in this step.

pre-layout simulation The second synthesis has mapped the circuit to the target library – in our case the Altera APEX library. The logical function units are constructed with gates of the target library and so the number of gates and their standard delay is known and used for the simulation. However, place & route has not been performed. Therefore, for the delay of the wire default values have to be used.

post-layout simulation This representation contains the whole timing information of the design. Every gate as well as each wire delay is known and used for simulation. This leads to a high complexity of the simulation and consequently to a very long simulation duration.

The motivation for a sophisticated simulation method is clear and it is based on the design flow: The data types of the signals change with every step towards the real hardware. Nevertheless, it should be possible to use the same testbench for all four simulation levels. As described earlier in this chapter the starting point of the designs is the behavioral style – in our case `cal_logic`. The signals in the design as well as the ports are `cal_logic`. By the next step these types are transformed to `cal_rail_logic`. Therefore, the ports are also translated and the signals – using the same names as before – are now composed of `cal_rail_logic`.

The two rails are combined with the specific type to one record. After place & route the design consists of `std_logic` signals and so the ports are converted once again. Furthermore, the number of ports doubles with the last step and so each signal becomes a vector of two `std_logic` rails. In the same way the width of each vector doubles.

Although with each step the level of detail and therefore the refinement of delay increases, these three formats still represent the same design with the same functionality. The testbench is also written by the designer and therefore the `cal_logic` style is used. As shown in Figure 3.10(a) it is straightforward to perform the first simulation – the behavioral simulation – because the types of the ports match with the signal types of the testbench.

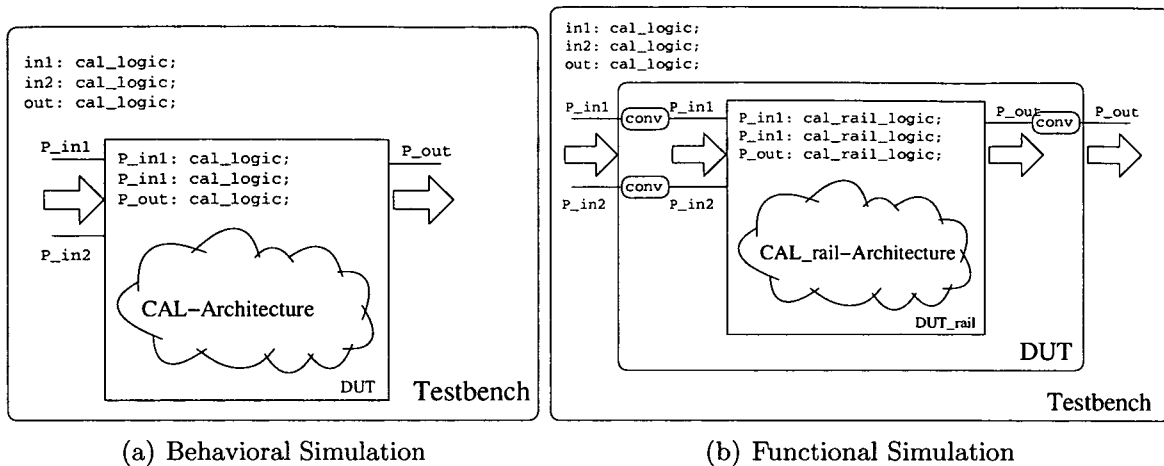


Figure 3.10: Simulation Concept

The next simulation steps cannot be performed so easily. In these cases the types of the ports of the device under test (DUT) are not equal to those of the testbench. Conversion functions have to be inserted to connect the DUT to the signals of the testbench. As shown in Figure 3.10(b) this is done automatically by our tool: To be able to simulate a design a configuration is used anyway to select and combine the architecture for a specific entity. A new architecture DUT is automatically created in which the original design DUT_rail is instanced. The architecture itself consists of just this instance and the appropriate conversion functions. While the designer has to write the testbench and the first configuration as in the synchronous case, the CAL specific parts are generated automatically.

In Figure 3.11 a post-layout simulation example is depicted that shows the value of the program counter and the output of a ROM:

In the first two lines the `pass` and `capture_done` signals are shown, followed by the address and the instruction. In this example only the lowest four bits are displayed. In line three and four the signals are shown in `cal_logic` style as they can be seen at every level of simulation. This is followed by the signals without being mapped to `cal_logic`. Every vector consists of eight `std_logic` signals, those of address are shown in detail. As depicted in Figure 3.11, it is very difficult to derive the value of the busses from the `std_logic` description: The address is incremented by one every step and the instruction remains zero.

This strategy has finally allowed us reaching the intended goal mentioned at the beginning of this chapter – one testbench for all simulations. The same procedures can be used to automate the verification process: The behavioral simulation must be checked by the designer manually whether the specification is met or not. If the required functionality is fulfilled, the remaining simulation steps are performed automatically by the tools and the results can be crosschecked with those from the behavioral simulation.

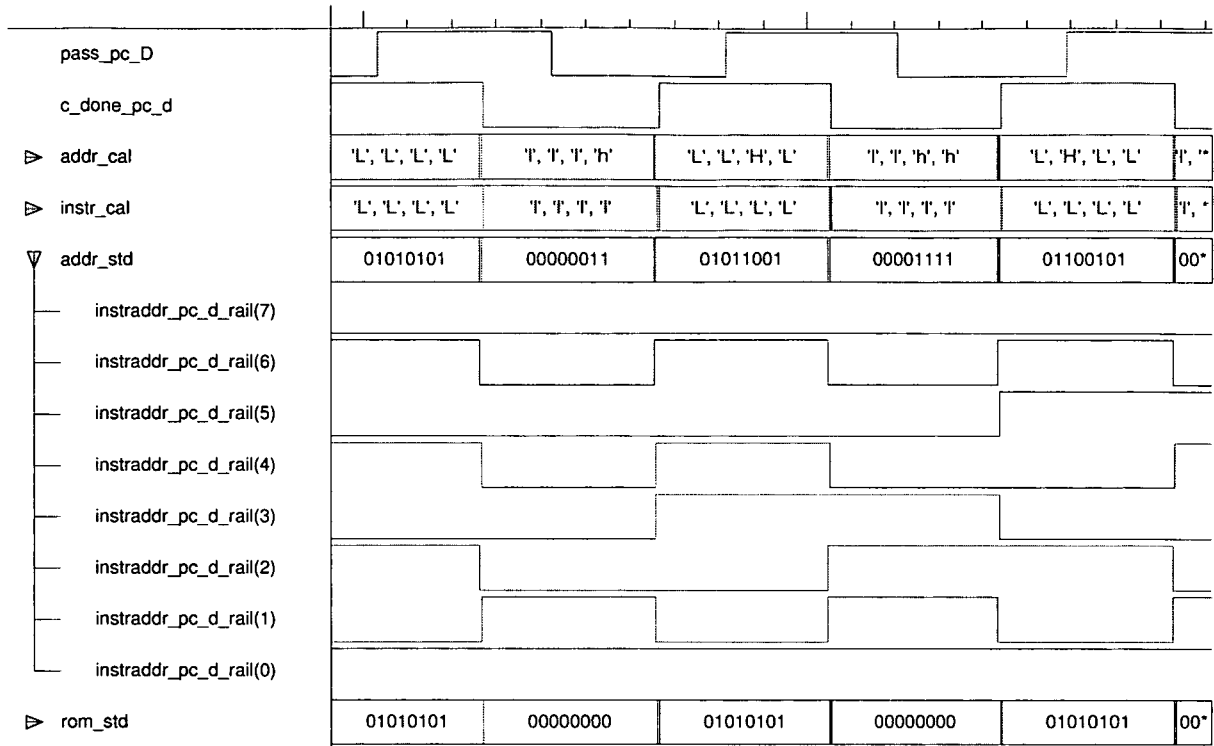


Figure 3.11: Post-layout Simulation Example

3.8 Summary

CAL is a design technique using signal coding. A dense code is utilized where two representations of each logic value "LOW" and "HIGH" – one for φ_0 and one for φ_1 – are given. Our approach is similar to NCL with some important advantages: There is no need for the so-called spacer or NULL-waves in CAL which doubles the throughput compared with NCL. Furthermore, the energy overhead in terms of transitions per bit is low: Exactly one rail transition per bit is required.

CAL is classified as hybrid solution to manage the fundamental design problem. A design built purely with CAL-gates is delay insensitive and so validity and consistency are needed to tackle the problem in the information domain. For the basic gates internal delay assumptions are made, yielding to design constraints. This is the part of the fundamental design problem solved in the time domain. The implementation of basic gates is demonstrated on appropriate candidates: The internal structure of an AND-gate as well as of a complex CAL register is described in detail.

The human interface to build CAL circuits – `cal_logic` – and the coding style on gate level – `cal_rail_logic` – are introduced. Furthermore, the methodology and the used libraries for the CAL design flow demonstrate the automated way from the design written by the engineer to the download file for a particular target architecture. This and the simulation concept show the practical applicability of our CAL approach.

Chapter 4

Prototyping Environment

Contents

4.1	The SPEAR Processor	56
4.1.1	Core Architecture	56
4.1.2	Extension Modules	57
4.1.3	Implementation Results	58
4.2	The Hardware Platform	58
4.2.1	APEX FPGA Family	59
4.2.2	Limitations	60

In this chapter the environment for the evaluation is presented: The synchronous reference design is shown, which is the starting point of our asynchronous implementation. The motivation to build a processor ourselves was the possibility to have a deep knowledge of design details, because it is very hard to derive the internal functionality from a standard microprocessor – like an ARM. Furthermore, the dependencies between the control signal among pipeline stages are very hard to explore, which is, however, one of the key points of our design. To avoid such troubles we decided to build our own processor – SPEAR.

The target platform for the design is an FPGA evaluation board. In the following a look at the underlying concepts of SPEAR and the evaluation boards is given and the advantages and drawbacks of the FPGA implementation are discussed.

4.1 The SPEAR Processor

4.1.1 Core Architecture

SPEAR is the acronym for "Scalable Processor for Embedded Applications in Real-time environments" [21] and the main goal of several design decisions [24] was to build a processor which has a well known temporal behavior [23]. The processor executes *every* instruction in exactly one cycle and also all instructions are one word wide. The SPEAR design has been developed to provide moderate computational power and represents a RISC architecture, which executes instructions through a three-stage-deep pipeline. The instruction set comprises 80 instructions, further a compiler suite [54] comprising the GCC [97] and the LCC has been developed supporting this instruction set. Most of these instructions are implemented as *conditional instructions* [98], which means that an instruction is executed or replaced by a NOP depending on the condition flag. A preceding test instruction sets this flag once and it is valid until the next test instruction. For example, a move instruction with condition false is executed when the result of the test instruction is false.

Instruction and data memory are both 4 kB in size, but it is possible to add up to 128 kB of external instruction memory and 127 kB of additional data memory. The uppermost 1 kB of the data memory is reserved for memory mapping of the extension modules. These modules (see Section 4.1.2) are used to customize SPEAR to the needs of the environmental interaction. As a result of the memory mapping, no dedicated instructions for extension module access are needed – common load/store instructions are used – which satisfies the RISC [47] philosophy of our design approach. The register file holds 32 registers which are split up into 26 general purpose and 6 special function registers, three of them are used to construct stacks efficiently using frame pointer operations. The remaining three special function registers are used to save the return address in case of an interrupt or subroutine call. SPEAR supports 32 exceptions, 16 of them are hardware exceptions – interrupts – and 16 can be activated by software, we call them traps. The entries of the exception vector table hold the

corresponding jump addresses to the interrupt/exception service routines for each interrupt or exception. The SPEAR ALU performs all provided arithmetic and logical functions, but it is also responsible for offset calculation on jumps. Furthermore, the ALU is used to pass through data from the exception vector table or register file. Figure 4.1 shows a block diagram of the SPEAR processor.

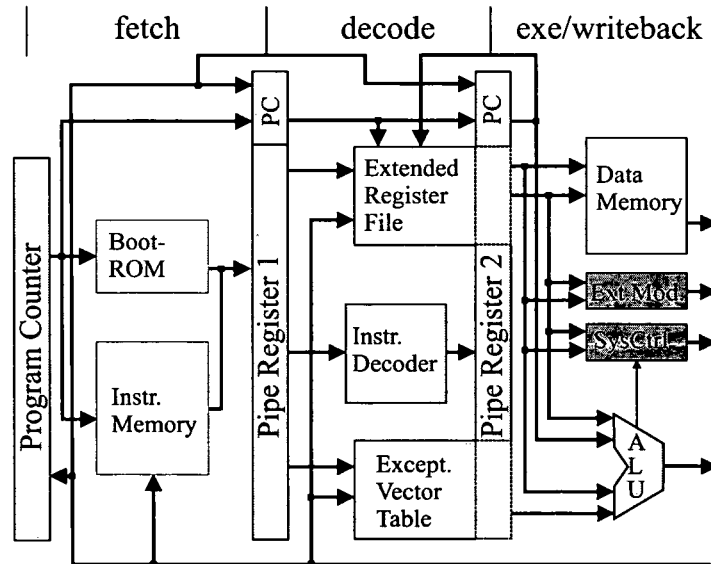


Figure 4.1: SPEAR Architecture

The SPEAR pipeline is structured into an instruction fetch (FE), an instruction decode (DE) and a combined execute/write-back (EX/WB) stage. In the fetch cycle, the instruction memory is accessed and one instruction opcode is passed to the decode stage. During the decode cycle the control signals for the memories and the ALU are generated, furthermore the operands of the actual instruction are retrieved from the register file. The execute/write-back stage performs the intended operation of the instruction and writes the resulting value to the appropriate memory location. When an extension module access (EXT) happens, it is also executed during the EX/WB cycle.

4.1.2 Extension Modules

As mentioned above, extension modules are used to fit the processor for different applications. For reasons of simplicity and lucidity, the integration of and the access to the extension modules should be normalized. Thus, a generic interface for all extension modules has been defined [50]. All extension modules are mapped to a unique location at the uppermost region of the data memory. The modules are accessed via eight registers using simple load and store instructions, as from the processor's point of view the extension modules are simply memory locations. A block diagram of the

generic extension module interface is shown in Figure 4.2. The first two registers are the *status* and *config* register of the module. The *status* register tells the processor the current state of the extension module. Among other things it shows if an interrupt has been activated, an error has occurred, or if the extension module is still busy. The *config* register is used to specify parameters for the operations of the module. Next to a soft-reset bit, which is used to deactivate the extension module, an interrupt acknowledge bit exists to reset the interrupt status. The remaining six registers Data 0 – Data 5 are available for module specific issues.

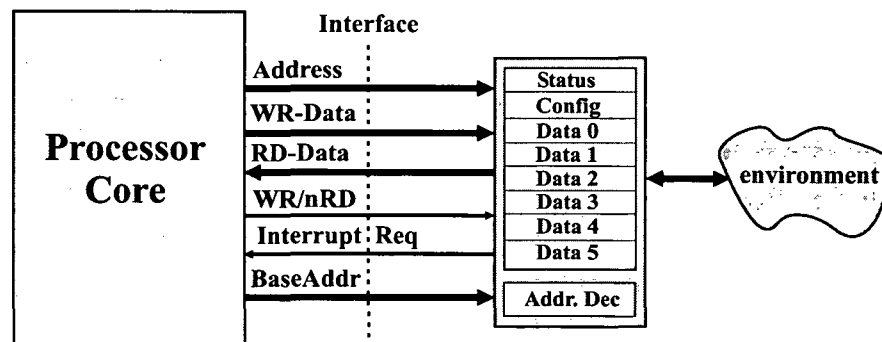


Figure 4.2: Generic Extension Module Interface

There is a special extension module – the processor control unit which has to be used in every design. It comprises functional blocks which are essential for the processor, e. g. the processor status word.

4.1.3 Implementation Results

Some implementation details are given here to finish the description of the synchronous reference design: Our processor SPEAR utilizes 1,794 logic elements of the APEX20KC FPGA (see 4.2.1). This is about 15 % of the total number of logic elements. Further, the on-board data and instruction memories as well as the register file use more than 70,000 memory bits - which is about 47 % of the number available. Finally, SPEAR runs with a maximum clock frequency of 46 MHz on this FPGA.

4.2 The Hardware Platform

The target technology for the synthesis and the following place & route steps are FPGAs¹. The decision to build hardware on FPGAs instead of using full- or semi-custom ASIC-chips is based on the fact, that it is much faster and much cheaper to develop a prototype. The SPEAR processor as well as the asynchronous designs should

¹We use the term FPGA for off-the-shelf components. However, there are some approaches for bundled-data systems STACC[94], PGA-STC[67] and one for general purpose architectures – Montage[46].

be tested as a physical implementation to prove the functionality, e.g. by displaying several buses on a logic analyzer. Modern FPGAs are nowadays quite fast and big enough to contain a processor design. Unfortunately, the use of FPGAs does not only cause advantages: The performance of a processor built with FPGA basic gates is not as high as it could be with an ASIC design, but designs should be proof-of-concept and therefore the performance is not the key achievement.

Our prototyping board called megAPEX[32] is built by *El Camino*[31] and it is equipped with an FPGA out of the APEX Family, which is described in detail in the following section.

4.2.1 APEX FPGA Family

An FPGA is an integrated circuit that consists of an array or a regular pattern of logic cells. The logic cells can be configured to represent a limited set of functions. These individual cells are connected by a matrix of programmable switches. The developer's design is implemented by specifying the logic function for each cell and selectively closing switches in the interconnect matrix. The array of logic cells and the interconnect matrix are taken from a set of basic building blocks for logic circuits. These basic blocks are combined to achieve the intended behavior of more complex designs.

The logic cell architecture varies between different device families. In general, each logic cell combines a few binary inputs (typically between 3 and 10) to one or two outputs according to a boolean logic function specified in the programmed design. In most FPGA families, there exists the possibility of registering the combinational output of the cell, so that clocked logic (like counters or state-machines) can be implemented easily. The combinational logic of the cell can be physically implemented as a small look-up table (LUT) or as a set of multiplexors and gates.

The APEX family represents highly integrated FPGA devices which are manufactured in 0.22 μm to 0.15 μm processes. APEX devices are available in ranges from 30,000 to over 1.5 million gates. The APEX architecture consists of so-called MegaLABs[2]: These function blocks can be connected with each other as well as to I/O Pins. LUT-based logic provides optimized performance for data-path and register-intensive designs, whereas product-term-based logic is optimized for combinational paths, such as state machines. Embedded system blocks (ESB)[2] can implement a variety of memory functions, including first-in-first-out (FIFO) buffers, ROM or dual-port RAM functions. The ESBs support memory block sizes of 128x16, 256x8, 512x4, 1024x2 and 2048x1, but can be cascaded to implement larger sizes. The MegaLAB structure comprises a set of logic array blocks (LABs), one ESB, and a MegaLAB interconnect, which routes signals within the MegaLAB structure. The amount of LABs inside each MegaLAB depends on the specific APEX device, and can range from 10 to 24 LABs. Signal interconnections between MegaLABs and I/O pins are provided by the *FastTrack Interconnect*, a set of fast column and row channels (additionally LABs at the edge of MegaLABs can be driven by I/O pins via the local interconnect).

Each LAB consists of 10 logic elements (LE) and the associated local interconnect.

Signals are transferred between LEs in the same or adjacent LABs, ESBs or IOEs via high-speed local interconnects. The LAB-wide control signals can be generated from the LAB's local interconnect, global signals, or dedicated clock pins.

The logic element (LE), the smallest addressable logic unit in the APEX architecture, is very compact and provides efficient logic usage. Figure 4.3 shows a block diagram of an LE. Each logic element contains a four-input LUT, which is a function generator that is able to implement any function of four input variables. Furthermore, carry and cascade chains as well as a programmable register for D-, T-, JK-flip-flop and a shift register implementation are part of each LE. LEs can drive the local interconnect, the MegaLAB interconnect, and the FastTrack interconnect structures.

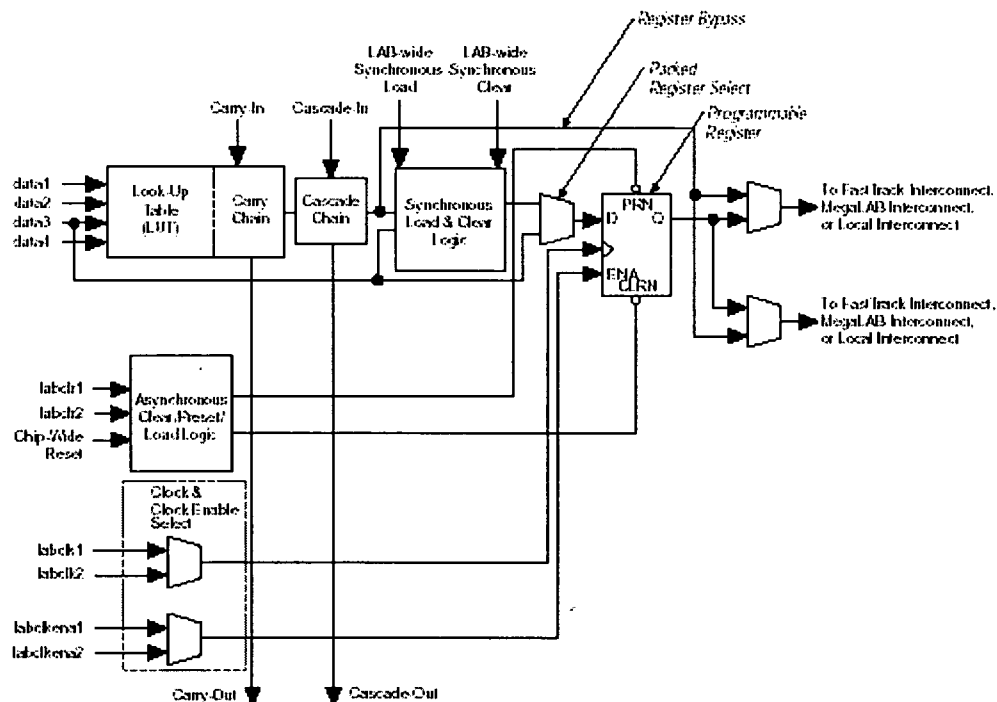


Figure 4.3: Logic Element Structure [2]

For our experiments we use 20KC1000 devices, that feature the 0.15 μm process and all-layer-copper interconnect. This FPGA is equipped with 38,400 Logic Elements – it is comparable to 1,000,000 typical gates. Further information can be found at [2].

4.2.2 Limitations

FPGAs are designed and optimized for synchronous designs and this clearly has an impact for the implementation of purely asynchronous circuits. Our experience with APEX devices lead to the following points:

Wire delay: As mentioned in the introduction the wire delay becomes more and more

important in the chip design. In ASICs this drawback can be tackled by optimizing the routing. In FPGAs, however, this is not possible, because the wires are built during the manufacture of the FPGA and only the interconnects are programmed by the design. This leads to longer wires and thus to a larger delay. It can be seen that the wire delay limits the performance in synchronous FPGA designs. The design of the super-scalar variant of the SPEAR namely LANCE shows this effect [38].

Logic elements (LEs): As shown in the section above, Altera FPGAs are composed of LEs. Four input signals can be combined to one output. This does not meet our requirements: In a CAL design each gate has a dual-rail output and in the case of feedbacks it has more than four inputs. If more than four inputs for one output are required, additional LEs have to be utilized and so the design grows very fast.

Synchronous register: Every LE is equipped with an edge-triggered register, which reflects the optimization for synchronous designs. In the case of CAL, however, they are useless.

RS-flipflops: In basic gates (see 3.5) an RS-flipflop is used as a memory cell to hold the old state of the output. Unfortunately, the APEX FPGA does not offer an RS-flipflop as a component in an LE. It must be built with an LE and an external feedback. This external feedback can lead to problematic race conditions with other signals.

Place & route tools: The tools for place & route as well as the timing analysis tools are also optimized for the use with synchronous designs. They are built to optimize the register to register delay. This leads to very long execution times for the tools as well as to not optimized results for asynchronous designs.

In summary, FPGAs are principally not intended and not well suited for asynchronous logic designs. Asynchronous designs implemented in FPGAs have many disadvantages compared to synchronous FPGA designs on the one hand and asynchronous ASICs on the other. However, we found the reconfigurability of the FPGA platform worth the price and as shown later, we have built an asynchronous version of SPEAR on an FPGA.

Chapter 5

Delay-Insensitivity of Circuits Built with CAL

Contents

5.1	The Uppaal Tool Suite	63
5.2	Delay Insensitivity Analysis of CAL-Registers and the Pipeline Structure	65
5.2.1	Schematic Pipeline	65
5.2.2	Pipeline Implementation	69
5.2.3	Pipeline Model with Synchronized Capture Done	73
5.2.4	Pipeline Implementation with Latched Capture Done	74
5.3	The Combinational Functions ($f(x)$)	78
5.3.1	Circuits built with CAL-Gates	78
5.4	Summary	81

In context with Chapter 6, this chapter treats the delay insensitive properties of circuits built with CAL. This is performed by analyzing the structure of the circuits as well as by analyzing the hardware implementation.

Simulations can be used to detect problems within the design, but it is very hard to argue that there are no further deficiencies with delays or anything similar to it. To obtain more complete and meaningful results the verifications should be done with a *model checker* using an appropriate model of the hardware implementation. We have chosen the *Uppaal* system[116], which uses timed automata to describe the systems processed by the checker.

The results derived by methods of formal verification and their correctness depend on the underlying models. So one of the main parts in this section will be the confirmation of the used model. Therefore, the problem is decomposed into small and manageable pieces and the models, which are used for these small subsystems are motivated. The focus of this chapter is a high level investigation of CAL: Registers and the combinational logic functions between those registers are analyzed under the assumption that they are built with CAL basic gates. These gates are examined in Chapter 6.

5.1 The Uppaal Tool Suite

Uppaal is a tool suite for automatic verification of safety and bounded liveness properties of real-time systems modeled as networks of timed automata[4]. Uppaal has been developed by the Department of Computer Systems, Uppsala University, Sweden and the Basic Research in Computer Science, Aalborg University. In this section a brief introduction on the usage and the basic types of Uppaal is given.

The template shown in Figure 5.1 is used to explain the parts used by Uppaal. Templates and processes are used to build the Uppaal system where these processes are processed concurrently. A process is an instantiated template, whereat the parameters are defined. However, if a template has no parameter, it can be used directly and no instantiation is necessary. The template in Figure 5.1 shows four so-called *locations* (start, step1, step2, and last) which are connected by *transitions*.

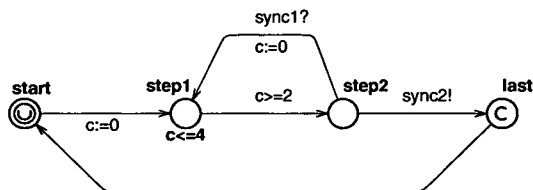


Figure 5.1: Uppaal Template Example: P

Location *step1* is equipped with an *invariant* ($c \leq 4$), which is a progress condition. The system is not allowed to stay in the state more than four time units, so that

the transition has to be taken. If no enabled transition is available, a deadlock occurs. Invariants are used to ensure progress. A location has optionally the following properties:

Initial A location with this property denotes the starting location of a process. Every template must have exactly one location with this property. The initial location is marked with an additional circle as depicted by location `start` in Figure 5.1

Urgent An urgent location is equivalent to a location with incoming edges resetting a designated clock c and labeled with the invariant $c \leq 0$. Time may not progress in an urgent state, but interleaving with normal states is allowed. Normal states are those which have neither the urgent nor the committed property set.

Committed A committed location (e.g. `last` in Figure 5.1) is more restrictive than the urgent location: Consider a system comprising several processes. In all states where a committed location is active, the only possible transition is the one that fires the edge outgoing from this location. No delay is allowed and so the committed location must be left in one of the successor states (in our example there is only one).

Transitions can have expressions which must be true (so-called *guards*) to be enabled. In Figure 5.1 such a guard ($c \geq 2$) can be viewed at the transition from `step1` to `step2`. Furthermore, some activities can be performed if a transition is taken. For example, such an *assignment* ($c := 0$) is shown at the transition from location `start` to `step1`. Binary synchronization channels `sync1` and `sync2` are utilized to synchronize two edges. Consequently, these features can be summarized as follows [6]:

Guard A guard is an expression which must be satisfied to enable a transition.

Synchronization A binary synchronization is a pair consisting of `sync!` and `sync?`. One transition `sync?` must be enabled so that `sync!` can fire. These transitions are taken concurrently.

Assignment An assignment label is a comma separated list of expressions with side-effects.

A template can be instanced several times, e.g. our sample template shown in Figure 5.1 can be applied twice with $p1 := P(\text{do1}, \text{do2})$; $p2 := P(\text{do2}, \text{do1})$; with changed synchronizers as parameters to synchronize each other (see Figure 5.1).

The interested reader can find additional information and the underlying concepts in [60, 7, 61, 15, 6].

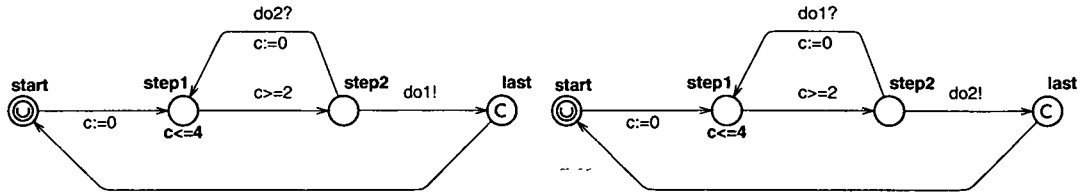


Figure 5.2: Uppaal Template Example: Two Instances of P: p1 and p2

5.2 Delay Insensitivity Analysis of CAL-Registers and the Pipeline Structure

5.2.1 Schematic Pipeline

As a starting point for modeling the schematic pipeline structure from Section 3.3 is recalled (see Figure 5.3). This simple structure of a pipeline is used to show the transformation to a timed automaton in detail.

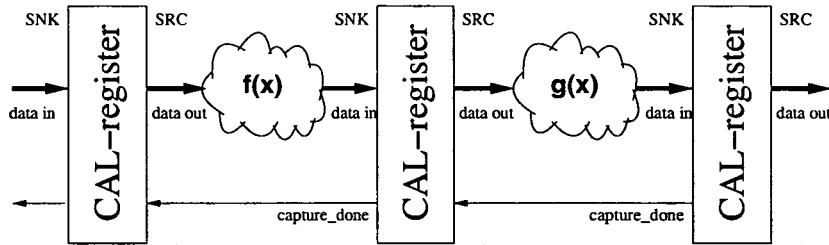


Figure 5.3: Schematic CAL Pipeline Structure

We want to prove whether the principle function of a pipeline is delay insensitive or if delay assumptions must be made. Recall the three rules of Chapter 3.2: Rule I claims that there is an alternation of phases between two data waves. In our model we have to build start and end-models which stimulate the pipeline with the needed control signal. These models define alternating input phases. If the pipeline mixed up the phases, there would be a deadlock, because the steering modules at the beginning and at the end of the pipeline will point out any minor error. Further, the correct function of the pipeline can be checked by observing a register while it is transparent. The latches in a register become transparent, if the phase of the input equals the expected phase of the new input phase. Thus, while the latches are transparent, this phase must not change. In this simple model of the pipeline we have not defined an intermediate state where some inputs are in phase φ_0 and some in φ_1 , so Rule II+III are fulfilled automatically. Source 5.2.1 shows these claims translated to Uppaal-queries¹. The second one is an example for a query checking one pipeline register, e.g. p1 is used:

¹ $\forall \square$ is formed as A[] with ASCII-codes.

```
A[] not deadlock
A[] p1.data_trans imply not p1.p_data_in != p1.value
```

Source 5.2.1: Uppaal-Query for the Schematic Pipeline

To use these queries the pipeline must be represented in a model, which can be used by Uppaal. As a first approach the representation of the signal is reduced to the phase of the information. To show the sequences of a pipeline it makes no difference whether the logic value of a signal is "HIGH" or "LOW". However, it makes the model simpler and more clear. Figure 5.4(a) shows the model of the CAL register:

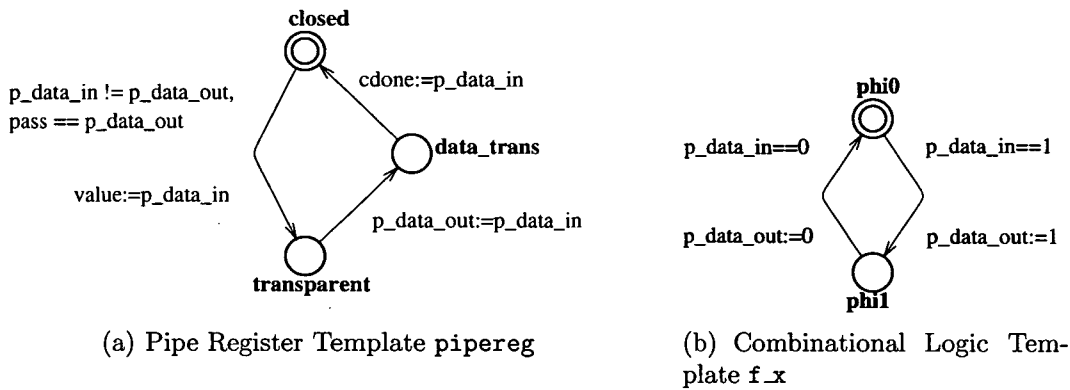


Figure 5.4: Schematic Pipeline

The functionality is described in three states: The register can be closed, transparent and in `data_trans`, which is utilized to describe that the data have reached the output. Guards use comparison expressions to ensure a fact (e.g. `p_data_in != p_data_out` in Figure 5.4) in contrast to assignments, which assign values (`value:=p_data_in`).

The pipe register starts in the state `closed` and the guard of the first transition models the rules from Section 3.3: The latches become transparent if (i) the phase at the input differs from the phase at the output and (ii) the phase of the downstream stage is the same as the phase stored. If the transition is taken, the phase on the input of the register is stored in variable `value`. The model resides in state `transparent` and after an arbitrary amount of time the value of the input is transferred to output; this is described by `p_data_out:= p_data_in` and the model is now in the state `data_trans`. The last action performed by the register is set to capture done.

The combinational logic ($f(x), g(x)$) is modeled by a template shown in Figure 5.4(b). It starts in state `phi0`² and the guard `p_data_in==1` ensures that after all signals of $f(x)$ are consistent in phase φ_0 the output of $f(x)$ changes to zero (representing φ_0).

²The model is symmetric, hence we might as well assume `phi1` as the starting point.

There are two things missing in order to be able to simulate the system: The beginning and the end of the pipeline must be controlled in the appropriate way. New data-waves have to be provided when the pipeline is ready – this is done by the start logic shown in Figure 5.5(a).

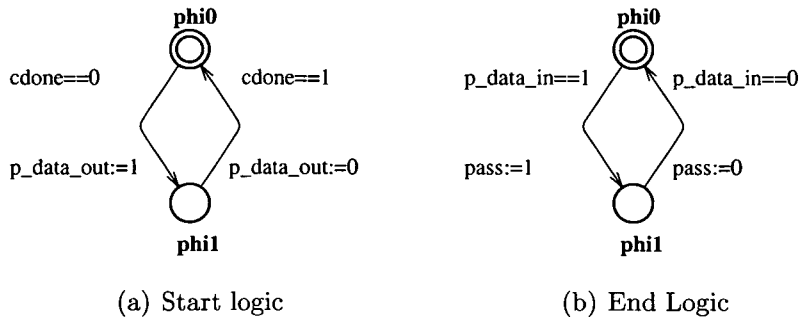


Figure 5.5: Completing the Schematic Pipeline System

Further, the data supplied by the last pipeline stage must be consumed in such a way that the pipeline is enabled to proceed with the next wave (see Figure 5.5(b)). To illustrate the design entry in this thesis, the definition of the system is shown in Source 5.2.2.

```

global definitions:
  const REG 3;
  int[0,1] passend,cdone[REG+1], data_in[REG+1], data_out[REG+1];
-----
process assignments:
  ps:=pipestart(data_in[1],cdone[1]);
  p1:=pipereg(data_in[1],data_out[1],cdone[1],cdone[2]);
  f_x1:=f_x(data_out[1],data_in[2]);
  p2:=pipereg(data_in[2],data_out[2],cdone[2],cdone[3]);
  f_x2:=f_x(data_out[2],data_in[3]);
  p3:=pipereg(data_in[3],data_out[3],cdone[3],passend);
  pe:=pipeend(data_out[3],passend);
-----
system definition:
  system ps,p1,f_x1,p2,f_x2,p3,pe;
    
```

Source 5.2.2: Definition of the Schematic Pipeline

In addition to the templates shown above, three sections must be defined in Uppaal: **global definitions** As implied by the name variables and constants visible to all templates can be defined in this section.

process assignments There are two possibilities for using a template. First, it can be a model without parameters and so it is used once as it is defined. Otherwise, templates can be provided with parameters, so it is possible to adapt them. In our case it makes sense to define a pipeline register once and then instance it several times. As shown in Source 5.2.2 the pipeline registers p1, p2 and p3 are processes of pipereg with the used data signals as parameters.

system definition Finally, using the system directive, Uppaal is told which templates or instances to use for simulation.

The whole model as defined above can be seen in Figure 5.6. This figure illustrates the simulation mode of Uppaal.

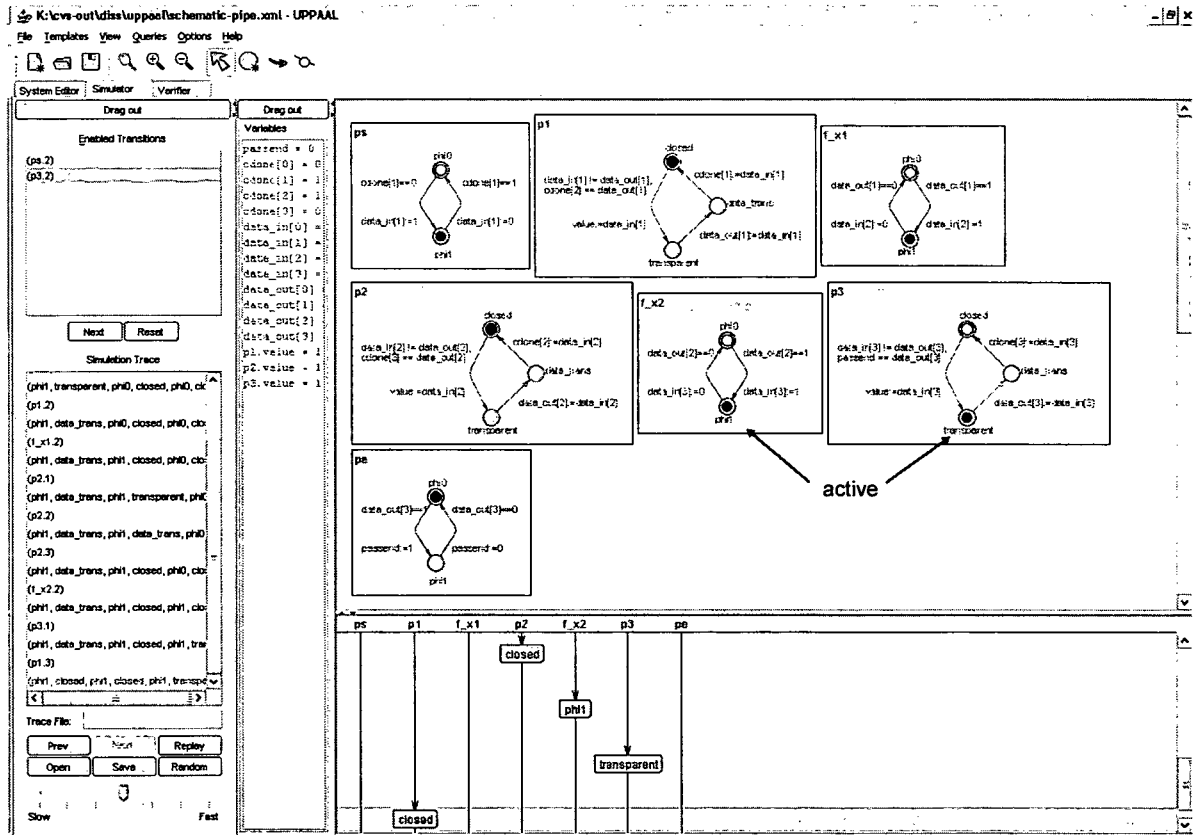


Figure 5.6: Uppaal Simulator (Schematic Pipeline System)

On the left side all enabled transitions for the next step as well as the trace of the simulation can be viewed. The main screen displays the configured processes where the currently used locations are marked with a red point (They are marked with "active" in Figure 5.6). Also, while running the simulation every active transition is colored. Finally, the bottom of the screen displays the transition sequence of the past simulation steps.

The queries³ defined in Source 5.2.1 have been checked with the verifier of Uppaal and the results are summarized in Table 5.1:

Nr.	Query	Uppaal result
1	$E\langle\rangle$ pe.phi1	is satisfied.
2	$E\langle\rangle$ pe.phi0	is satisfied.
3	$A\Box$ not deadlock	is satisfied.
4	$A\Box$ p1.data.trans imply not p1.p_data.in != p1.value	is satisfied.
5	$A\Box$ p2.data.trans imply not p2.p_data.in != p2.value	is satisfied.
6	$A\Box$ p3.data.trans imply not p3.p_data.in != p3.value	is satisfied.

Table 5.1: Results of Uppaal checking the Schematic Pipeline

Query 1+2: The output of the last pipeline stage can reach phase φ_1 or φ_0 respectively. This is just a functionality test to see whether the pipeline functions or not. The model of our pipeline works as shown by the result `is satisfied`.

Query 3: Start and end model of the linear pipeline enforce alternating data phases. As mentioned at the beginning of this section, an erroneous behavior will lead to a deadlock of the system. Uppaal has formally proven that there is no possibility of a deadlock.

Query 4 to 6: CAL-registers let exactly one wave pass while they are transparent. This can be shown for all three registers used in this system. This is a necessary condition for the correct CAL-functionality.

The results confirm that Rule II is held. As aforementioned, the main task of this simple example concerning the schematic pipeline structure is a brief introduction into the usage of Uppaal.

5.2.2 Pipeline Implementation

Now the first real implementation of a pipeline is under examination. The model consists of all parts of a real hardware implementation, but the data signal is simplified as described later. Figure 5.7 shows the proposed pipeline implementation.

The three rules of Section 3.2 are applied to the pipeline model. Rule I claims alternating phases of the data waves. As considered in the model above, this is once again checked with the deadlock property. Rule II requires that only valid signals are processed and in combination with rule III it requires that the output must remain in the current state if the phases of the inputs differ. In the case of our CAL-registers this means that the register must be closed, if the input signals are not in the same phase, the output will remain in the old state automatically.

³ $\exists\Diamond$ is formed as `E<>` with ASCII-codes.

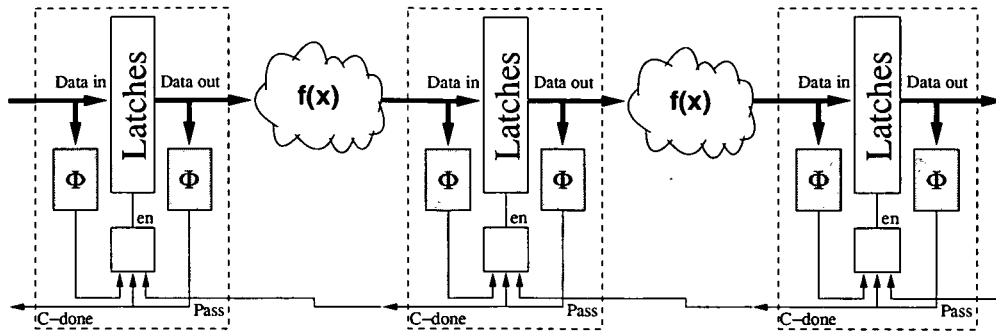


Figure 5.7: CAL Pipeline Structure

Therefore, the model used above is no longer suitable. There are more details that have to be considered in a realistic approach. The first important modification concerns the representation of the combinational logic – called $f(x)$. In the model used in Section 5.2.1 the data signal could only be in phase φ_0 or φ_1 . However, we have to take a closer look at the switching behavior from one complete data wave to the next one concerning signals with more than one bit. The procedure differs from that described above: Consider a bus with N signals and all of them are in phase φ_0 ("0" in the Uppaal model). To proceed to the next data-wave every member must switch to phase φ_1 ("1"), but this will not be performed at the same point in time for all members. Therefore, there will be some signals still remaining in phase "0" and others will already have switched to "1". This is not a problem in principle (recall Section 3.2). However, the circuit may only process data signals which are in the same phase. This must also be true for a CAL-register. This behavior of the pipeline is checked with the rules shown in Source 5.2.3. In particular this new aspect is verified by the second query:

```

A[] not deadlock
A[] f1_x.mixedphi imply not p2.transparent
A[] p2.data_trans imply not(data_in[2] != p2.value)
    
```

Source 5.2.3: Uppaal-Query for the First real Pipeline Implementation

The model from Figure 5.4(b) is modified. A third state denoted as `mixed_phi` is added to cover this case (see Figure 5.8(a)).

Furthermore, the start part of the pipeline is expanded with this `mixed_phi` state (see Figure 5.8(b)). Some additional function blocks have to be modeled to describe a pipeline as shown in Figure 5.7. First, the functionality of a φ -detector is transformed to an Uppaal model (see Figure 5.9(a)). The upper part of the model describes the activities that are performed if the inputs are in φ_0 . The transition from `mixed_phi` to `phi0` is enabled, if the input `data_in` has zero value. The part of the model to set the output of the φ -detector is divided into two locations. Thus, after the input of

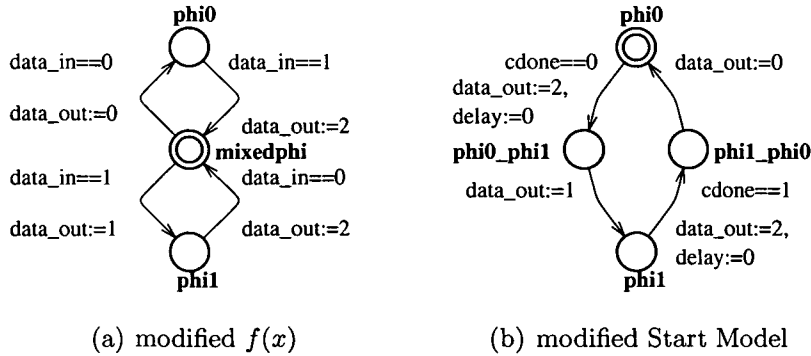


Figure 5.8: Components of Pipeline Implementation

the φ -detector is zero, the output will eventually (i.e. after some delay) go to zero. Afterwards, the automaton remains in location `phi0_out`, until the input differs from zero. Now the transition back to the `mixedphi` location is enabled. The lower part of the model is able to perform the same task for φ_1 .

The outputs of the φ -detectors located at the input and the output of a CAL register are used to calculate the enable-signal in the so-called enable logic, shown in Figure 5.9(b). In addition, the pass signal from the downstream stage is used for this calculation using the rules for a pipeline register given in Section 3.5.4. The globally defined `enable` variable is utilized to control the modified latch shown in Figure 5.9(c). In contrast to the model used before, the control logic is sourced out to the enable logic.

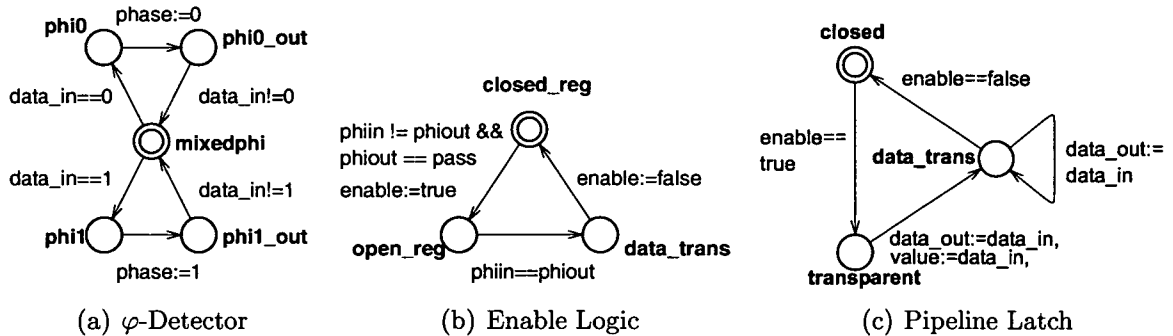


Figure 5.9: Pipeline Implementation

The queries defined in Source 5.2.3 are now applied to the model comprising two pipeline stages and a start and end logic using the templates defined above. The results of these verification runs are shown in Table 5.2:

Query 1-3: The output of the last pipeline stage can reach phase φ_1 . Further, pipeline stage 2 is able to switch to transparent mode and $f(x)$ can also switch to φ_1 . This functionality tests whether the pipeline is able to progress or not.

Nr.	Query	Uppaal result
1	E<> pe.p1	is satisfied.
2	E<> p2.data_trans	is satisfied.
3	E<> f1.x.phi1	is satisfied.
4	<i>A// not deadlock</i>	<i>is NOT satisfied.</i>
5	<i>A// f1.x.mixedphi imply not p2.transparent</i>	<i>is NOT satisfied.</i>
6	<i>A// ps.phi0_phi1 imply not p1.transparent</i>	<i>is NOT satisfied.</i>
7	<i>A// p2.data_trans imply not(data_in[2] != p2.value)</i>	<i>is NOT satisfied.</i>
8	<i>A// p1.data_trans imply not(data_in[1] != p1.value)</i>	<i>is NOT satisfied.</i>

Table 5.2: Results of Uppaal checking the Real Pipeline

Query 4: Start and end model of the linear pipeline enforce alternating data phases. The Uppaal runs, however, show that the system does not guarantee this behavior.

Query 5 to 8: Exactly one wave should be able to pass while one latch is transparent. Unfortunately, this is not guaranteed in this implementation⁴. This deficiency can be shown for both registers considered in this implementation.

To identify the problems recall Figure 5.7. Through the observation of values calculated by the φ -detector at the output and the pass signal of stage two, we may *conclude* that input has been captured by stage two. This is, however, not a safe conclusion with respect to delay insensitivity. In particular, the output of the φ -detector at the output of a stage forks (a) to initiate capturing for the considered stage one and (b) to switch the upstream pipeline register to transparent. If the capture path is slower, the pipeline will not work.

In order to improve the situation let us have a closer look at the fork mentioned above and the respective delays in the two concurrent paths (Figure 5.10):

1. Path A (plotted slash-dotted in Figure 5.10) : Capturing at stage two:

$$t_1 = t_{w1_2} + \delta_{enable} + t_{w2_2} \quad (5.1)$$

2. Path B: Switching stage one to transparent:

$$t_2 = t_{w1_1} + \delta_{register} + \delta_{f(x)} + t_{w3_1} \quad (5.2)$$

Since we want stage two to finish capturing before stage one passes the next data word, the condition for a correct control flow is

$$t_1 < t_2. \quad (5.3)$$

⁴This is illustrated by the italic font in Table 5.2.

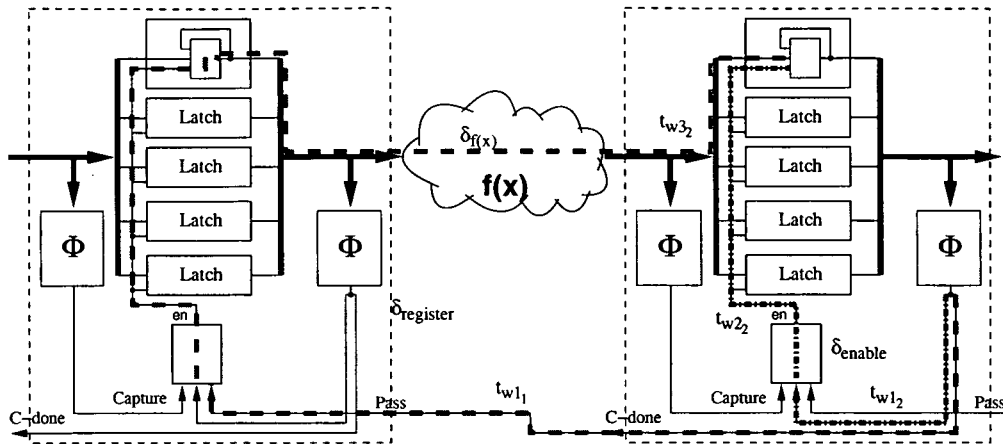


Figure 5.10: Critical Delay Paths of the Basic Pipeline Implementation

Let us check, under which condition this is true. To be on the safe side we assume the all delays with zero cannot be influenced on the current stage. This leads to

$$t_{w1_2} + \delta_{enable} + t_{w2_2} < t_{w1_1} + t_{w3_1}. \quad (5.4)$$

The only delays remaining on the right side in equation 5.4 are wire delays. Usually, we can assume that wire delays within a CAL register are closely matched and smaller than δ_{enable} . Thus, the condition $\delta_{enable} < 0$ forces inevitably the pipeline to fail. The solution is clear: We must eliminate δ_{enable} from this equation. For the implementation this means moving the fork into the circuit.

5.2.3 Pipeline Model with Synchronized Capture Done

This section does not propose a hardware implementation. As a possible solution to the method mentioned above, the synchronization of capture done in order to move the fork should be proven in theory. It is tested here, whether this theoretical implementation would satisfy the CAL-rules or not. So the queries shown in Source 5.2.3 are again used in this example. Most of the templates built in Section 5.2.2 are reused here. Figure 5.11(b) depicts one template, which has to be defined newly. In interaction with the modified latch (see Figure 5.11(a)) it synchronizes the capture done transmission to the upstream stage.

Starting point is a closed register and an open synchronization unit. When the enable-unit switches the latch to transparent, the synchronizer `close_cd` takes the capture done template to the closed location. The next register functionality is not changed any more, but with the closing transition of the latch, the synchronizer model transits to the open location. Now the transition updating the `cdone` variable is enabled and will be taken eventually.

In contrast to the results shown in Table 5.2 the synchronization of capture done leads now to the desired behavior.

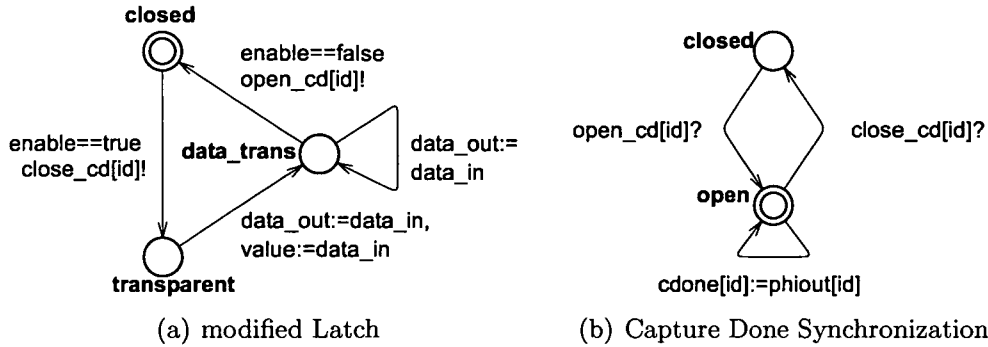


Figure 5.11: Pipeline Model with Synchronized Capture Done

Nr.	Query	Uppaal result
4	$A \square$ not deadlock	is satisfied.
5	$A \square$ fl.x.mixedphi imply not p2.transparent	is satisfied.
6	$A \square$ ps.phi0_phi1 imply not p1.transparent	is satisfied.
7	$A \square$ p2.data_trans imply not(data_in[2] != p2.value)	is satisfied.
8	$A \square$ p1.data_trans imply not(data_in[1] != p1.value)	is satisfied.

Table 5.3: Results of Uppaal checking the Pipeline Model with Synchronization

Query 4: Start and end model of the linear pipeline enforce alternating data phases. Uppaal has proven that in this system a deadlock is not possible, which ensures compliance with CAL-Rule I.

Query 5 and 6: Observance of CAL-Rule II is shown. Due to the simple task of a latch – passing the input to the output while being transparent – Rule III is also ensured.

Query 7 and 8: It is guaranteed that exactly one wave passes the latch while it is transparent. This behavior is shown for both registers used in the system.

The next task is now to find an appropriate hardware implementation for the proposed method.

5.2.4 Pipeline Implementation with Latched Capture Done

The previous section proposes a synchronization of the capture done signal. One possible implementation is to latch this signal with the inverse enable signal. When the data-latches are transparent, `data_in` will be transferred to `data_out`. In this case the "capture-done"-latch is closed. After the φ -detector Φ_{out} has indicated that the new phase is consistent, its output will not be signalled directly to the upstream stage, before the enable logic has decided to close the data-latches and so the gate of the data latches is disabled. At this point the inverse gate of the capture-done latch will

be enabled and the new phase is transmitted. Figure 5.12 shows the proposed implementation.

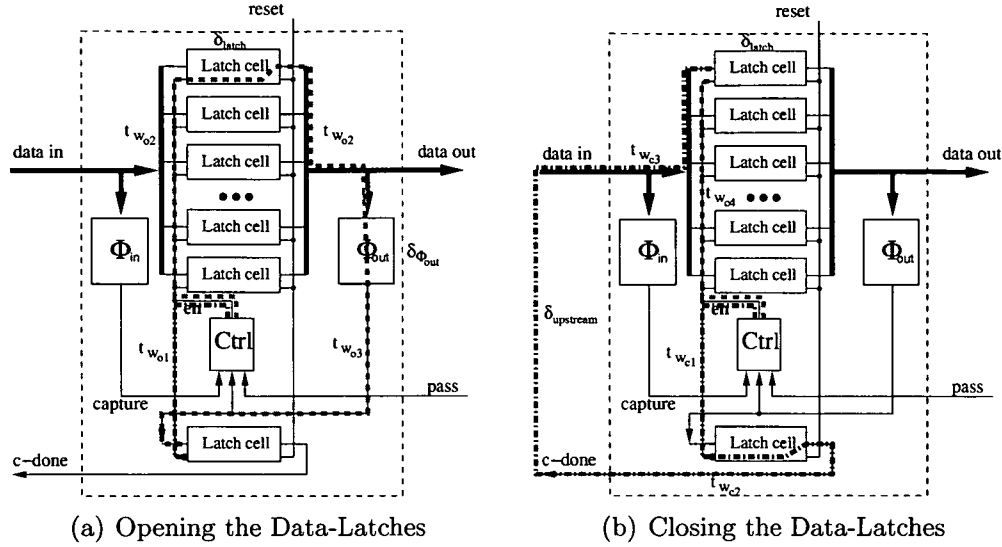


Figure 5.12: CAL Register with latched Capture Done

Taking a closer look at the timing while opening and closing the data-latches: Figure 5.12(a) shows the concurrent paths while opening the latches. The fork is situated at the output of the enable logic and the critical merging point is the input of the capture done latch. The path t_{o1} directly to the latch is marked with the slashed-dotted line and it is only made of the wire delay (5.5). The other path t_{o2} comprises the gate delay of the latches, the φ -detector Φ_{out} and three wire delays connect them (5.6):

$$t_{o1} = t_{w_{o1}} \quad (5.5)$$

$$t_{o2} = t_{w_{o2}} + \delta_{latch} + t_{w_{o3}} + \delta_{\Phi_{out}} + t_{w_{o4}} \quad (5.6)$$

The correct functionality is guaranteed when the capture done latch is closed before the new phase may arrive at the input of this latch, so it must be ensured that $t_{o1} < t_{o2}$ leads to the equation:

$$t_{w_{o1}} < t_{w_{o2}} + \delta_{latch} + t_{w_{o3}} + \delta_{\Phi_{out}} + t_{w_{o4}} \quad (5.7)$$

Figure 5.12(b) depicts the two relevant paths for closing the data latches. The fork is located at the output of the enable logic, in contrast to the opening situation the critical point merging the two concurrent paths is now located at the input of a data latch. Notice, the latch cell used for these considerations is one which completes the new data word. The slashed-dotted line depicts path t_{c1} concerning the capture done latch. The delay of the upstream units is summarized in $\delta_{upstream}$. The path comprises

the gate delay of the capture done latch, the delay of the upstream unit as mentioned before and three additional wire delays. These delays are summarized in (5.8). In this case path t_{c_2} to the input of the data latch is very simple and consists of one wire delay (5.9).

$$t_{c_1} = t_{w_{c1}} + \delta_{latch} + t_{w_{c2}} + \delta_{upstream} + t_{w_{c3}} \quad (5.8)$$

$$t_{c_2} = t_{w_{c4}} \quad (5.9)$$

To ensure proper functionality, the data latches must have been closed before new data arrive on their inputs. Thus, t_{c_1} must be smaller than t_{c_2} , which leads to (5.10):

$$t_{w_{c4}} < t_{w_{c1}} + \delta_{latch} + t_{w_{c2}} + \delta_{upstream} + t_{w_{c3}} \quad (5.10)$$

Using (5.7) and (5.10) we want to develop a rule for a correctly functioning CAL register without any restrictions to the surrounding logic. Therefore, we pessimistically assume $\delta_{upstream} = 0$, so that only effects inside of the CAL register will influence the result. In general the width of the utilized signals are unknown and so we also have to eliminate the delay of $\delta_{\Phi_{out}}$ in (5.7). Furthermore, we set all wire delays on the right side to zero. This leads to the sufficient condition that a register will be delay insensitive to its environment, namely if

$$t_{w_{o1}} < \delta_{latch} \quad (5.11)$$

$$\text{and } t_{w_{c4}} < \delta_{latch} \quad (5.12)$$

These findings shall be proven by Uppaal models. Therefore, the models are extended in a way that the upper bounds for the wire delays $t_{w_{o1}}$ and $t_{w_{c4}}$ as well as the lower bound for the latch delay δ_{latch} are ensured. To stay comparable with the preceding section, the queries to test the model are reused from Source 5.2.3. As shown in Figure 5.13 three additional templates have been designed for each CAL register. Furthermore, the templates for the latches and the enable logic have been modified:

Figure 5.13(a) models the fork at the output of the latch enable unit. If the output of this unit has been changed, the synchronizer `senlogic` is activated. Thus, the automaton is activated and the committed properties of the locations force the automaton to proceed its work immediately, in order to activate both wire models shown in Figure 5.13(b) and 5.13(c). Due to this, the delay is set to zero and an invariant on the location ensures that it will be retained at most $t_{w_{o1}}$, respectively $t_{w_{c4}}$, which are modeled both with `W_MAX` at this point. Afterwards, the two delayed versions of the enable signal are forwarded to the latch and the capture done latch. The enhancement of the enable logic is illustrated in Figure 5.13(d). Every transition modifying the original enable signal is extended with a synchronizer in order to activate the fork model. Last but not least, the model of the latch is extended with a guard to ensure that the output will not be updated before δ_{latch} (`MIN_LAT`) has elapsed.

Table 5.4 depicts the results of the Uppaal runs:

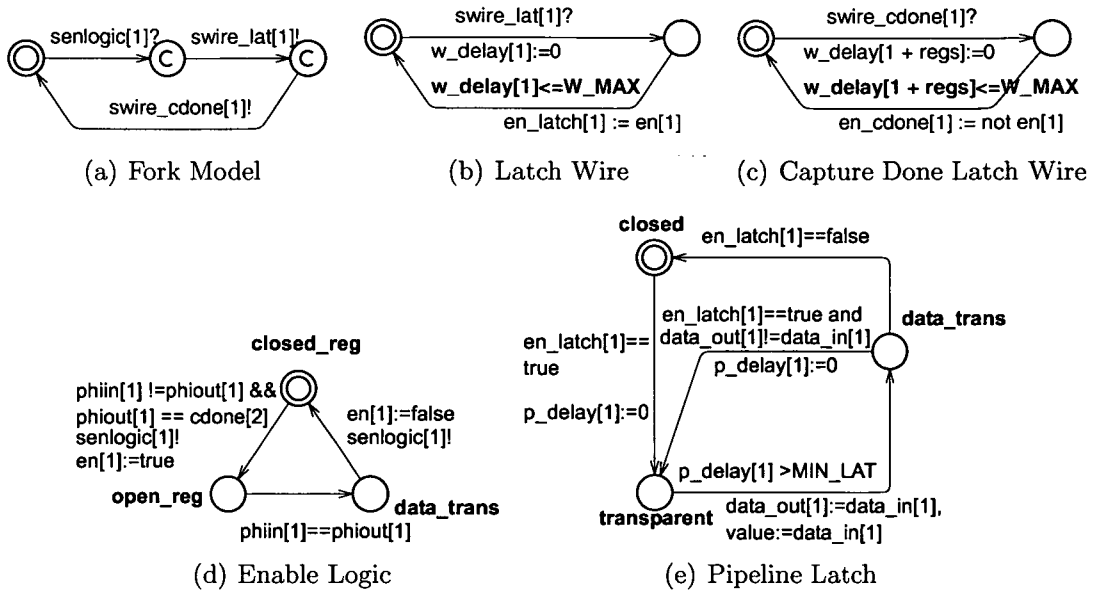


Figure 5.13: CAL Register with latched Capture Done signal

1	$E \langle \rangle$ pe.p1	is satisfied.
2	$E \langle \rangle$ p2.data_trans	is satisfied.
3	$E \langle \rangle$ f1.x.phil	is satisfied.
4	$A \square$ p2.data_trans and f1.x.mixedphi imply data_out[2] == p2.value	is satisfied.
5	$A \square$ not deadlock	is satisfied.
6	$A \square$ f1.x.mixedphi imply not p2.transparent	is satisfied.
7	$A \square$ ps.phil0_phil1 imply not p1.transparent	is satisfied.

Table 5.4: Results of Uppaal checking the "Latched Capture Done" Version

Query 1 to 3: To ensure that the model is still working after the insertion of synchronizers, the principle function tests are shown again. These results are positive.

Query 4: Using Uppaal it has been proven that in this system a deadlock is not possible, which ensures compliance with CAL-Rule I.

Query 5 and 6: Furthermore, CAL-Rule II is obeyed. Due to the functionality of a latch Rule III is also ensured.

We have shown that two conditions must be guaranteed when building a CAL register: The wire delay from the enable logic to the data latch cell as well as the delay from the enable logic to the capture done latch must be smaller than the minimal latch delay. Such CAL registers can be designed and constructed once and afterwards these basic gates can be instantiated.

Notice that with these provisions no assumptions on the surrounding logic are necessary. Thus, built once in a careful defined way, CAL registers can be used without restrictions.

5.3 The Combinational Functions ($f(x)$)

The section above has proven that it is possible to build delay insensitive circuits. This model supposes that the logic between the registers follows the rules for CAL-Logic – defined in Section 3.3. Now, the next step is to show that circuits built with CAL gates only follow these rules.

Remember the design-flow described in Section 3.6: After the functional description a design is built only with elements of the CAL-Library. Furthermore, the subsequent replace and synthesis steps transform the design to a cal_rail description. In this section we assume that the basic gates which are used here work as specified. Thus, we only have to study the interaction between these gates. The gates themselves and the timing behavior will be our focal point of Section 6.

5.3.1 Circuits built with CAL-Gates

In our approach every combinational function is built using **and**, **or** and **inversion** gates during the first synthesis. So a model for these three functions has to be built for formal verification. The transformation to an Uppaal model is described with the **AND-Gate** in detail. Models for an **OR** and an **INV**-gate are built in a similar way. Recall the schematic and the truth table of the **AND**-gate given in Section 3.5.1: If both input rails are in the same phase, the circuit will perform the requested action. Otherwise the gate will remain in its state. This behavior is modeled as portrayed in Figure 5.14:

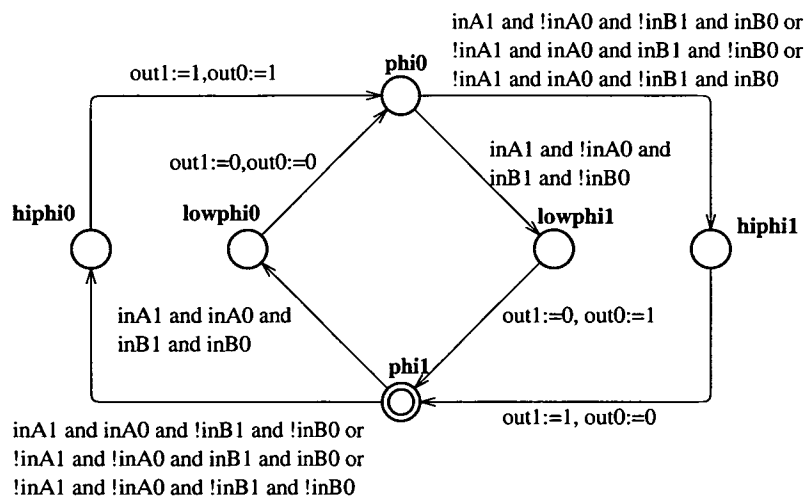


Figure 5.14: Uppaal-Model of an AND-Gate

There are two states ϕ_0 and ϕ_1 . In each of them the output of the gate can hold the logic value "HIGH" or "LOW". All valid transitions are derived from the truth-table depicted in Figure 3.5. Transitions are given from ϕ_0 to ϕ_1 , but there are none from ϕ_0 to ϕ_0 and vice versa. It depends on the input values, whether the next output will be "HIGH" or "LOW". There is an intermediate state where the model can stay without any duration restrictions. Afterwards, the corresponding output values are assigned. The automaton resides now in the opposite phase where it is able to remain even if the inputs change. After changing the inputs the automaton eventually will take the enabled transition which leaves this state. The intermediate state is an appropriate method to model arbitrary delays between the case that the inputs have changed and the activity where the output changes. The model of an OR-gate is very similar and is shown in Figure 5.15(a).

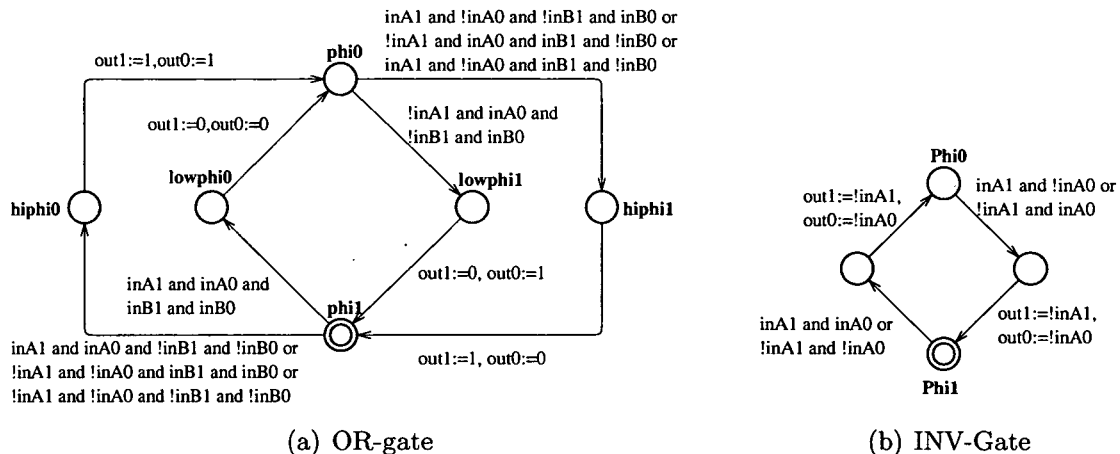


Figure 5.15: Uppaal-Models

There is no need for a storage element to build an inverter in CAL. So the model of an INV-gate differs from those of the AND/OR ones. It can be easily seen in Table 3.1 that inverting the logic value and remaining in the phase is performed by inverting both rails of the signal. This is done by the automaton shown in Figure 5.15(b). However, the intermediate states are utilized to model the delay between the input and the corresponding output.

In the following paragraphs we will show that any combination of these gates follows the rules defined in Section 3.2. Therefore, a system describing a possible $f(x)$ using all types of basic gates is used. Fulfilling Rule II the output may only change if all inputs are in the same phase. This is done by query two and three portrayed in Table 5.5. In addition, to meet rule I+III the output should only be altered once, afterwards it should keep its value. If such an unwanted event occurs, it is detected by query four and five.

To test these cases, all possible input patterns have to be created, which is done by the automaton given in Figure 5.16. Without loss of generality, input rails signal A in

this automaton are altered before those of input B. After leaving the `reset` state the model starts generating inputs for the $f(x)$ simulation.

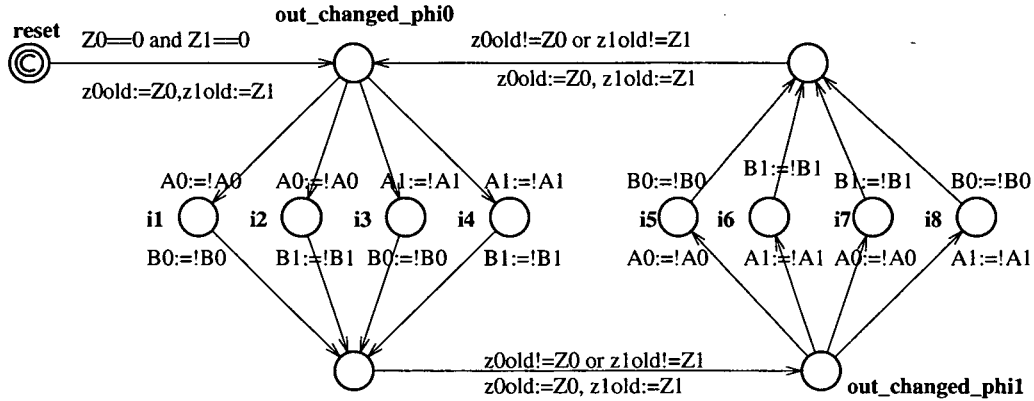


Figure 5.16: Input Generation Function proving the $f(x)$ Model

There are two important nodes – `out_changed_phi0` and `out_changed_phi1` – where the output of the logic should be stable. Furthermore, two Boolean Variables – `Z0old` and `Z1old` – are used in this automaton to store the old value of the outputs. The principle is very simple: After reset the output of the $f(x)$ simulation will be in $\varphi 0$. This state can remain with one of the four possible transitions. According to Rule I, all others are forbidden. First, one of the two rails of input A is altered. Afterwards, input B will be changed in the same way. After both changes, the automaton can stay in the states arbitrarily long. The transition to one of the `out_changed_phi[0|1]` states is enabled not until one of the outputs has changed the value. This transition updates the variables storing the old values.

Table 5.5 shows the queries which are checked with Uppaal. The first one tries to detect, whether the system can get caught in a deadlock or not. Furthermore, queries two and three prove if there is any activity on the outputs after only one input rail has changed. The last two queries ensure that there is no activity on the output after the output has changed once. This ensures that no glitch will occur. So the problem is described as follows: The automaton could reach the states `out_changed_phi0` and `out_changed_phi1` when one rail of the output has been altered. There must not be a change on an output rail when the automaton has reached one of those states.

The results of the model checker are given in the third column.

Query Nr. 1: There is no deadlock in the system. Altering one rail of each input will eventually lead to an activity of an output.

Query Nr. 2 & 3: The result of Uppaal approves that the output will never change if only one input is updated.

Query Nr. 4 & 5: There is no activity on an output when it has been altered once.

Nr.	Query	Uppaal result
1	$A \square$ not deadlock	is satisfied.
2	$E \langle \rangle (p_init.i1 \text{ or } p_init.i2 \text{ or } p_init.i3 \text{ or } p_init.i4)$ and $(p_init.z0old \neq Z0 \text{ and } p_init.z1old \neq Z1)$	is NOT satisfied.
3	$E \langle \rangle (p_init.i5 \text{ or } p_init.i6 \text{ or } p_init.i7 \text{ or } p_init.i8)$ and $(p_init.z0old \neq Z0 \text{ and } p_init.z1old \neq Z1)$	is NOT satisfied.
4	$E \langle \rangle p_init.out_changed_phi0$ and $(p_init.z0old \neq Z0 \text{ and } p_init.z1old \neq Z1)$	is NOT satisfied.
5	$E \langle \rangle p_init.out_changed_phi1$ and $(p_init.z0old \neq Z0 \text{ and } p_init.z1old \neq Z1)$	is NOT satisfied.

Table 5.5: Results of Uppaal checking the $f(x)$ simulation

So the important conclusion of this proof is: Correctly changing the input of a logic function $f(x)$ will lead to exactly one transition on one of the two output rails. So it is shown that an $f(x)$ logic will follow the CAL rules given in Section 3.2.

5.4 Summary

After a brief introduction to Uppaal, the functionality of a schematic pipeline has been used to demonstrate the procedure, of the usage of Uppaal models while proving our claims. The compliance of the principle pipeline with the CAL rules defined in Section 3.2 is shown as well. A gradual refinement of the pipeline model, however, has shown deficiencies with respect to delay insensitivity. Synchronizing capture done has turned out as a crux. The approach latching capture done with the inverted enable signal is proposed as a solution. However, two conditions must be considered in the implementation of the pipeline registers. When using a CAL register that conforms to these conditions as a basic gate there are no restrictions for building CAL circuits. At this level of abstraction CAL is delay insensitive. In the following section the focus will be on the behavior of the basic gates.

Furthermore, it has been shown that combinational functions $f(x)$ fulfill the rules for CAL circuits. Thus, no restrictions have to be applied.

Chapter 6

Delay-Insensitivity of CAL Basic Gates

Contents

6.1	Modeling Altera FPGA Designs	83
6.2	The AND Gate	88
6.2.1	Synopsys-edif-Version	88
6.2.2	Quartus-only-Version	93
6.3	The OR Gate	95
6.4	The INV Gate	97
6.5	N-Signal φ-Detector	98
6.5.1	Two Signal φ -Detector	98
6.5.2	Four Signal φ -Detector	100
6.5.3	Generic N-Rail φ -Detector	101
6.6	Latch	102
6.6.1	One Signal wide Latch	102
6.6.2	Latch with Enable Logic	104
6.7	Summary	106

We have shown in Section 5 that pipeline and $f(x)$ logic functions behave properly if gates fulfill certain rules. The purpose of this section is to point out that gates fulfill these rules, if pipe and $f(x)$ behave properly. Till now the evaluation of delay insensitivity of CAL implementations is target independent. So there are no decisions made in respect of the final hardware platform. Furthermore, all of the basic gates are considered as “black-boxes“. But now a detailed look into these boxes is necessary and therefore a precise implementation is required to investigate their behavior.

First, the proposed model of an Altera FPGA is discussed. This model is used for further examinations.

6.1 Modeling Altera FPGA Designs

As mentioned in Section 4.2.1 the target technology used for the first implementations is an Altera FPGA. Altera offers a manufacture-specific tool called Quartus[3]. In our design flow we use Quartus for performing place & route only, but it is possible to perform the whole design process with this Altera specific program.

During the compile and place & route procedure, Quartus generates a multitude of log-files. One of them is the so-called equations file [3] (see a partial equations file in Source 6.1.1).

```
--A1L61 is Z.line1~0 at LC5_14_Z1 --operation mode is normal
A1L61 = A1L01 & A1L8 & (A1L61 # !A1L7);

--A1L5 is i11~18 at LC6_15_Z1 --operation mode is normal
A1L5 = !A.line0 & A.line1 & B.line1 & !B.line0;

--A.line1 is A.line1 at Pin_106 --operation mode is input
A.line1 = INPUT();

--Z.line1 is Z.line1 at Pin_84 --operation mode is output
Z.line1 = OUTPUT(A1L61);

--Z.line0 is Z.line0 at Pin_89 --operation mode is output
Z.line0 = OUTPUT(A1L41);
```

Source 6.1.1: Example of a Quartus Equation-File

Every line describes the logical function of one LUT: The output of the LUT is the variable on the left side of the equation. On the right side, the up to four inputs and the Boolean operation are given. Details about the operators can be found in [3]. Furthermore, the equation file defines each input and output of the compiled circuit. The inputs are only characterized by the use of the function INPUT(). Every output needs the assignment of a result of the final LUT of the mathematical operation. So the names of the LUTs are mapped to the environment and vice versa.

As mentioned above, the equations are generated automatically, to be more precise, Quartus writes two versions of the equations to a file: First by the fitting step and second by mapping the design. The equations are defined during the fitting step, after the mapping is performed only the position of the LUTs on the FPGA is added. We use only the equation itself, so for our purpose it makes no difference, which of these two files is used.

The equation of a design is the starting point to build an Uppaal-model of the circuit. However, this a very error-prone operation. To overcome these problems the models are generated automatically by awk[1, 102] scripts. The whole system used by the Uppaalsimulator is described by XML-files. The conversion tools transform the information crucial to the design from the equation file to the XML-file, which can be used by the model checker.

The Boolean operation of every LUT is transferred to an automaton, as shown in Figure 6.1. We need a model which allows us to simulate various timing behavior of one LUT. Therefore, two constants are defined to specify the minimal $-\delta_{LUT_{min}}$ - and maximal $-\delta_{LUT_{max}}$ - LUT delay: MIN_GATE and MAX_GATE. Furthermore, we should be able to model the behavior of the environment to derive the possible restrictions. So the minimal delay between a transition on the output and the next input wave $-(\delta_{idle})$ - can be defined (called MIN_INIT as Uppaal constant). In a first approach we assume that the feedback delay is zero, but in further investigations it should be possible to define a non-zero feedback delay $\delta_{feedback}$.

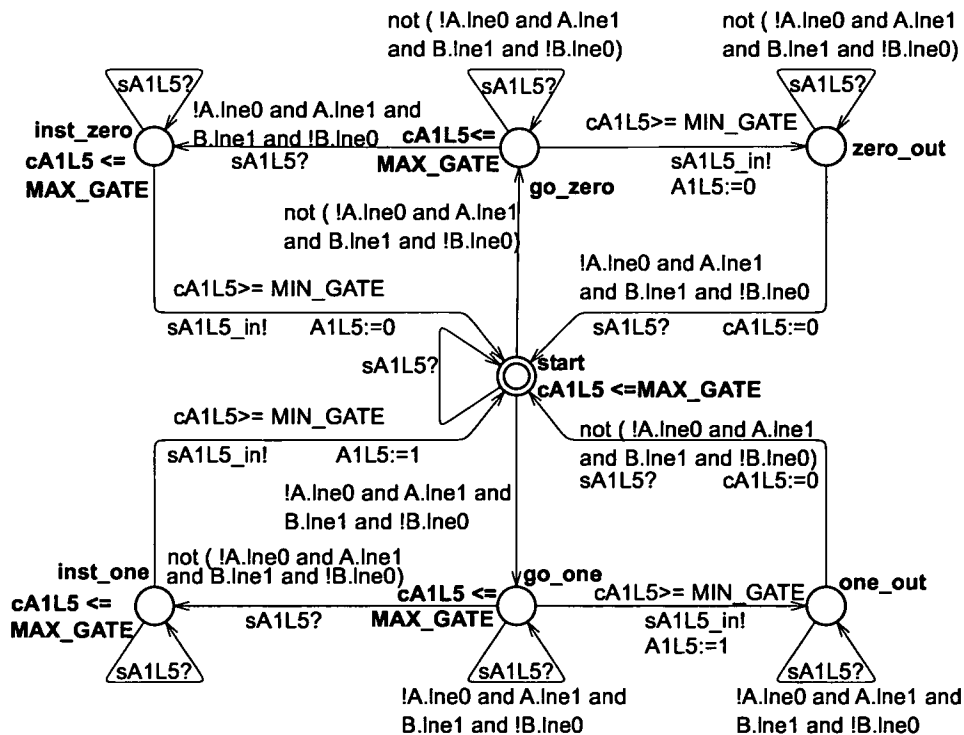


Figure 6.1: Altera Apex LUT Model

Every automaton describing a LUT consists of five so-called Locations and transitions between them. The upper part is used for assigning a "LOW" to the output of the LUT, the lower part for assigning a "HIGH". Recall Source 6.1.1, where the equation for LUT A1L5 is given as $A1L5 = !A0 \& A1 \& B1 \& !B0$. This Boolean operand is transferred to the term " $!A0$ and $A1$ and $B1$ and $!B0$ " which can be found several times in Figure 6.1: First, one can find it as the "guard" of the transition from `start` to `go_one`. The reason is intuitive: If the equation is true, the output should become "HIGH", so the reversion is also valid: If this equation is not fulfilled, the output should go to "LOW". This fact is modeled by the transition from `start` to `go_zero` with the negated Boolean operation. These two transitions describe the main functionality of the LUT. So the state to assign the derived output value is subdivided into two locations. For both values there exist corresponding `go_[zero|one]` states and with the transition from these states to the final ones the output is assigned. So there can pass some time between the decision to put the output to "HIGH" and the action where the output is assigned.

Every automaton has its own `clock[7, 6]` for modeling temporal behavior. It is used to measure the time, which has passed since the last action and the value of this clock is compared to the minimal ($\delta_{LUT_{min}}$) and maximal ($\delta_{LUT_{max}}$) LUT delay: `MAX_GATE` and `MIN_GATE`. The clock is cleared every time the automaton enters the `start` location. The automaton is allowed to stay in the `start` state, as well as in both of the `go_[zero|one]` states while the clock is less or equal to `MAX_GATE`. This is ensured by so-called Uppaal invariants[6]. The minimal delay ($\delta_{LUT_{min}}$) is ensured by the guard of the transition from the `go_[zero|one]` location to the final states `[zero|one]_out`: The clock value must be greater than `MIN_GATE`. It is important to note that the automaton is allowed to stay an arbitrary amount of time only in the `[zero|one]_out` states.

This leads to a case which has to be considered separately: The automaton must be permitted to stay as long as necessary, but it must react to changes on the inputs eventually. Synchronizers[6] are used to abut the automaton if one of the inputs has been changed. Every LUT model has one corresponding synchronizer. In Figure 6.1 it is named `sA1L5`, which brings the automaton into the `start` location. Further, altering the output must be broadcasted to all others which use the same value. This is done by activating the output synchronizer related to this LUT named `sA1L5_in` in Figure 6.1.

Uppaal synchronizers have one major drawback: A single synchronizer activated with `sync!`, is able to start *exactly one* enabled transition waiting with `sync?`. This has an impact on the whole model: It must be possible that one activity on an input triggers more than one LUT model. This problem is solved with synchronizer chains as shown in Figure 6.2. This leads to another requirement: The possibility to run the synchronizer on each location should be accomplished, otherwise the chain would be stopped which leads to a deadlock of the system. In Figure 6.1 a transition with `sA1L5?` can be found in every location, e.g. there is a transition with source and sink in location `start`:

The principle of such a chain is simple: It remains in the `ready` state until the

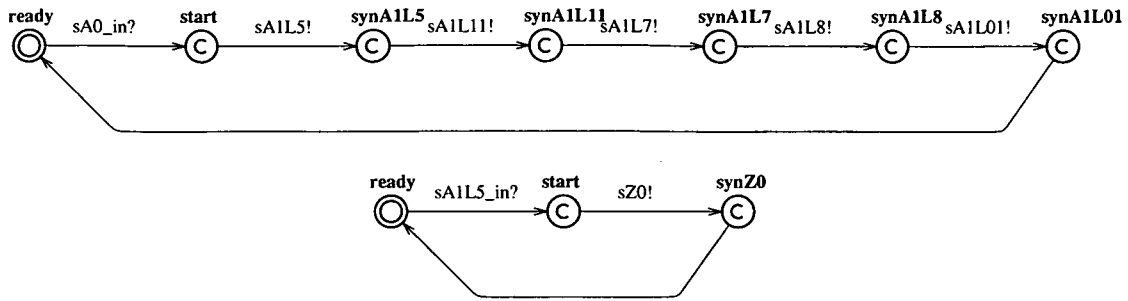


Figure 6.2: Altera Apex Wire Model

synchronizer couple is activated with one output. This occurs when the output is altered. Now the model should trigger all others locations where the only modified output is used as an input. Therefore, all corresponding synchronizers are chained together. So it becomes clear that the length can vary, as shown in Figure 6.2. The "committed" property ensures that the chain is processed completely. Afterwards, the input values altered before will be handled by the LUT models. The upper automaton in Figure 6.2 shows the chain of input A0, which is used in LUT A1L5, A1L11, ... A1L01. Remember Figure 6.1 where the value of A1L5 is changed; this triggers the second part of Figure 6.2. Here just one LUT uses the output for further calculations.

The model is able to handle propagation delays of a LUT and the wire delay between diverse LUTs. These delays are combined and modeled at once with the `MAX_GATE` and `MIN_GATE` constants. However, there is no possibility to describe the feedback delay of an output of a LUT back to one of its own inputs. Therefore, the wire-model shown above is replaced by one which has a feedback delay ($\delta_{feedback}$) - `D_WIRE`:

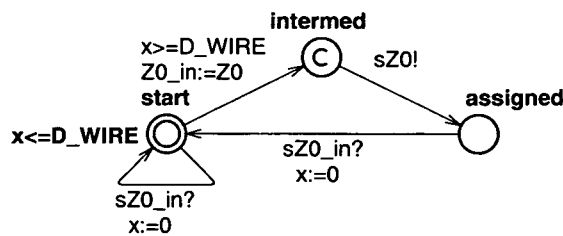


Figure 6.3: Feedback Model

The automaton sets on working in the `start` location. However, entering the state will cause a reset of the internal clock `x`. The time is measured while the automaton remains in this state. Two rules guarantee that the input value `Z0` will be assigned to the output value - `Z0_in` after the defined delay `D_WIRE`: The guard of the transition leaving the `start` location is enabled when the clock `x` is greater or equal than `D_WIRE`. Further, the invariant of the `start` location allows the automaton to stay while the clock `x` is less or equal than the predefined delay. Exactly at the predefined delay the

output will get the value of the input and the automaton transits to the assigned state. Changing the input will cause an activity of the synchronizer $sZ0_in$, which brings the automaton back to the start location. Also, the clock x will be put back to zero.

Finally, a model to generate the inputs for the gates is needed. This is performed by an automaton as shown in Figure 6.4:

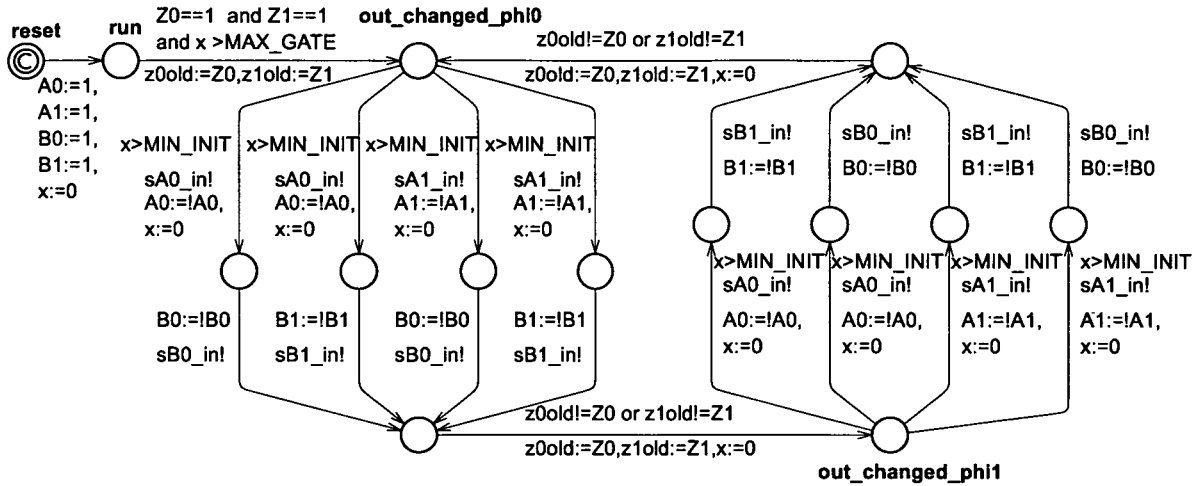


Figure 6.4: Stimuli-Generation

At the beginning all inputs are initialized and the global clock x is set to zero. There are two locations where the output of the gate under investigation should not change its value: $out_changed_phi0$ and $out_changed_phi1$. In these locations the automaton should remain as long as the clock is less than the minimal input idle time (δ_{idle}) MIN_INIT . Afterwards, one of the input rails is altered and the automaton resides in one of the intermediate states. Eventually, the transition to the next common location is taken which is left after one of the output rails changes. This enables the transition to $out_changed_phi1$ or $out_changed_phi0$. The variables containing the former value of the outputs ($Z0old$ and $Z1old$) are altered and the system is now in a stable state with outputs in phase $\varphi1$. The procedure from $\varphi1$ to $\varphi0$ is similar.

So there are several constants influencing the behavior of the Uppaal-model:

$\delta_{LUT_{min}}$, MIN_GATE : this constant defines the minimal delay for every LUT in the system to assign the new output value.

$\delta_{LUT_{max}}$, MAX_GATE Similar to the point above, the maximal gate delay can be set with this constant. In conjunction with the minimal gate delay a range ($\Delta_{\delta_{LUT}}$) can be defined in which the outputs of all LUTs will definitely reach their new value. Both delays comprise the LUT delay as well as the wire delay between the LUTs.

δ_{idle} , MIN_INIT is the minimal delay between the transition of the output and the next change of an input. So this is the time where the model can stabilize its state.

However, in practice this delay depends on the environment.

$\delta_{feedback}$, D_WIRE describes the exact delay of a feedback wire from the output of a LUT to the input of this LUT. This constant is necessary to build storage loops, e.g. for RS-latches. However, $\delta_{LUT_{min}}$ and $\delta_{LUT_{max}}$ are used to define a range of possible output transitions concerning the wire and LUT delays. $\delta_{feedback}$ is a fixed value used to define the additional feedback delay.

Using these four constants many variations of the following models are generated and investigated.

6.2 The AND Gate

The schematic and the truth table of the AND-gate given in Section 3.5.1. In this section its hardware implementation is described. As a first approach our "standard" CAL-design flow (cf. Section 3.6) is used and the results are examined. The description of the functionality written in VHDL is processed by the Synopsys design-analyzer. The resulting hardware is exported as an edif[29, 56] netlist. Moreover, Quartus is able to read this format and so the place & route step can be performed. Additionally, Quartus produces the download file as well as the equation file.

6.2.1 Synopsys-edif-Version

The schematic of the result of synthesis and place & route can be viewed in Figure 6.5. It depicts the LUT structure of the four inputs (A0, A1, B0, and B1) and the two outputs Z0 and Z1.

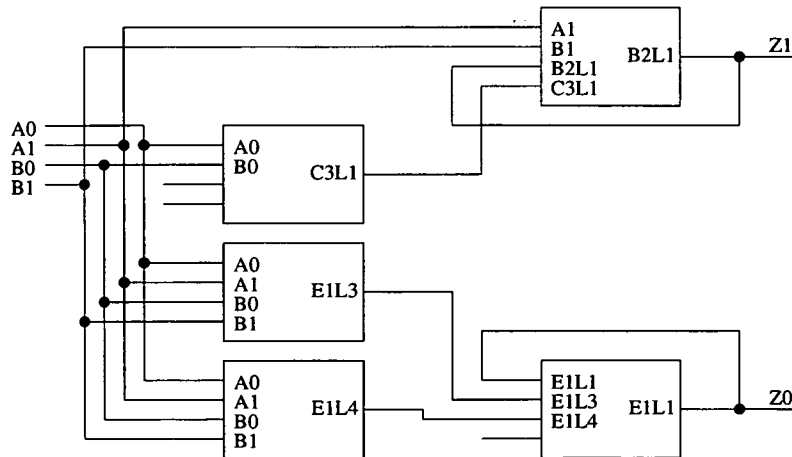


Figure 6.5: AND-LUT Schematic generated by Synopsys

However, the functionality is implemented by five LUT's. Recall the functional description given in Section 3.5.1: Two RS-latches with the corresponding logic to

compute the `set` and `reset` signals of both RS-latches are needed. LUT E1L1 and B2L1 form such an RS-latch with feedback wires. There are two LUTs generating the inputs for the RS-latch for Z0, but there is only one for Z1. So parts of the set or reset logic are merged within LUT C3L1.

The structure and the equation of every LUT is transferred to an Uppaal-model using the parts described in Section 6.1. Using these models several examinations are conducted.

The first model that has been used consists of five LUT automata and a wire-automaton for each connection between the LUTs. In a first approach the impact of LUT delays is checked. So the feedback is only built with a wire-automaton, i.e. there is no special treatment of these lines. The `MIN_INIT` constant is used to simulate the behavior of the environment. These assumptions can be enough to meet external restrictions, which lead to a functioning circuit. However, these conjectures can be too weak and the result will be an incorrect gate.

The goal of these investigations is to answer the question whether or not this implementation is delay insensitive. Furthermore, it is important to derive the constraints which have to be fulfilled to guarantee a DI circuit. Valid CAL designs imply an important property: There is only one transition on one of the two rails in one data wave. Using this, an event can be used to perform some kind of completion detection. This leads to a rule for gates based on CAL: Whenever a rail changes its value, it must be the final update of the values of both rails. So there must not be a transition when one of the rails has modified the state. However, it is irrelevant whether the modification was correct or not. Further, remember Figure 6.4: The stimuli generation progresses after one output has been altered. So if there is more than one output change, the automaton will continue working, but the output is not in a correct phase. This leads to a deadlock of the model, because the phases of the inputs will never agree again. The disprove of these effects is done with the Uppaal-queries shown in Source 6.2.1:

```
A[] not deadlock %
E<> p_stimuli.out_changed_phi0 and (Z1 != p_stimuli.z1old or Z0 != p_stimuli.z0old)%
E<> p_stimuli.out_changed_phi1 and (Z1 != p_stimuli.z1old or Z0 != p_stimuli.z0old)%
```

Source 6.2.1: Uppaal-Queries

The first query is intuitive: Is it possible to have no deadlock? The others check if there is a possibility that an output could be altered after it has been altered once. If the implementation with an assumed delay configuration is DI, the first query should be satisfied, the others should not¹.

Every line in Table 6.1 shows the results of these three queries using the parameterized model for the AND gate. First, only the influence of the gate and wire delays is tested. The wire delay is combined with the LUT delay of the following instance.

¹However, every query given in the form $\forall a \rightarrow b$ can be transformed to $\neg\exists a \rightarrow \neg b$

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$		\forall not deadlock	$\exists \varphi0$ -change	$\exists \varphi1$ -change
1	10	100	20	<i>NOT satisfied</i>	<i>satisfied</i>	<i>satisfied</i>
2	10	100	95	<i>NOT satisfied</i>	<i>satisfied</i>	<i>satisfied</i>
3	30	100	200	<i>satisfied</i>	<i>satisfied</i>	<i>satisfied</i>
4	50	100	200	<i>satisfied</i>	<i>satisfied</i>	<i>satisfied</i>
5	50	100	51	<i>NOT satisfied</i>	<i>satisfied</i>	<i>satisfied</i>
6	60	100	0	<i>satisfied</i>	<i>NOT satisfied</i>	<i>NOT satisfied</i>
7	90	100	20	<i>satisfied</i>	<i>NOT satisfied</i>	<i>NOT satisfied</i>

Table 6.1: Gate Delay and the AND-gate without Feedback

With these Uppaal models some very interesting facts can be observed: The first two runs fail because the difference between the minimal and the maximal LUT is too great. Recall Figure 6.5 and assume that E1L3 and E1L4 calculate "set" and "reset" for the RS-latch E1L1. However, in our example Synopsys did not partition the logic exactly in those three parts. A closer look at the equations of the LUTs discovers that by a combination of E1L3 and E1L4 the output of Z0 is driven to "HIGH". Changing the inputs from one phase to the next will cause two activities on the inputs; each input signal will change exactly one rail. So it is possible that one of the inputs has already changed its value, but the second has not. The set and reset logic are now not clearly defined. In the case that the inputs are not in the same phase there should be no action on the output. So there must not be an active signal neither on set nor on reset. After the second input has reached its new value, set and reset are recomputed and the output is driven to needed state.

But if the duration of this calculation differs too much, there will be a hazard. E.g., the output is "HIGH" and the time for the computation of set is twice as the time the process for reset requires. In this case the output is high and the calculation of the set signal is started, but would not be needed in this case. In addition, the other rail of the output changes very fast and so the whole logic may advance, but the set logic is still computing the next value. So the next inputs can arrive, where the reset logic has to drive the output to "LOW" and in this example it can be performed very fast. If the reset-logic is more than twice as fast as the set-logic, the output will be driven to "LOW" before the set activity of the preceding step has come to an end. Unfortunately, now a hazard may occur: The output will be driven to "LOW", the whole circuit may advance again, but now the set logic changes this output to one once again. However, this looks like the next correct code word and this may cause a new phase.

The simulation of an AND gate (see Figure 6.6) is used to confirm these investigations. It depicts the transition from a "HIGH" in phase $\varphi0$ to "HIGH" in phase $\varphi1$ and the result of the AND-operation. The last five waves in this figure show the outputs of the LUTs in the FPGA. Furthermore, the output of LUT E1L1 and B2L1 form the output signal Z of the gate and as the result of an AND-operation the output is also

”HIGH” in both phases. However, Figure 6.6 shows a working scenario, because the given implementation of the FPGA with the small delays leads to a working circuit. The results using other delay values are presented later in this section.

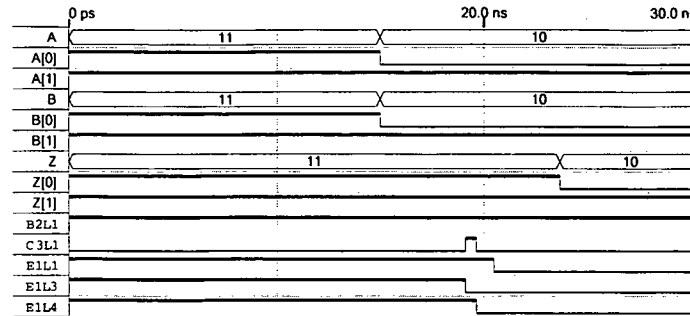


Figure 6.6: Simulation of the AND2-model

As mentioned above the output has to switch from ”HIGH” in phase φ_0 to φ_1 – so output Z0 is driven from one to zero and Z1 should remain one. As shown in Figure 6.5, LUT E1L3, E1L4 and finally E1L3 are used to compute Z0. The computation of E1L1, E1L4 runs in parallel, unfortunately these two LUTs are not the initial set- and reset-function blocks. In our case – driving the output to ”HIGH” – set should be ”HIGH” and reset should be ”LOW” and as figured out above, both inputs of E1L3 are ”HIGH”. The synthesis has melt the set, reset and RS-latch function blocks and divides it on three LUTs. So changing one of them leads to an update of an output.

The second problem occurs due to the two unbalanced paths: If δ_{idle} is less than $\delta_{LUT_{min}}$ the feedback of LUT B2L1 generates an error. Since $\delta_{feedback}$ is assumed with zero, the time from changing the input of LUT B2L1 to the point in time where the new output of B2L1 is on the input again includes also $\delta_{LUT_{min}}$. One possibility to overcome the problem described above is to delay the changes of the input. This is done with the stimuli delay MIN_INIT in the used model. In Table 6.1 the parameters and results of these trials are listed (line three to five) and the different effects are shown. Model number five simulates the case that δ_{idle} is greater than $\Delta_{\delta_{LUT}}$. Also in this case there is a possibility the outputs may change after they have changed once. In contrast, the time between input variances is much greater in the cases shown in line three and four. However, in both cases the minimal value to get a delay insensitive behavior of the stimuli delay – derived with further models – is shown.

This long period avoids the deadlock case, but the outputs are still out of specification. The last two cases are feasible: No deadlock occurs and the output changes only once and remains in this state.

The result of the tests shown in Table 6.1 can be summarized as follows. However, there were much more experiments, but only the significant cases are listed in Table 6.1:

$$\Delta_{\delta_{LUT}} < \delta_{LUT_{min}} \quad (6.1)$$

In other words: If the LUTs are built equally – the difference of the delay is nearly zero, there are no restrictions on the inputs. It is important to note that the feedback delay is assumed as zero.

This leads to the next analysis where the delay of the feedback wire is modeled as shown in Figure 6.3. The stimuli delay is defined as specified above, only the wire delay is under examination. Table 6.2 shows the result of these Uppaal-runs.

Nr.	$\Delta_{\delta_{LUT}}$		δ_{idle}	$\delta_{feedback}$	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$			\forall not deadlock	$\exists \varphi_0$ -change	$\exists \varphi_1$ -change
1	10	100	100	20	NOT satisfied	satisfied	satisfied
2	100	100	0	0	NOT satisfied	satisfied	satisfied
3	100	100	0	10	NOT satisfied	satisfied	satisfied
4	90	100	100	1	NOT satisfied	satisfied	satisfied
5	90	100	20	21	NOT satisfied	satisfied	satisfied
6	90	100	20	20	satisfied	NOT satisfied	NOT satisfied
7	90	100	20	11	satisfied	NOT satisfied	NOT satisfied
8	90	100	20	9	NOT satisfied	satisfied	satisfied
9	90	100	20	95	NOT satisfied	satisfied	satisfied

Table 6.2: LUT Simulation with Feedback (AND-gate)

The asymmetric construction of the gate combined with the logic construction – set and reset are not exactly built as separate LUTs – leads to a problematic behavior. First, the result of query one confirms the finding of the previous analysis: $\Delta_{\delta_{LUT}}$ may not be greater than $\delta_{LUT_{min}}$. Otherwise, the circuit will not work properly. Query two and three investigate a close defined case. Both delays are equal ($\Delta_{\delta_{LUT}} = 0$), so every LUT has to assign the output after this delay. Furthermore, δ_{idle} is set to zero, because it should not be necessary to have any restrictions on the inputs. Unfortunately, both models are not free of deadlocks. Neither a feedback delay of zero nor ten leads to a satisfying model. The rest of the simulation runs are performed with the delay setup of $\delta_{LUT_{min}} = 90$ and $\delta_{LUT_{max}} = 100$, so $\Delta_{\delta_{LUT}}$ is quite small. Recall Table 6.1 where this delay configuration in conjunction with δ_{idle} of 20 leads to a stable system. However, the addition of feedback changes the behavior fundamentally: If a wire delay is greater than the minimal gate delay, it leads to an unstable behavior. Anyhow, this is quite intuitive. By reducing the feedback delay it does not lead to a functioning model. As recently as the wire delay is smaller than the stimuli delay the circuit is working as specified. But when the wire delay is smaller than the difference of the LUT-delays, the behavior becomes unstable again. This outcome poses a great dilemma. Summarizing the results:

$$\delta_{idle} \geq \Delta_{\delta_{LUT}} \quad (6.2)$$

$$\delta_{feedback} \geq \Delta_{\delta_{LUT}} \quad (6.3)$$

The conjunction of both results is unsatisfying. On the one hand $\Delta_{\delta_{LUT}}$ should be as short as possible to minimize the input restrictions. On the other hand the

feedback delay must be shorter than the minimal delay, but has to be greater than this difference.

The problem is the construction of output Z1: Synopsys optimizes the circuit and so the set, reset and RS-latch-parts are melt together. This problem can be solved with special design rules as discussed in the following section.

6.2.2 Quartus-only-Version

In contrast to the previous approach, constraints are used to influence the synthesis. The goal is to build symmetric logic blocks; this means to prevent the tools to combine functionality and to keep distinct set and reset logic blocks. As shown in Figure 6.7, it is possible to build such a symmetric structure. However, two additional LUTs are needed.

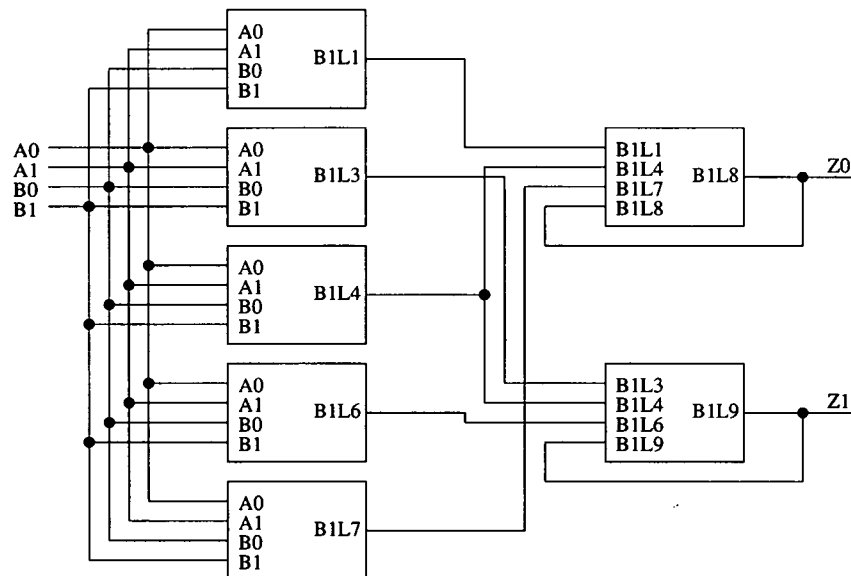


Figure 6.7: AND-LUT Schematic

A closer look at the equations-file makes sure that LUT B1L8 and B1L8 are really only RS-latches. They have no other functionality, which avoids the problems described above.

The queries from Source 6.2.1 are used here once more to prove the DI capabilities. The results of the experiments with a zero-delay feedback of the RS-latch are shown in Table 6.3. They describe a different behavior than the first implementation of the AND-gate, although the same variations are used with these models:

The results describe the following behavior: Only the first example does not work as specified: In this example the difference between minimal and maximal delay is great and the corresponding time between the two data waves is too small. So there is the possibility to proceed with the next wave, while one part of the current wave

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$		\forall not deadlock	$\exists \varphi_0$ -change	$\exists \varphi_1$ -change
1	10	100	20	<i>NOT satisfied</i>	<i>satisfied</i>	<i>satisfied</i>
2	10	100	95	satisfied	NOT satisfied	NOT satisfied
3	30	100	200	satisfied	NOT satisfied	NOT satisfied
4	50	100	200	satisfied	NOT satisfied	NOT satisfied
5	50	100	51	satisfied	NOT satisfied	NOT satisfied
6	60	100	0	satisfied	NOT satisfied	NOT satisfied
7	90	100	20	satisfied	NOT satisfied	NOT satisfied

Table 6.3: Gate Delays of an AND-gate

is still computed. After completion of this held up computation, the update of an output is performed in the wrong context. Instead of confirming a state, an once altered output is updated for a second time and therefore set to a wrong value. All other implementations work as expected. A difference between the delay of the LUTs smaller than the minimal delay will cause no restrictions on the input timing. If the gate delays vary to a greater extent than the minimal one, the time between the completion of one wave and the first update of one input rail must be greater than the value of the delay-difference (see Query Nr. 1 and 2 in Table 6.3).

The next step is to take a closer look on the influences of the feedback. Here queries and delay configuration of the edif-example are used once again. However, the results of the Uppaal runs are much more satisfying.

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	$\delta_{feedback}$	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$			\forall not deadlock	$\exists \varphi_0$ -change	$\exists \varphi_1$ -change
1	10	100	100	20	satisfied	NOT satisfied	NOT satisfied
2	100	100	0	0	satisfied	NOT satisfied	NOT satisfied
3	100	100	0	10	satisfied	NOT satisfied	NOT satisfied
4	90	100	100	1	satisfied	NOT satisfied	NOT satisfied
5	90	100	20	110	satisfied	NOT satisfied	NOT satisfied
6	90	100	20	20	satisfied	NOT satisfied	NOT satisfied
7	90	100	20	111	<i>NOT satisfied</i>	<i>satisfied</i>	<i>satisfied</i>

Table 6.4: LUT Simulation with Feedback (AND-gate)

First, the influence of a great difference of the LUT delays is investigated (Results see Table 6.4). If δ_{idle} is long enough there will be a stable system. In contrast the next two queries check the case where the minimal and maximal LUT delay are equal and so $\delta_{idle} = 0$. It makes no difference if $\delta_{feedback}$ is zero or has any other small value – the system behaves as required by the CAL-specification. It is important to note that these two cases require no restriction concerning the input delays. The remaining four queries check the influence of $\delta_{feedback}$ in conjunction with δ_{idle} . Query four shows

that the wire delay may be smaller than the difference between the LUT delays $\Delta_{\delta_{LUT}}$. The outcomes of query five to seven can be summarized as follows: The minimal LUT delay in addition with the stimuli delay must be smaller than the wire delay to have a correct functionality. So the wire delay must be at least as long as the minimal LUT delay:

$$\delta_{idle} \geq \Delta_{\delta_{LUT}} \tag{6.4}$$

$$\delta_{feedback} \leq \delta_{LUT_{min}} + \delta_{idle} \tag{6.5}$$

Rules to build a delay insensitive AND-GATE with LUTs

- The resulting hardware must have a symmetric structure.
- $\delta_{LUT_{min}} \geq \Delta_{\delta_{LUT}} - \delta_{LUT_{min}}$
- $\Delta_{\delta_{LUT}} \leq \delta_{LUT_{min}} \Rightarrow \delta_{idle} = 0$
- $\delta_{feedback} \leq \delta_{LUT_{min}} + \delta_{idle}$

6.3 The OR Gate

The first implementation of an OR-gate is done in the same way as described in Section 6.2.1. So it is not really surprising that the result is similar. As shown in Figure 6.8, the logic blocks for Z1 are melt and the OR-gate is built by means of five LUTs.

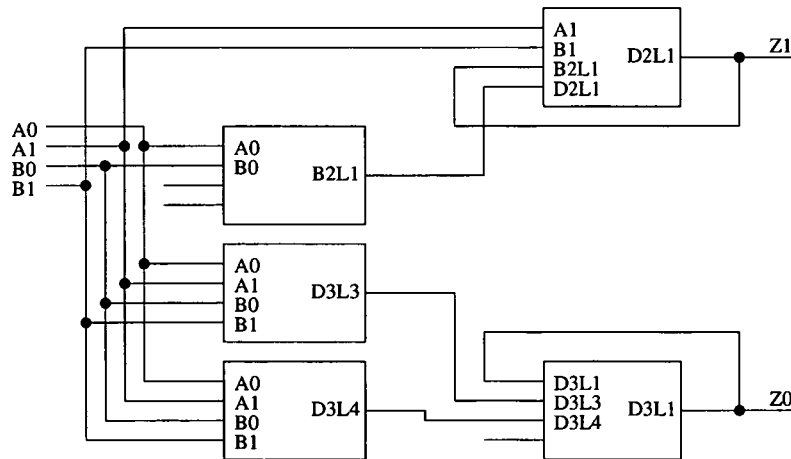


Figure 6.8: OR2-LUT EDIF-Schematic

This structure leads to the same problems as described in the previous section.

The results of the simulation performed on the model without feedback are the same compared with the AND-gate. Furthermore, the solution to overcome these problems is to perform synthesis and place & route with constraints. The resulting hardware of this approach is shown in Figure 6.9 and also displays a symmetric schematic.

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$		\forall not deadlock	$\exists \varphi_0$ -change	$\exists \varphi_1$ -change
1	10	100	20	NOT satisfied	satisfied	satisfied
2	10	100	95	NOT satisfied	satisfied	satisfied
3	30	100	200	satisfied	satisfied	satisfied
4	50	100	200	satisfied	satisfied	satisfied
5	50	100	51	NOT satisfied	satisfied	satisfied
6	60	100	0	satisfied	NOT satisfied	NOT satisfied
7	90	100	20	satisfied	NOT satisfied	NOT satisfied

Table 6.5: Gate Delays of an OR-Gate

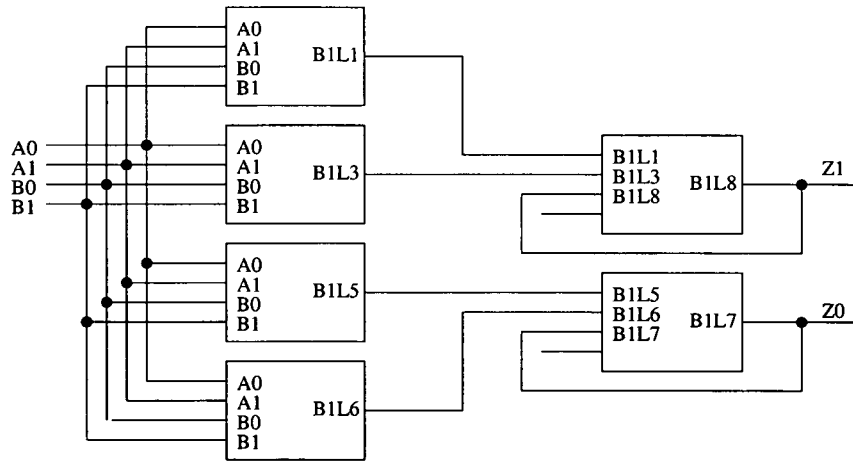


Figure 6.9: Symmetric OR-Gate

The time-behavior of an OR-gate is investigated with the rules defined in Source 6.2.1. The results of the test are equal to the results of the AND-gate, so here only the results concerning feedback are presented in Table 6.6:

Therefore, the rules to build a working OR-gate are the same as those for an AND-gate:

- The resulting hardware must be symmetric.
- $\delta_{LUT_{min}} \geq \Delta\delta_{LUT} - \delta_{LUT_{min}}$
- $\Delta\delta_{LUT} \leq \delta_{LUT_{min}} \Rightarrow \delta_{idle} = 0$
- $\delta_{feedback} \leq \delta_{LUT_{min}} + \delta_{idle}$

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	$\delta_{feedback}$	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$			\forall not deadlock	$\exists \varphi0$ -change	$\exists \varphi1$ -change
1	10	100	100	20	satisfied	NOT satisfied	NOT satisfied
2	100	100	0	0	satisfied	NOT satisfied	NOT satisfied
3	100	100	0	10	satisfied	NOT satisfied	NOT satisfied
4	90	100	100	1	satisfied	NOT satisfied	NOT satisfied
5	90	100	100	20	satisfied	NOT satisfied	NOT satisfied
6	90	100	20	20	satisfied	NOT satisfied	NOT satisfied
7	90	100	20	111	<i>NOT satisfied</i>	<i>satisfied</i>	<i>satisfied</i>

Table 6.6: Gate Delays of an OR-Gate (full-Quartus) with Feedback

6.4 The INV Gate

As mentioned in Section 3.4.2, the inversion gate is very easy to build. Both input lines have to be inverted to get the CAL-inverter. Figure 6.10 shows the result of synthesis and place & route:

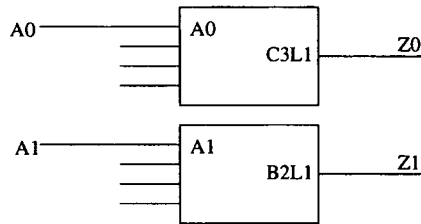


Figure 6.10: INV-LUT Schematic

A LUT is used in both rails and thus a single inverter is rather expensive. However, in many cases Quartus will merge this inverter with surrounding logic elements. The result of Uppaal models of a stand-alone inverter is given in Table 6.7 and looks quite dull:

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$		\forall not deadlock	$\exists \varphi0$ -change	$\exists \varphi1$ -change
1	10	100	20	satisfied	NOT satisfied	NOT satisfied
2	10	100	95	satisfied	NOT satisfied	NOT satisfied
3	30	100	200	satisfied	NOT satisfied	NOT satisfied
4	50	100	200	satisfied	NOT satisfied	NOT satisfied
5	50	100	51	satisfied	NOT satisfied	NOT satisfied
6	60	100	0	satisfied	NOT satisfied	NOT satisfied
7	0	100	0	satisfied	NOT satisfied	NOT satisfied

Table 6.7: Gate Delays of an Inverter

As anticipated, every timing pattern used above leads to a working inverter. Furthermore, no timing assumptions have to be made and the inverter will work with any delay constellation of the used LUTs and wires. So there are no complications if INV-gates are merged into surrounding elements.

6.5 N-Signal φ -Detector

Recall the function of an φ -detector as described in Section 3.5.2: Starting with an n-bit wide signal, the result of the computation is the phase of this bus. A very important rule is that the output is only changed if all signals are in the same phase. However, in a design various φ -detectors are used and so a diversity of width of such φ -detectors is applied. Here a digression to the design flow is necessary. The behavioral CAL description is able to handle changing bus widths easily. Using VHDL, these buses are defined with a particular width and the according φ -detector is defined with a variable input-width in the library. The adaption is done automatically. Unfortunately, adjustment of hardware, as described with `cal_rail_logic` is not as simple. Special VHDL constructs, e.g. `generics` and `generate`, are used to build the needed φ -detectors on demand during the synthesis. Synopsys is able to build them, but the optimization using Quartus pre-compiled blocks as described above is not applicable easily. First, two automatically generated φ -detectors are described, followed by a generic solution for n-bit wide φ -detectors.

6.5.1 Two Signal φ -Detector

The first circuit under investigation is the two signal wide φ -detector. Due to its nature it is the smallest that makes sense to build. The two signals use four rails A0 – A3 to represent the inputs. Figure 6.11 shows results of synthesis and place & route. After synthesis and place & route, the inputs are not automatically separated on signal level, e.g. represented as `A[0].a` and `A[0].b`, `A[1].a` and `A[1].b`. Instead, the inputs are merged to one bus.

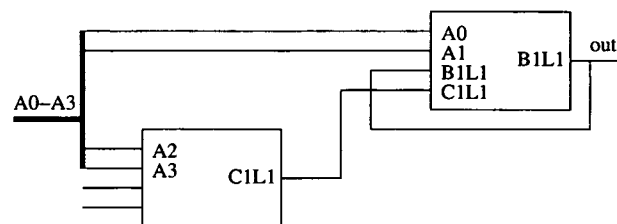


Figure 6.11: Two Signal φ -Detector LUT Schematic

The design is tested with two Uppaal model types. The first model uses ideal RS-latches. This is done by modeling the feedback delay with zero. In Table 6.9 the finding of the Uppaal runs using models without feedback are listed.

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$		\forall not deadlock	$\exists \varphi 0$ -change	$\exists \varphi 1$ -change
1	10	100	20	satisfied	NOT satisfied	NOT satisfied
2	10	100	95	satisfied	NOT satisfied	NOT satisfied
3	30	100	200	satisfied	NOT satisfied	NOT satisfied
4	50	100	200	satisfied	NOT satisfied	NOT satisfied
5	50	100	51	satisfied	NOT satisfied	NOT satisfied
6	60	100	0	satisfied	NOT satisfied	NOT satisfied
7	90	100	20	satisfied	NOT satisfied	NOT satisfied

Table 6.8: Gate Delays of a Four Rail φ -Detector

The assumption of a zero-delay feedback leads to a well-working circuit as shown in Table 6.8. However, the fiction of an ideal RS-latch is abandoned using the more realistic models containing feedback and the outcomes of these computations show a different situation (see Table 6.9):

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	$\delta_{feedback}$	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$			\forall not deadlock	$\exists \varphi 0$ -change	$\exists \varphi 1$ -change
1	10	100	100	20	satisfied	NOT satisfied	NOT satisfied
2	100	100	0	0	satisfied	NOT satisfied	NOT satisfied
3	100	100	0	10	<i>NOT satisfied</i>	<i>satisfied</i>	<i>satisfied</i>
4	90	100	100	1	satisfied	NOT satisfied	NOT satisfied
5	90	100	20	105	<i>NOT satisfied</i>	<i>satisfied</i>	<i>satisfied</i>
6	0	100	20	20	satisfied	NOT satisfied	NOT satisfied
7	90	100	20	95	<i>NOT satisfied</i>	<i>satisfied</i>	<i>satisfied</i>

Table 6.9: Gate Delays of a Four Rail φ -Detector with Feedback

Query one and two succeed because δ_{idle} is greater (or equal) than $\delta_{feedback}$. In contrast, configuration three fails, because the feedback delay is greater than δ_{idle} . The reason for this behavior is based on the asymmetric treatment of inputs. In contrast to inputs A0 and A1, which are inputs of the RS-latch constructing LUT B1L1, A2 and A3 are preprocessed. The result of this XOR-gate is delayed to a minimal extent with $\delta_{LUT_{min}}$, whereby the other two inputs are not held up. This determines the design restrictions (6.6), which is mainly driven by the value of δ_{idle} . As it can be seen, $\Delta\delta_{LUT}$ has no influence on whether the circuit works or not. The relation of δ_{idle} and $\delta_{feedback}$ determines the behavior of these models. The design rule can be summarized as:

$$\delta_{idle} \geq \delta_{feedback} \quad (6.6)$$

The correct functionality depends on the idle time of the inputs and thus on design constraints, because $\delta_{feedback}$ will always be greater than zero. Nevertheless, this situation can be improved using construction methods proposed in Section 6.5.3.

6.5.2 Four Signal φ -Detector

The Synopsys generated eight rail φ -detector is shown in Figure 6.12.

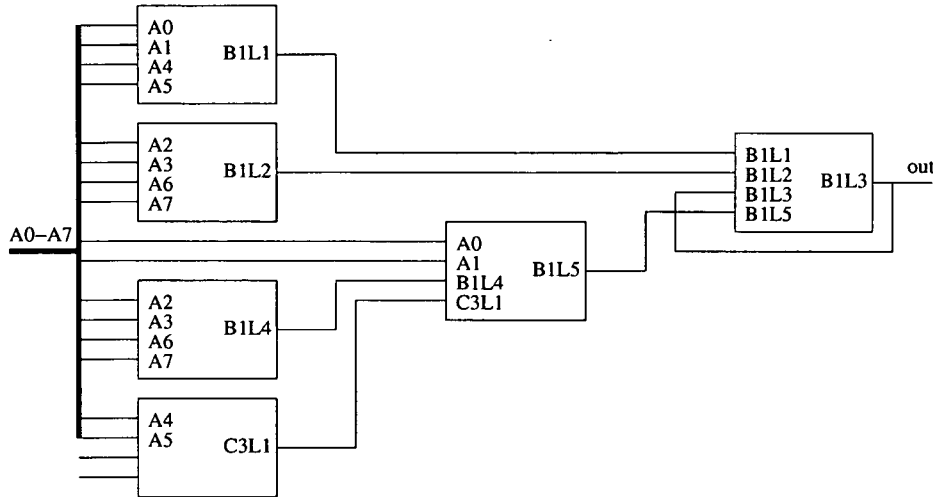


Figure 6.12: Four Signal φ -Detector LUT Schematic

LUT B1L3 implements the needed RS-latch, the remaining LUTs build the XOR-combination and the "all-ones" and "all-zero" detections as described in Section 3.5.2. Even so, in contrast to the schematic, Synopsys has doubled the XOR-gates and merged them with AND and OR computations. For example, B1L1 and B1L2 are the pairwise XOR-conjunction of all inputs combined with an OR-function. To complete the "all-zero" detection, the results of both LUTs are combined by the LUT acting as RS-latch. Unfortunately, five instead of the given four inputs would be required to do the same job for the "all-ones" detector. This problem is solved by the additional LUT B1L5, but this leads to an asymmetric design again.

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$		\forall not deadlock	$\exists \varphi 0$ -change	$\exists \varphi 1$ -change
1	10	100	20	satisfied	NOT satisfied	NOT satisfied
2	30	100	200	satisfied	NOT satisfied	NOT satisfied
3	50	100	200	satisfied	NOT satisfied	NOT satisfied
4	50	100	51	satisfied	NOT satisfied	NOT satisfied
5	60	100	0	satisfied	NOT satisfied	NOT satisfied
6	90	100	20	satisfied	NOT satisfied	NOT satisfied

Table 6.10: Gate Delays of a Eight Rail φ -Detector

There are no timing restrictions, because as shown in Figure 6.12 every input value is processed at least by one LUT before the output of these LUTs are connected to the input of the latch-building LUT.

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	$\delta_{feedback}$	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$			\forall not deadlock	$\exists \varphi0$ -change	$\exists \varphi1$ -change
1	10	100	100	20	satisfied	NOT satisfied	NOT satisfied
2	100	100	0	0	satisfied	NOT satisfied	NOT satisfied
3	100	100	0	10	satisfied	NOT satisfied	NOT satisfied
4	90	100	100	1	satisfied	NOT satisfied	NOT satisfied
5	90	100	20	105	satisfied	NOT satisfied	NOT satisfied
6	90	100	20	20	satisfied	NOT satisfied	NOT satisfied
7	90	100	20	95	satisfied	NOT satisfied	NOT satisfied

Table 6.11: Gate Delays of a Eight Rail φ -Detector with Feedback

6.5.3 Generic N-Rail φ -Detector

In this section an implementation rule is presented that holds regardless of the width of the input bus. Recall the LUT-schematic in Figure 6.11 where the main problem is the asymmetric processing of the input signals. As shown in the Uppaal result, this leads to a faulty behavior. Therefore, we propose the implementation as shown in Figure 6.13:

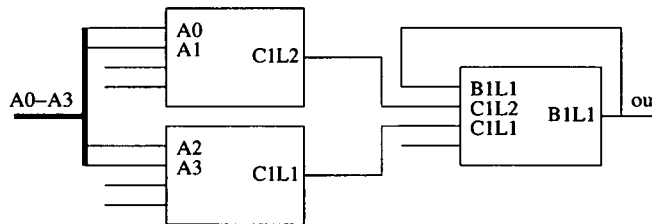


Figure 6.13: Two Signal wide Instance of a Generic φ -Detector

In contrast to Figure 6.11, inputs of two corresponding rails – former parts of one signal – are processed separately and in parallel to each other. So the difficulties described in Section 6.5.1 do not occur.

The principle for the generation of an n-signal wide φ -detector can be viewed in Figure 6.14. Every signal is handled by a separate XOR-gate. However, two input lines of each LUT are wasted.

Demonstrating the functionality, the results of an eight rail φ -detector are shown in Table 6.12.

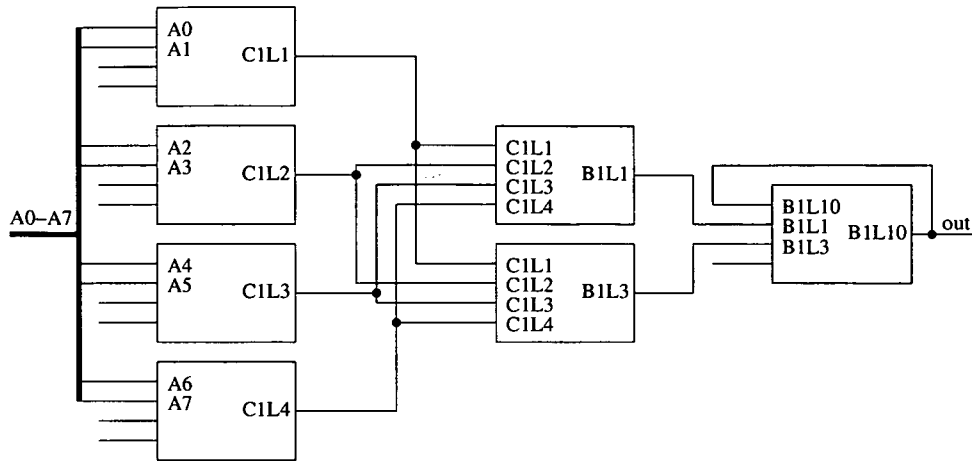


Figure 6.14: Four Signal wide Instance of a Generic φ -Detector

Nr.	$\Delta\delta_{LUT}$		δ_{idle}	$\delta_{feedback}$	Uppaal-results		
	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$			\forall not deadlock	$\exists \varphi_0$ -change	$\exists \varphi_1$ -change
1	10	100	100	20	satisfied	NOT satisfied	NOT satisfied
2	100	100	0	0	satisfied	NOT satisfied	NOT satisfied
3	100	100	0	10	satisfied	NOT satisfied	NOT satisfied
4	90	100	100	1	satisfied	NOT satisfied	NOT satisfied
5	90	100	20	105	satisfied	NOT satisfied	NOT satisfied
6	90	100	20	20	satisfied	NOT satisfied	NOT satisfied
7	90	100	20	95	satisfied	NOT satisfied	NOT satisfied

Table 6.12: Gate Delays of a 4-Signal Instance of a Generic φ -Detector with Feedback

6.6 Latch

6.6.1 One Signal wide Latch

Figure 6.15 depicts the compilation result of a one signal wide latch. The structure is quite simple. A latch for each rail is built using one LUT.

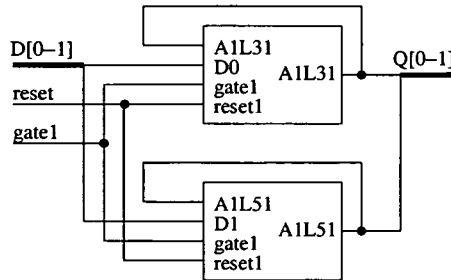


Figure 6.15: One Signal wide Latch Cell

Every input of the used LUT is used – one for the new input $D[i]$, ($i \in \{0, 1\}$), the `gate1` and `reset1` values and the last for the feedback constructing the latch. However, as a result of the design-flow the names for `reset` and `gate` have a "1" suffixed. In contrast to the models investigated so far, a latch has two types of inputs – the data signal and control rails. So the Uppaal-models have to be adapted to fulfill these new requirements. Therefore, a new parameter $\delta_{control}$ (`MIN_ST`) is introduced to control the timing behavior of the control signals. δ_{idle} is applied for delaying the data signal. The new stimuli generation template used with Uppaal is shown in Figure 6.16

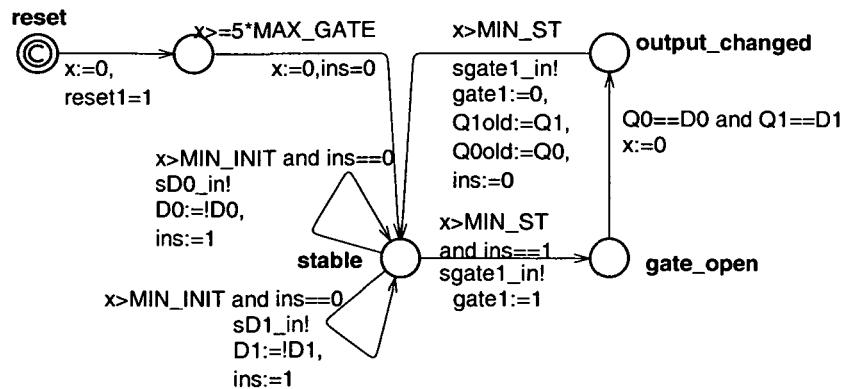


Figure 6.16: Stimuli Generation for an one Signal wide Latch Cell

Starting at the `reset` location, the reset will be cleared after a period of five $\delta_{LUT_{max}}$. The initial wait is required so that the system reaches the `stable` location, which is also the starting point of the operation loop. With the transition to this location the clock `x` is reset and so time measurements can be activated. The transitions positioned as a leaf with source and sink in location `stable` are used to alter one of the inputs after the minimal delay δ_{idle} has elapsed. After $\delta_{control}$ is passed the transition to `gate_open` is enabled, the assignments activate the gate, and the latch is now transparent. The model is now in the `gate opened` location. Now the system generates its new values. When those values are present to the output, the transition to `output passed` is enabled. Now the sequence closing the latch by deactivating the gate might be performed. The time $\delta_{control}$ must have elapsed before changing the gate rail, indicating to close the latch.

We test, if the latch is really closed and stores the value. This can be done with the query shown in Source 6.6.1 which means that the output of the latch must not change its value after gate is deactivated.

```
E<> p_stimuli_latch.stable and
(Q0 != p_stimuli_latch.Q0old or Q1 != p_stimuli_latch.Q1old)
```

Source 6.6.1: Uppaal-Query for the Latch Models

Several runs have been performed and just the significant ones are shown in Table 6.13. It is important to note that the result **satisfied** means that there is a possibility that the output changes in the stable state. So this is a negative result for the latch test.

Nr.	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$	δ_{idle}	$\delta_{control}$	$\delta_{feedback}$	\exists output-change
1	60	100	150	0	90	satisfied
2	60	100	150	10	0	NOT satisfied
3	60	100	150	10	10	NOT satisfied
4	60	100	150	10	70	satisfied
5	60	100	50	0	0	NOT satisfied
6	10	100	0	19	20	satisfied
7	10	100	0	20	20	NOT satisfied

Table 6.13: Gate Delays of one Signal wide Latch with Feedback

Recall Figure 6.15 to point out the problems: If the rail gate changes too early, the feedback-rails will not be able to update the input of the LUT. So the precedent output value is used for the calculation and the output is changed to the old value. The result can be outlined as:

$$\delta_{feedback} \leq \delta_{control} \tag{6.7}$$

Nevertheless, delaying the control signal gate with $\delta_{control}$ automatically holds up the inputs at the same value. The inputs can change immediately after the gate signal closes the latch.

6.6.2 Latch with Enable Logic

The next step is to merge one latch cell with the corresponding enable-logic as shown in Figure 6.17. The gate rail for the latch is now derived by the enable-logic. So a gate is replaced by **capture**, **capture_done** and **pass** as inputs to the entire latch cell (see Section 3.5.4).

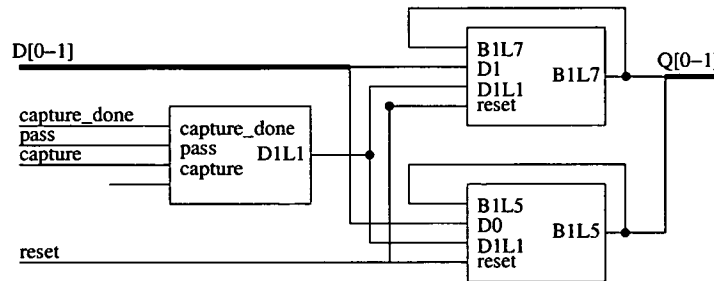


Figure 6.17: One Signal wide Latch Cell with Enable Logic

However, this requires the modification of the stimuli generation logic, as depicted in Figure 6.18: Starting with the model shown in Figure 6.16, a transition is added to

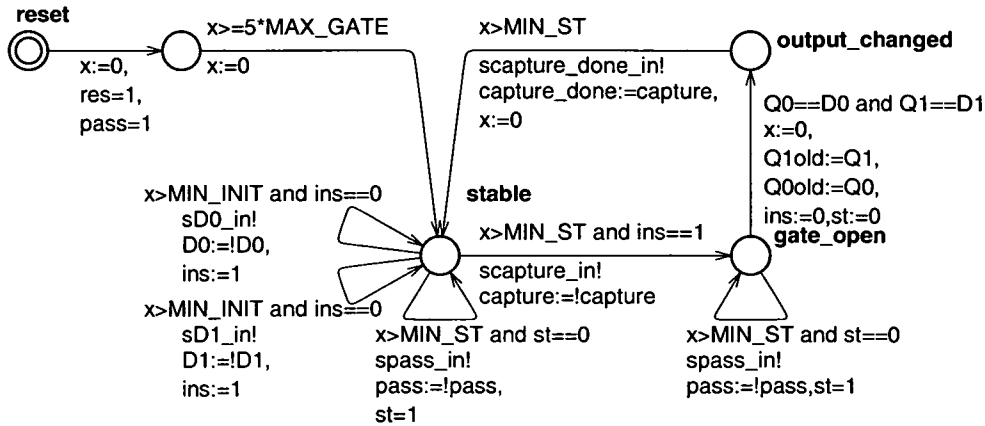


Figure 6.18: Stimuli Generation for a Latch Cell with Enable Logic

the `stable` location as well as to the `gate_open` location. This transition models the pass rail from the downstream register. It should be able to activate the pass-signal before the inputs are altered. This can be realized in location `stable`, or later. The transition leaving `stable` does not modify `gate` directly but controls the `capture` rail. In a real design this would be done by a φ -detector. In the same way, the transition from `output_changed` to `stable` alters not `gate` but `capture_done`.

Query 6.6.1 is used for the experiments and results that are shown in Table 6.14 (*NOT* satisfying the Uppaal-query means a working circuit):

Nr.	$\delta_{LUT_{min}}$	$\delta_{LUT_{max}}$	δ_{idle}	$\delta_{control}$	$\delta_{feedback}$	\exists output-change
1	60	100	100	0	0	NOT satisfied
2	60	100	150	0	10	NOT satisfied
3	60	100	150	0	70	<i>satisfied</i>
4	40	100	100	0	41	<i>satisfied</i>
5	40	100	100	1	41	NOT satisfied
81	80	100	200	0	0	NOT satisfied
84	80	100	200	0	90	<i>satisfied</i>
88	80	100	200	10	90	NOT satisfied
29	60	100	200	90	0	NOT satisfied

Table 6.14: Latch with Enable Cell and Feedback

It can be seen that δ_{idle} must be greater than $\delta_{LUT_{max}}$. In Figure 6.16 this claim can be viewed in detail. LUT D1L1 forms the enable logic and thus the new value for the gate rail – close the latch – required by the following LUTs must have been computed before one of the inputs changes. When this computation has completed, the inputs

can change immediately. Furthermore, the delay of D1L1 gives the feedback time to update the value on the input of the LUTs. If the feedback delay $\delta_{feedback}$ is less than the minimal gate delay $\delta_{LUT_{min}}$, there are no restrictions to the control signals. If this does not occur, then $\delta_{control}$ is needed to ensure a working circuit. Summarizing the results :

$$\delta_{feedback} \leq \delta_{LUT_{min}} + \delta_{control} \quad (6.8)$$

$$\delta_{idle} \geq \delta_{LUT_{max}} \quad (6.9)$$

6.7 Summary

In this chapter we have outlined construction conditions and requirements for the environment to guarantee a delay insensitive behavior.

We have shown that the AND and OR gate can be used without restrictions to the environment in a CAL-circuit, if the construction ensures that the difference between the minimal and maximal gate delay of a LUT ($\Delta_{\delta_{LUT}}$) is less or equal than the minimal gate delay ($\delta_{LUT_{min}}$). In general, the delay of a LUT on an FPGA does not vary in a wide range. If the LUTs can be placed at close quarters, wire delays connecting these LUTs will be in a close range and so the requirement will be fulfilled. Working Quartus, this can be enforced using so-called "lock-regions" where parts of a design can be placed and the composition is fixed and used several times. Furthermore, the INV-gate operates delay insensitive without any restrictions.

It has been shown that the main problem when constructing delay insensitive basic gates is an asymmetric structure of the gates. This can be especially seen in the φ -detector section where regulation of the structure leads to the desired behavior. This leads to a generic construction scheme for delay insensitive implementations of n signal wide φ -detectors.

Concerning CAL-latches, it is important to note that the delays of the control signals (capture, pass and capture_done) in relation to the feedback delay $\delta_{feedback}$ are critical. If the control signal meets its requirements elaboration in Section 6.6.2, input signals are not constrained.

All of these findings are based on the assumption that the gates are used in a pipeline as well as the handshake structure are fulfilling the CAL-rules. However, in Section 5 we have shown that the pipeline and $f(x)$ logic structure works correctly, if the basic gates fulfill their requirements.

This looks like a circular argument, but it is solved at the startup using the reset. With a reset, the pipes as well as the gates are set to a well-defined starting state where all CAL-rules defined in Section 3.2 are fulfilled².

²The setup and startup of pipelines is discussed in [22].

Chapter 7

Hardware Implementation: Asynchronous SPEAR

Contents

7.1	General Description	108
7.1.1	Design Migration Issues	108
7.2	Pipeline Improvements	109
7.2.1	Providing Latches with Different Initialization Values	110
7.2.2	"Capture Done Latches" in the Feedback Path	111
7.3	Adapting the Design-Flow	112
7.3.1	Pre-compiled Quartus Gates	113
7.3.2	Additional Target Library CALRAILLIB	114
7.3.3	Simulation Support	114
7.4	Implementation Results	115

7.1 General Description

The conversion of the synchronous SPEAR into an asynchronous design shall serve as a proof-of-concept for the CAL design technique as well as for the implementation constraints required for design of delay insensitive pipelines and basic gates outlined in Chapter 5 and 6. The resulting asynchronous processor represents an important platform for many types of experimental assessments.

7.1.1 Design Migration Issues

Starting point for our implementation is the synchronous processor SPEAR described in Chapter 4. The VHDL-sources describing SPEAR are reused to build its asynchronous counterpart ASPEAR. However, several modifications to the sources have been applied to construct a functioning asynchronous processor. Major changes concerning the coding style and the design entry have been applied to the following parts of the processor:

Data Types: In the design entry every signal data type has been changed from `std_logic` to `cal_logic`. Furthermore, logic operations that use constants like '0' have been adapted. There are functions to assist these modifications, e.g. the function `isnull()`, which is also defined for the `cal_logic` data type.

Inputs and Outputs: Inputs must be phase-aligned before being further processed, which is done with `std_logic` to `cal_logic` conversion functions. This process is similar to registered inputs in synchronous designs. The environment has no need for the phase. Here purely the value of the signal is relevant. Thus, the phase information is removed at the outputs.

Register and Pipeline: Traditional registers with `clock'event` cannot be used because of the absence of a clock. Every register must be transformed into handshake constructs. There is one big difference to the use of a global clock: The clock is the same throughout the whole clock domain. Thus, a unit does not need to know the logic units which provide the data. In our CAL approach data source and sink must be recognizable to establish a proper handshake. This information has to be added in order to transform a design from the synchronous style to CAL.

Memory: Synchronous memories cannot be used in CAL. However, it is not possible to use a memory with twice the capacity in order to connect the `cal_rail_logic` vectors instead of the original `std_logic` vector. The addresses in both phases, φ_0 and φ_1 , should access the same data word. Doubling the memory generates two memory cells for each phase and so separated values can be stored. The proposed solution is to remove the phase information from the address, search the value in a standard memory and rebuild the phase afterwards. However, this

is a bundled data approach, and delay elements have to be used to produce the new phase at the correct time. This proposed solution, which is used to design our processor is described in [22].

7.2 Pipeline Improvements

Recall the CAL pipeline example shown in Figure 3.4.2: Instead of using the `clock'event` clause, a component of a CAL-register is instantiated. The required width of this register as well as the initialization value after reset are both assigned as a generic. The different methods for building these registers, by defining the architecture, are the focus of this section. First of all, the entity of a `cal_reg` encapsulates all implementation details. From the designer's point of view it makes no difference, whether the functionality of the `cal_reg` component is provided by a Quartus pre-compiled gate or an architecture is used to build it – in any case the designer can simply instantiate it. This encapsulation makes it easy to change the architecture and compare different implementations.

First, a number of latches must be implemented in the register according to the bus width. This is elegantly performed in the architecture with a `for i in range` generate directive as shown in Source 7.2.1:

```
latches: for i in d'range generate
  init00: if reset_value = 00 generate
    latch_cell_proc_reset_00(gate => gate, reset => reset, D => d(i),
      Q => next_out(i));
  end generate init00;
  init01: if reset_value = 01 generate
    latch_cell_proc_reset_01(gate => gate, reset => reset, D => d(i),
      Q => next_out(i));
  end generate init01;
  init10: if reset_value = 10 generate
    latch_cell_proc_reset_10(gate => gate, reset => reset, D => d(i),
      Q => next_out(i));
  end generate init10;
  init11: if reset_value = 11 generate
    latch_cell_proc_reset_11(gate => gate, reset => reset, D => d(i),
      Q => next_out(i));
  end generate init11;
end generate latches;
```

Source 7.2.1: Register Implementation in CAL

The `for` clause generates `d'range` instances of a procedure and connects them to the common reset and gate signal. Which procedure is selected depends on the desired

initialization value after reset. This is discussed in 7.2.1. Furthermore, a φ -detector with the required width is generated and instantiated.

7.2.1 Providing Latches with Different Initialization Values

In synchronous designs the initialization value after reset is defined with the `if (reset=ACT)` clause in the design process. An appropriate method is also needed for CAL designs. Since in CAL are four different values and as shown in [22] a simple initialization value is generally not sufficient, there must be a possibility to initialize a register with one of the four values. An obvious solution is adding a signal with the desired initial value. However, this has an unintended side-effect in our implementation with Apex LUT's: To build one latch we already need all four inputs of a LUT: One for the input data value, the gate, the reset, and finally the feedback. The initial reset value would be the fifth input and so every latch would demand two LUTs to be constructed instead of one. This is a waste of resources, further it makes the timing behavior much more difficult to handle.

Our solution is based on the fact that the initial value is defined at design time and will not be altered afterwards. The solution is clear: We designed four different latch types – one for each possible start value. During the creation of the register the appropriate latch is selected in order to provide the desired initial value. This can be viewed in Source 7.2.1: The body of the `for . . . generate` contains `if`-clauses where the appropriate latch for the given `reset_value` is selected. As mentioned above, in this step a procedure is mapped into the design. One of these procedures is shown in Source 7.2.2 and will be described in detail later on.

```

procedure latch_cell_proc_reset_11 ( signal gate, reset : in  cal_ctrl;
  signal D: in  cal_logic; signal Q : out cal_logic) is
begin
  -- pragma map_to_entity LATCH_CELL_RESET_11
  -- synopsys synthesis_off
  if reset = RESETECT then
    Q <= 'h';
  elsif gate = CAL_LATCH_ENABLE then
    Q <= D after 10 ns;
  end if;
  -- synopsys synthesis_on
end;
```

Source 7.2.2: Register Cell Implementation in CAL

The procedure shown above provides "HIGH" in phase φ_0 ("h") as start value and describes the behavior of this latch for the simulation: The output is set to "h" in the case of reset, otherwise the output Q is set to the value of the input D, if the gate is enabled. This assignment is delayed for 10 ns for simulation purposes. To

use pre-compiled gates this code is enclosed with the `synopsys synthesis_off` and `synopsys synthesis_on` paradigms. This forces Synopsys to pass the code without synthesis. Furthermore, with the `pragma map_to_entity LATCH_CELL_RESET_11` directive Synopsys is instructed to use the entity `LATCH_CELL_RESET_11`, instead of the functionality of the code. Thus, a pre-compiled component can be selected and there is no need to synthesize the code of the procedure. As shown in Section 7.3.1, these four latch implementations are part of the library `CALRAILLIB`.

7.2.2 "Capture Done Latches" in the Feedback Path

One major finding of Chapter 6 is the demand of latching the capture done signal. Thus, the insertion of the reverse latches in the pipeline feedback path is performed automatically while generating the register. Source 7.2.3 illustrates this step:

```
cdone_latch0: if reset_value = 00 or reset_value = 11 generate
  latch_cell_proc_reset_00(gate => inv_gate, reset => reset, D => c_done_cal,
    Q => c_done_cal_next);
  end generate cdone_latch0;
cdone_latch1: if reset_value = 01 or reset_value = 10 generate
  latch_cell_proc_reset_11(gate => inv_gate, reset => reset, D => c_done_cal,
    Q => c_done_cal_next);
end generate cdone_latch1;
```

Source 7.2.3: Capture Done Latch Implementation

There are two instantiations shown in Figure 7.2.3, however, only one of them is really used during the generation step. It depends on the initialization value after reset to find out which of the latches will be built. The reason for this conditioned instantiation is the need to match the output value of the register with the capture done signal. If the initial output of register is in phase $\varphi 0$ (the first instance in Source 7.2.3), capture done must be $\varphi 0$ too. Respectively, if the register starts with outputs in $\varphi 1$, capture done is set to $\varphi 1$.

An implementation example using a four signal wide register with initial value "HIGH" in phase $\varphi 1$ is depicted in Figure 7.1¹:

On the CAL level there are four latches used to store the values of the bus D. Notice, this bus is four signals (of type `cal_logic`) wide, therefore the actual hardware consists of eight rails and eight latches. In the middle of Figure 7.1, the two φ -detectors are shown: One of them is connected to the input, the other one is used to determine the phase of the output. The output of these φ -detectors, as well as the pass signal are used by the `enable logic` to calculate the `gate` signal. This control signal is directly

¹Due to the fact that Figure 7.1 is a Synopsys screenshot, there is no possibility to influence the layout. Unfortunately, one latch has been moved to the inputs on the left side.

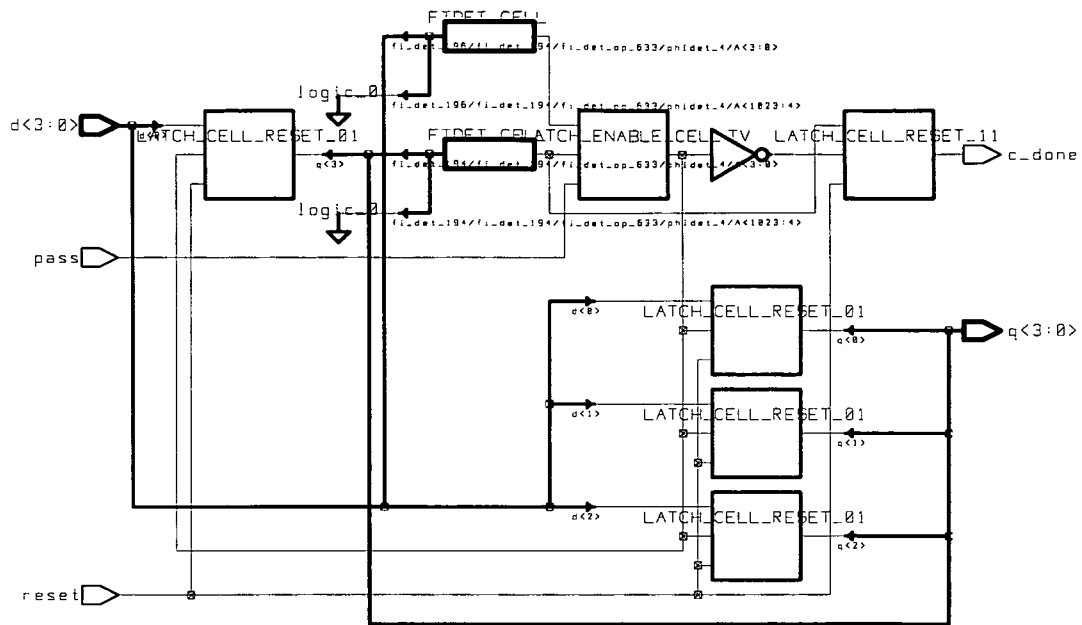


Figure 7.1: Implementation Result of a Four Signal wide Register

used by the four latches and further in its inverted form by the latch controlling the capture done signal.

7.3 Adapting the Design-Flow

The findings from Chapter 5 and 6 have impacts on the CAL design flow. Thus, the design flow presented in Section 3.6 is adapted to fulfill the new requirements. Figure 7.2 depicts the improved CAL design flow.

The beginning of the improved design flow is equal to the "normal" CAL design flow. The first synthesis to transform `cal_logic` gates to `cal_rail_logic` is performed as in the normal design flow. However, the improved design flow uses an additional library during the second synthesis. The `CALRAILLIB` is used as an additional `target library` (see Section 7.3.2) to introduce the symmetric gates as desired in Chapter 6. The resulting design descriptions comprise not only cells from the APEX library, but also gates from `CALRAILLIB`. Therefore, it is necessary to add simulation models of the pre-compiled Quartus gates to be able to perform the pre-layout simulation (see Section 7.3.3). Furthermore, the gates described in this library must be provided to the place & route step using Quartus (`procomp. basic gates`) in Figure 7.3). The result of Quartus is similar to all three design flows and is shown in Figure 7.2: The logic design is represented by a downloadable file for programming the FPGA. Furthermore, a simulation model is produced, which only consists of gates provided by the target

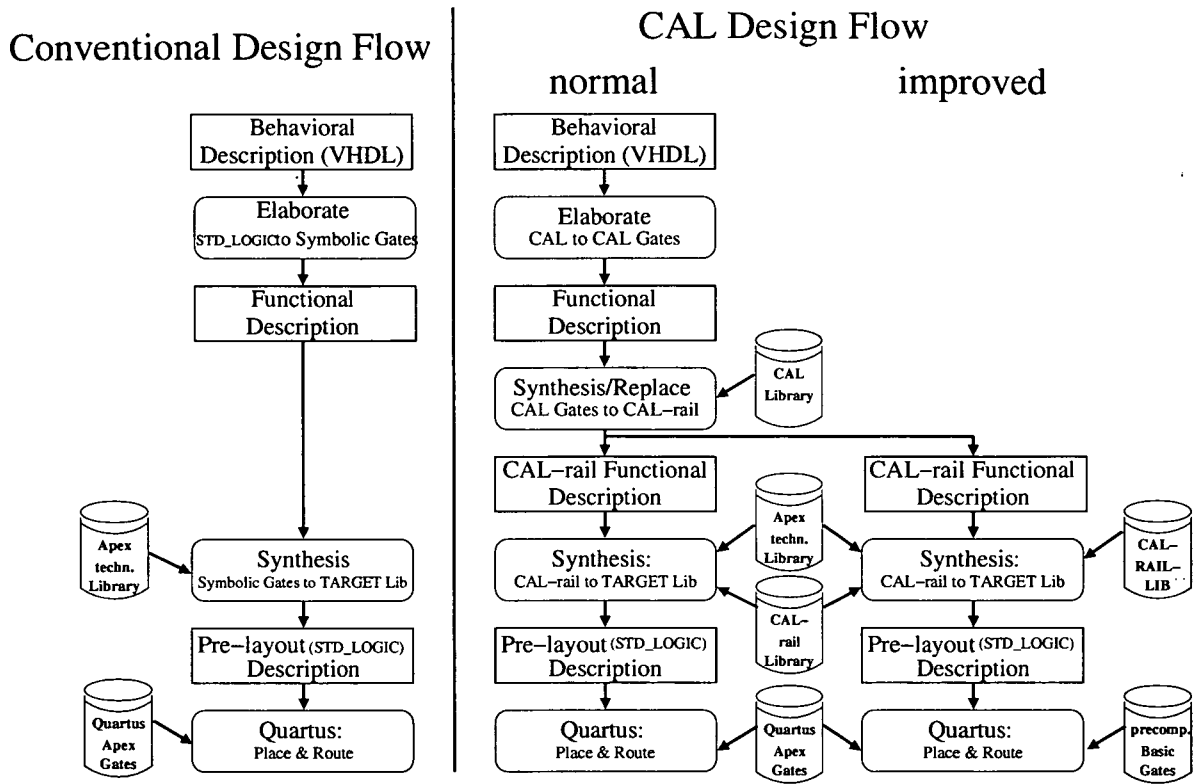


Figure 7.2: Improved CAL Design Flow

gate-level library, the so-called VITAL²[124, 125]-library.

7.3.1 Pre-compiled Quartus Gates

One of the main outcomes of Chapter 6 is the need for symmetric basic gates. The proposed solution is to once compile these gates in Quartus carefully considering all constraints and then using them as pre-compiled gates in CAL-designs.

The proposed basic gates and register components are compiled with Quartus from scratch using the behavioral description available in VHDL-sources. Pre-compiled gates are implemented for the following entities:

- Basic Gates: AND, OR, and INV gates
- Register Components: Latches (four types with the four possible initialization values), latch enable cell
- Conversion Functions: TO_CAL_LOGIC_CELL

²VHDL Initiative Towards ASIC Libraries

The resulting hardware is transferred to a verilog Quartus mapping(.vqm) file and can be used in subsequent designs. In Figure 7.3 the box on the left side marks the usage of a pre-compiled AND-gate.

7.3.2 Additional Target Library CALRAILLIB

The library CALRAILLIB is used to provide Synopsys with the names and the types of the gates which are implemented as pre-compiled Quartus gates. Using this information Synopsys is able to regard these components as parts of the target library. Thus, no further activities occur and instances of these gates are placed in the design. Source 7.3.1 lists the definition of the library element of an AND-gate.

```

cell(AN2) {
  area : 2;
  bus (A){
    bus_type : bus2;
    direction : input;
    capacitance : 1;
  }
  bus (B){
    bus_type : bus2;
    direction : input;
    capacitance : 1;
  }
}

bus(Z) {
  bus_type : bus2;
  direction : output;
  timing() {
    ...
    related_bus_pins : "A";
  }
  timing() {
    ...
    related_bus_pins : "B";
  }
}

```

Source 7.3.1: Library Definition of an AND-Gate

However, these gates are only used to force Synopsys to utilize the pre-compiled gates in the design. Therefore, the `area` value is set to a very small value in order to pretend to Synopsys that these gates are very small and economical to use.

Source 7.3.2 shows the result of performing the `report_lib` command in Synopsys:

The attribute `b` denotes that these cells are only black boxes. Thus, they are used as place holders and the required functionality is provided by the pre-compiled gates.

7.3.3 Simulation Support

As mentioned in Section 7.3, the pre-layout description of the design contains only instantiations of APEX-cells (e.g. LUTs) and pre-compiled cells. The simulation model of the APEX-cells is provided by a library shipped with Quartus. This information is also required for our pre-compiled gates.

During the synthesis and place & route steps, Quartus generates appropriate VHDL models for the resulting hardware. Furthermore, a standard delay description file (.sdf) is produced, which contains the adequate timing information for the VHDL model.

Cell	Attributes
AN2	b
IV	b
LATCH_CELL_RESET_init_value0	b
LATCH_CELL_RESET_init_value1	b
LATCH_CELL_RESET_init_value10	b
LATCH_CELL_RESET_init_value11	b
LATCH_ENABLE_CELL	b
OR2	b
TO_CAL_LOGIC_CELL	b

Source 7.3.2: Result of Synopsys Library Report: CALRAILLIB-Members

This information is collected for all pre-compiled gates and stored in a library, which can be used without difficulty as additional target gate library.

Nevertheless, fundament of this new library is the APEX-cell library mentioned above. All new gates defined in the new library are composed of elements of the APEX-cell library, which makes it easy to change the target technology.

7.4 Implementation Results

Figure 7.3 displays the final report performing the place & route step with Quartus. The target technology for the implementation of the asynchronous SPEAR is the APEX EP20K1000C FPGA.

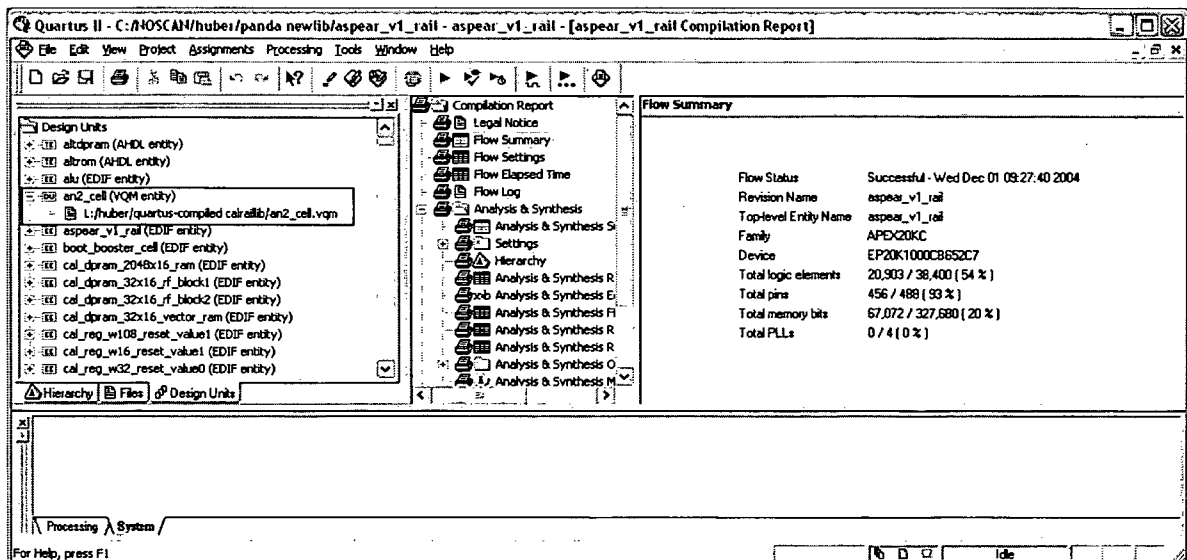


Figure 7.3: Implementation Results of ASPEAR

The proposed implementation utilizes about 20,900 logic elements, which is eleven times more than its synchronous counterpart, described in Section 4.1.3. In addition, the processor executes the instructions comparable to a synchronous SPEAR running with 2.5 MHz. Both benchmark data are not really outstanding, however, it is a first successful approach implementing SPEAR with the asynchronous CAL-methodology and many optimizations are still possible.

Chapter 8

Conclusion

In this thesis an introduction into Code Alternation Logic (CAL) has been given. A dense code has been utilized where two representations of the logic values "LOW" and "HIGH" – one for phase φ_0 and one for phase φ_1 – are defined. Furthermore, the human interface to build CAL circuits – `cal_logic` – as well as the coding style on gate level – `cal_rail_logic` – have been introduced. The methodology and the required libraries for the CAL design flow demonstrate the automated flow from the design written by the engineer to the final download file for an ApexFPGA, which is used as a prototyping environment in our case. The CAL approach is similar to NCL with some essential advantages: There is no need for the so-called spacer or NULL-waves in CAL, which doubles the throughput compared with NCL. Furthermore, exactly one rail transition per bit is required, which reduces the energy overhead in terms of transitions per bit.

On a high level visible to the designer, CAL claims to be delay insensitive. It was a central aim of this thesis to investigate, whether this claim is actually valid. This investigation was performed by means of formal verification, in particular with the model-checker Uppaal. Three rules have been identified, which must be obeyed to ensure a properly working circuit. These rules have been translated into corresponding queries, which have been reassessed with Uppaal. In our analysis we decided in favor of a top-down approach in order to analyze the timing behavior of CAL. Pipeline structures with combinational functions, as well as the basic gates were transferred into Uppaal models in order to check our queries. The *findings in theory* can be summarized as follows:

- The schematic CAL pipeline is used to test the handshake sequence with queries. We have formally proven that this model of a pipeline fulfills the CAL rules without restrictions.
- A gradual refinement of the pipeline model, however, has shown deficiencies with respect to delay insensitivity. Synchronizing the capture done signal has turned out as a possible solution and this fact is assured by Uppaal results using a theoretical model.

- Furthermore, it has been verified that combinational functions $f(x)$ that only use basic gates fulfill the rules for CAL circuits. We have shown that there are no restrictions for building CAL circuits utilizing only basic gates.
- The basic gates internal delay assumptions are made, yielding to design constraints. These constraints solve parts of the fundamental design problem in the time domain.

We have derived and verified with Uppaal models the following condition that must be ensured by an FPGA-implementation of AND and OR gates: The implementation must guarantee that the difference between the minimal and maximal gate delay of one LUT ($\Delta_{\delta_{LUT}}$) is less or equal to the minimal gate delay ($\delta_{LUT_{min}}$).

$$\Delta_{\delta_{LUT}} \leq \delta_{LUT_{min}}$$

Inversion (INV) gates operate delay insensitive without any restrictions, due to their simple layout.

- The analysis of the CAL latches within the CAL registers pointed out that the delays of control signals (`capture`, `pass` and `capture_done`) in relation to the feedback delay $\delta_{feedback}$ are critical. The results concerning a single latch-cell can be outlined as:

$$\delta_{feedback} \leq \delta_{control}$$

The only restriction on the latch is that the feedback delay ($\delta_{feedback}$) is smaller than the interval between a transition on the output and the first resulting change of one of the control signals (indicated by $\delta_{control}$). Especially, there are no additional constraints on the input signals for this latch. Furthermore, details of the enable logic have been added to the model and this leads to following findings:

$$\delta_{feedback} \leq \delta_{LUT_{min}} (+\delta_{control})$$

$$\delta_{idle} \geq \delta_{LUT_{max}}$$

- All of these findings are based on the assumption that the gates, which are used in a pipeline and handshake structure, fulfil the CAL rules. However, the basic gates rely on the correct behavior of the pipelines and the combinational logic. This looks like a circular argument, but it can be solved at the startup using the reset. During a reset all pipes, as well as the gates are set in a well-defined starting state where all CAL rules defined in Section 3.2 are fulfilled¹.

In summary, CAL can be classified as a hybrid solution to manage the fundamental design problem. On a high abstraction level CAL designs are considered, which are solely built with CAL basic gates. The fundamental design problem is tackled in the

¹The startup issues of pipelines are discussed in [22].

information domain. Therefore, validity and consistency of the data are guaranteed. As our formal verification has proven, a delay insensitive behavior at this abstraction level is indeed ensured. The CAL basic gates themselves manage the fundamental design problem in the time domain. Implementation constraints that are necessary to guarantee a behavior that is compliant with the CAL design rules have been identified. The *practical applications of the findings* can be described as follows:

- As mentioned above, the implementation of an AND/OR gate must guarantee that the gate delay of every LUT is less or equal to the minimal gate delay ($\Delta_{\delta_{LUT}} \leq \delta_{LUT_{min}}$). However, in general the delay of a LUT on an FPGA does not vary in a wide range. With Quartus this can be enforced using so-called "lock-regions", where parts of a design can be placed manually and the composition is fixed and used several times. Accordingly, we have provided suitable pre-compiled library elements for our CAL design flow.
- To synchronize the capture done signal, we have proposed latching this signal with the inverted enable signal as the hardware solution. However, this introduces two additional conditions for the implementation of registers:

$$t_{w_o} < \delta_{latch} \quad \text{and} \quad t_{w_c} < \delta_{latch}$$

The wire delay from the enable logic to the data latch cell (t_{w_o}) as well as the delay from the enable logic to the capture done latch cell (t_{w_c}) must each be smaller than the minimal latch delay (δ_{latch}). In Figure 8.1 the relevant delays for constructing a latch are shown:

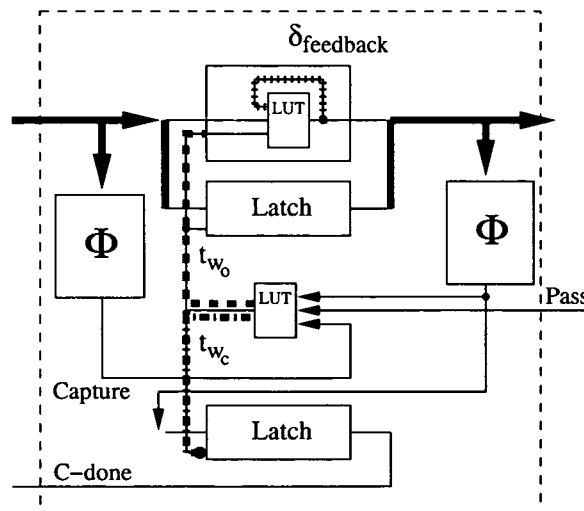


Figure 8.1: CAL-Register Delay Summary

The constraints to build a latch must also be considered. Thus, all latches work properly, if all three delays ($\delta_{feedback}$, t_{w_o} , and t_{w_c}) are less than the delay of one

LUT. Furthermore, δ_{idle} defines the time between the transition of the output and the first alteration of the input. Due to the fact that the enable control logic, the capture done latch, and a φ -detector are in the path of the capture done signal, it is ensured that $\delta_{idle} \geq \delta_{LUT_{max}}$. We propose to build these CAL registers once considering the above conditions and then provide them as basic gates.

- It has been shown that the main problem for constructing delay insensitive basic gates is the asymmetric structure of these gates. This can be observed for the φ -detector, where a regulation of the structure enabled us to obtain the desired properties. This leads to a generic construction scheme for delay insensitive implementations of n-signal wide φ -detectors.

The successful implementation of the asynchronous ASPEAR counterpart to our SPEAR-processor demonstrates the applicability of our CAL approach as well as the correctness of the ALTERA Apex FPGA implementation of the basic gates.

Further prospects

In this work we have performed a complete and detailed analysis of the delay insensitive behavior of CAL-based circuits and identified the crucial issues. Still, a lot of interesting ideas remain that could not be pursued within the scope of this thesis. Among these are:

Extension of the CAL library: The basic gates are the target of the first synthesis.

Thus, this synthesis constructs the whole design only using these gates. Although, every logic design can be built out of AND, OR, and INV gates, the result might not be really satisfying. On the one hand, it is useful to create gates performing other basic logic operations, e.g. XOR or NAND, which can be further utilized by the synthesis tool to gather better results. On the other hand, basic gates optimized for more than two inputs may also increase the synthesis result. There are many supposable extensions, e.g. four input AND-/OR-gates or special two output gates to support the requirements of building φ -detectors.

Optimization with respect to area as well as to power: Our implementation in an FPGA is a proof of concept and so there is a great potential for optimizations. As mentioned before in this thesis, Apex FPGAs are optimized for synchronous designs, e.g. registered outputs of a LUT are not useable in our CAL-approach. Therefore, an ASIC implementation of our basic gates will enhance the performance and will decrease the performance gap between the synchronous SPEAR and the asynchronous ASPEAR.

ASIC Implementation: We have shown the delay insensitive behavior on a high abstraction level. Furthermore, the constraints to build the basic gates have been derived in the previous sections. Some of the implementation related arguments

concerning the delay assumptions have been focused on the FPGA implementation that is based on LUTs. Although, the basic findings will still be held in an ASIC implementation. The rules for the basic cell design will have to be adapted.

Theoretical and practical findings elaborated in this thesis serve as a basis for further investigations in the domain of delay insensitive, asynchronous designs using the code alternation logic approach.

Bibliography

- [1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, Reading, MA, USA, 1988.
- [2] Apex 20kc programmable logic device data sheet.
http://www.altera.com/literature/ds/ds_apex20kc.pdf, February 2004.
- [3] Altera, 101 Innovation Drive, San Jose, CA. *Quartus II Version 5.0 Handbook*, 2005.
http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf.
- [4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994/4/25.
- [5] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, April 1997.
- [6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer-Verlag, September 2004.
- [7] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer-Verlag, October 1995.
- [8] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming and silicon compilation. In *Proc. International Conf. Computer Design (ICCD)*, pages 150–166, Rye Brook, New York, 1988. IEEE Computer Society Press.
- [9] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.
- [10] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

- [11] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag, 1995.
- [12] J.A. Brzozowski and S. Singh. Definite asynchronous sequential circuits. *IEEE Transactions on Computers*, C-17(1):18–26, January 1968.
- [13] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.
- [14] Tam-Anh Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *Proc. International Conf. Computer Design (ICCD)*, pages 220–223. IEEE Computer Society Press, 1987.
- [15] Alexandre David, Oliver Möller, and Wang Yi. Formal verification of UML statecharts with real-time extensions. In Ralf-Detler Kutsche and Herbert Weber, editors, *Proceedings of FASE 2002*, number 2306 in Lecture Notes in Computer Science, pages 218–232. Springer-Verlag, 2002.
- [16] Ilana David, Ran Ginosar, and Michael Yoeli. An efficient implementation of boolean functions as self-timed circuits. *IEEE Transactions on Computers*, 41(1):2–11, January 1992.
- [17] Ilana David, Ran Ginosar, and Michael Yoeli. Implementing sequential machines as self-timed circuits. *IEEE Transactions on Computers*, 41(1):12–17, January 1992.
- [18] A. Davis and S. M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, University of Utah, Department of Computer Science, 1997.
- [19] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Dept. of Computer Science, University of Utah, September 1997.
- [20] Mark Dean, Ted Williams, and David Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In Carlo H. Séquin, editor, *Advanced Research in VLSI*, pages 55–70. MIT Press, 1991.
- [21] Martin Delvai. Handbuch für SPEAR (Scalable Processor for Embedded Applications in Real-Time Environments). Research Report 70/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
- [22] Martin Delvai. *Design of an Asynchronous Processor Based on Code Alternation Logic - Treatment of Non-Linear Data Paths*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2005.
- [23] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor Support for Temporal Predictability - The SPEAR Design Example. In *Proc. 15th Euromicro International Conference on Real-Time Systems, Porto, Portugal*, 2003.

- [24] Martin Delvai, Wolfgang Huber, Babak Rahbaran, and Andreas Steininger. SPEAR - Design-Entscheidungen für den "Scalable Processor for Embedded Applications in Real-Time Environments". In *Proc. Austrochip 2001, Vienna, Austria*, 2001.
- [25] AMD Advanced Micro Devices. www.amd.com.
- [26] AMD Advanced Micro Devices. Amd PowerNow Technology. http://www.amd.com/us-en/assets/content_type/DownloadableAssets/Power_Now2.pdf, 2002.
- [27] Webmaster Dictionary. Moore's law. <http://www.webster-dictionary.org/definition/Moore's%20Law>.
- [28] R. Dobkin, R. Ginosar, and C. P. Sotiriou. Data synchronization issues in GALS SoCs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 170–179. IEEE Computer Society Press, April 2004.
- [29] Paul Stanford (ed.). Electronic design interchange format version 2 0 0, ansi/eia-548-1988, recommended standard eia-548, March 1988.
- [30] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9:90–99, March 1965.
- [31] The El Camino Homepage. <http://www.elca.de>.
- [32] Digilab megAPEX manual - apex 20k high-end prototyping system. [http://www.elca.de/Downloads/Manual Digilab megAPEX.pdf](http://www.elca.de/Downloads/Manual%20Digilab%20megAPEX.pdf), January 2003.
- [33] Karl M. Fant and Scott A. Brandt. Null convention logic system. US patent Nr. 5,305,463, April 1994.
- [34] Karl M. Fant and Scott A. Brandt. Null convention logic: A complete and consistent logic for asynchronous digital circuit synthesis. In *Proc. International Conference on Application Specific Systems, Architectures and Processors*, pages 261–273, August 1996.
- [35] Farlex. The free dictionary. <http://www.intel.com/products/processor/index.htm>.
- [36] C. Foley. Characterizing metastability. In *Advanced Research in Asynchronous Circuits and Systems*, pages 175 – 184, March 1996.
- [37] A. D. Friedman and P. R. Menon. Synthesis of asynchronous sequential circuits with multiple-input changes. *IEEE Transactions on Computers*, C-17(6):559–566, June 1968.
- [38] Gottfried Fuchs. A superscalar 16 bit microcontroller for real-time applications. Master's thesis, Technische Universität Wien, 2003.
- [39] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, and J. V. Woods. The design and evaluation of an asynchronous microprocessor. In *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, October 1994.

- [40] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, April 1997.
- [41] Stephen B. Furber, James D. Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, and Nigel C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, February 1999.
- [42] Jim Garside. The Asynchronous Logic Homepage.
<http://www.cs.man.ac.uk/amulet/async/>.
- [43] David Geer. Is it time for clockless chips? *IEEE Computer*, 38(3):18–21, 2005.
- [44] Mark R. Greenstreet and Brian de Alwis. How to achieve worst-case performance. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 206–216. IEEE Computer Society Press, March 2001.
- [45] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.
- [46] Scott Hauck, Steven Burns, Geatano Borriello, and Carl Ebeling. An FPGA for implementing asynchronous circuits. *IEEE Design & Test of Computers*, 11(3):60–69, 1994.
- [47] John L. Hennessy and David A. Patterson. *Computer Organization and Design*. Morgan Kaufmann Publisher, Inc., 1994.
- [48] M. Hevery. Asynchronous circuit completion detection by current sensing. In *Twelfth Annual IEEE International ASIC/SOC Conference*, pages 322–326, 1999.
- [49] Zhijun Huang and M. D. Ercegovac. Effect of wire delay on the design of prefix adders in deep-submicron technology. In *Conference on Signals, Systems and Computers, 2000*, October 2000.
- [50] Wolfgang Huber. Spezifikation der Schnittstelle zwischen Extension-Modulen und SPEAR. Technical report, Institute of Computer Engineering , VLSI - Design, Vienna, 2001.
- [51] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, March/April 1954.
- [52] Intel. www.intel.com.
- [53] Intel. Mobile intel pentium iii processors featuring intel speedstep technology.
http://www.intel.com/mobile/resources/downloads/pdf/P3P_fn.pdf, 2001.
- [54] Martin Jankela, Wolfgang Puffitsch, and Wolfgang Huber. Towards a rapid prototyping framework for architecture exploration in embedded systems. In *Proc. Workshop on Intelligent Solutions in Embedded Systems*, pages 117–128, Graz, Austria, June 2004.

- [55] Mark B. Josephs, Steven M. Nowick, and C. H. (Kees) van Berkel. Modeling and design of asynchronous circuits. *Proceedings of the IEEE*, 87(2):234–242, February 1999.
- [56] H. J. Kahn and R. F. Goldman. The electronic design interchange format EDIF: present and future. In *DAC '92: Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 666–671, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [57] Yun Y. Kenneth. Recent advances in asynchronous design methodologies. In *Asia and South Pacific Design Automation Conference 1999 (ASP-DAC'99)*, pages 253–259, January 1999.
- [58] Joep Kessels and Paul Marston. Designing asynchronous standby circuits for a low-power pager. *Proceedings of the IEEE*, 87(2):257–267, February 1999.
- [59] Miloš Krstić and Eckhard Grass. New GALs technique for datapath architectures. In Jorge Juan Chico and Enrico Macii, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, volume 2799 of *Lecture Notes in Computer Science*, pages 161–170, September 2003.
- [60] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, December 1995.
- [61] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in *Lecture Notes in Computer Science*, pages 62–88, August 1995.
- [62] Steven S. Leung and Michael A. Shanblatt. *ASIC System Design with VHDL: A Paradigm*. Kluwer Academic Publishers, 1990.
- [63] Hai Li, S. Bhunia, Y. Chen, T. N. Vijaykumar, and K. Roy. Deterministic clock gating for microprocessor power reduction. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003, HPCA-9 2003*, pages 113–122. IEEE Computer Society Press, February 2003.
- [64] Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, and Alex Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 114–125. IEEE Computer Society Press, April 2000.
- [65] D. W. Lloyd and J. D. Garside. A practical comparison of asynchronous design styles. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 36–45. IEEE Computer Society Press, March 2001.
- [66] G. Magó. Realization methods for asynchronous sequential circuits. *IEEE Transactions on Computers*, C-20(3):290–297, March 1971.
- [67] K. Maheswaran. Implementing self-timed circuits in field programmable gate arrays. Master's thesis, University of California, Davis, 1994.

- [68] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.
- [69] Alain J. Martin. Formal program transformations for VLSI circuit synthesis. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, UT Year of Programming Series, pages 59–80. Addison-Wesley, 1989.
- [70] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.
- [71] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278. MIT Press, 1990.
- [72] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The first asynchronous microprocessor: the test results. *Computer Architecture News*, 17(4):95–110, June 1989.
- [73] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Péntzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.
- [74] Alain J. Martin, Mika Nyström, Paul Péntzes, and Catherine Wong. Speed and energy performance of an asynchronous MIPS R3000 microprocessor. Technical Report CSTR:2001.012, California Institute of Technology, 2001.
- [75] Doug Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, 1997.
- [76] Anthony J. McAuley. Dynamic asynchronous logic for high-speed CMOS systems. *IEEE Journal of Solid-State Circuits*, 27(3):382–388, March 1992.
- [77] Anthony J. McAuley. Four state asynchronous architectures. *IEEE Transactions on Computers*, 41(2):129–142, February 1992.
- [78] John McCardle and Dr. David Chester. Measuring an asynchronous processor's power and noise. In *Synopsys Users Group Boston*, 2001.
- [79] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [80] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.
- [81] G.E. Moore. No exponential is forever: but "forever" can be delayed! [semiconductor industry]. In *Solid-State Circuits Conference, 2003*, volume 1, pages 20–23, 2003.
- [82] Gordon E. Moore. The experts look ahead: Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

- [83] David E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1962.
- [84] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.
- [85] Chris Myers. *Asynchronous Circuit Design*. John Wiley & Sons, 2001.
- [86] Chris J. Myers, Wendy Belluomini, Kip Killpack, Eric Mercer, Eric Peskin, and Hao Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, February 2001.
- [87] Christian Dalsgaard Nielsen, Jørgen Staunstrup, and Simon Jones. A delay-insensitive neural network engine. In Will R. Moore, editor, *Proceedings of the Workshop on VLSI for Neural Networks*, pages 367–376, September 1991.
- [88] L. S. Nielsen, C. Niessen, J. Sparsø, and C. H. van Berkel. Low-power operation using self-timed and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391–397, December 1994.
- [89] L. S. Nielsen and J. Sparsø. An $85\mu\text{W}$ asynchronous filter-bank for a digital hearing aid. In *International Solid State Circuits Conference*, February 1998.
- [90] Lars S. Nielsen and Jens Sparsø. Designing asynchronous circuits for low-power: An IFIR filter bank for a digital hearing aid. *Proceedings of the IEEE*, 87(2):268–281, February 1999.
- [91] M. Olivieri. Translating occam constructs into delay-insensitive circuits: a trace theory-based proof. Technical Report CPSI92-1, Dept. of Biophys. and Electronic Eng., Univ. of Genoa, Italy, 1992.
- [92] M. Olivieri. Design of synchronous and asynchronous variable-latency pipelined multipliers. *IEEE Transactions on VLSI Systems*, 9(2), May 2001.
- [93] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low-noise configurable self-timed DSP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 32–42, 1998.
- [94] R. E. Payne. Self-timed FPGA systems. In W. Moore and W. Luk, editors, *Fifth International workshop on Field Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 21–35, 1995.
- [95] Ad Peeters. The ‘Asynchronous’ Bibliography (BIB \TeX) database file `async.bib`. <http://www.win.tue.nl/async-bib/doc/async.bib>. Corresponding e-mail address: `async-bib@win.tue.nl`.
- [96] Christian Piguet. Logic synthesis of race-free asynchronous CMOS circuits. *IEEE Journal of Solid-State Circuits*, 26(3):371–380, March 1991.

- [97] Wolfgang Puffitsch and Wolfgang Huber. Porting the GNU Compiler Collection to the SPEAR microprocessor. Research Report 24/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.
- [98] P. Puschner and A. Burns. Writing temporally predictable code. In *Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems*, San Diego, California, USA, January 2002.
- [99] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Tech. Rep. 120, Massachusetts Inst. of Tech., February 1974.
- [100] R.B. Reese and S.B. Sikandar-Gani. Control versus compute power within a LEDR-style self-timed multiplier with bypass path. In *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on*, volume 2, pages II-302-II-305, 2002.
- [101] Robert B. Reese, Mitch A. Thornton, and Cherrice Traver. Arithmetic logic circuits using self-timed bit level dataflow and early evaluation. In *Proc. International Conf. Computer Design (ICCD)*, pages 18-23, November 2001.
- [102] Arnold Robbins. *Effective AWK Programming*. Specialized Systems Consultants, P.O. Box 55549, Seattle, WA 98155, 1996.
- [103] Charles L. Seitz. Asynchronous machines exhibiting concurrency, 1970. Record of the Project MAC Concurrent Parallel Computation.
- [104] International SEMATECH. International technology roadmap for semiconductors, 2003 edition.
<http://public.itrs.net/Files/2003ITRS/Home2003.htm>, 2003.
- [105] N. Shintel and M. Yoeli. Synthesis of modular networks from Petri-net specifications. Technical Report 743, Dept. Comp. Science, Technion, Haifa, Israel, 1992.
- [106] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, 1992.
- [107] Michael John Sebastian Smith. *Application-specific integrated circuits*. Addison-Wesley Longman Publishing Co., Inc., 1997.
- [108] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.
- [109] K. Stevens. *Private communication*, September 2000.
- [110] Marco Storto and Roberto Saletti. Time-multiplexed dual-rail protocol for low-power delay-insensitive asynchronous communication. In Anne-Marie Trullemans-Anckaert and Jens Sparsø, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 127-136, October 1998.
- [111] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720-738, June 1989.

- [112] Hiroaki Terada, Souichi Miyata, and Makoto Iwata. DDMP's: Self-timed super-pipelined data-driven multimedia processors. *Proceedings of the IEEE*, 87(2):282–296, February 1999.
- [113] G. K. Theodoropoulos, G. K. Tsakogiannis, and J. V. Woods. Occam: an asynchronous hardware description language? In *Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology*, pages 249–256, September 1997.
- [114] Augustus K. Uht. Going beyond worst-case specs with teatime. *IEEE Computer*, 37(3):51–56, 2004.
- [115] Stephen H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, 20(12):1437–1444, December 1971.
- [116] The UPPAAL2k Homepage. <http://www.uppaal.com>.
- [117] Hans van Gageldonk, Kees van Berkel, Ad Peeters, Daniel Baumann, Daniel Gloor, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. In *4th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '98)*, 1998.
- [118] Victor I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.
- [119] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.
- [120] Tom Verhoeff. Characterizations of delay-insensitive communication protocols. Computing Science Notes 89/06, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1989.
- [121] *IEEE Standards Navigation Bar IEEE Std 1364-1995 IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*.
- [122] *IEEE standard multivalued logic system for VHDL model interoperability (std.logic_1164)*. 1993. IEEE Std 1164-1993.
- [123] P. Vingron. Coherent design of asynchronous circuits. *IEE Proceedings, Computers and Digital Techniques*, 130(6):190–202, 1983.
- [124] *IEEE standard for VITAL Application-Specific Integrated Circuit (ASIC) modeling specification*. 1996. IEEE Std 1076.4-1995.
- [125] *IEEE standard for VITAL ASIC (application specific integrated circuit) modeling specification*. 2001. IEEE Std 1076.4-2000.
- [126] Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/Gray_coding.
- [127] Ted Williams, Niteen Patkar, and Gene Shen. SPARC64: A 64-b 64-active-instruction out-of-order-execution MCM processor. *IEEE Journal of Solid-State Circuits*, 30(11):1215–1226, November 1995.

- [128] Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.
- [129] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, and S. Temple. Amulet1: A micropipelined arm. In *IEEE Computer Conference*, pages 476–485, 1994.
- [130] Sheng-Fu Wu and P. David Fisher. Automating the design of asynchronous sequential logic circuits. *IEEE Journal of Solid-State Circuits*, 26(3):364–370, March 1991.
- [131] F. Xia, A. Yakovlev, D. Shang, A. Bystrov, A. Koelmans, and D. J. Kinniment. Asynchronous communication mechanisms using self-timed circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 150–159. IEEE Computer Society Press, April 2000.
- [132] Michael Yoeli. Examples of LOTOS-based verification of asynchronous circuits. Technical Report CS-2001-08, Dept. Comp. Science, Technion, Haifa, Israel, 2001.

List of Publications

- [1] Wolfgang Huber. Simulation einer SCSI-Festplatte unter Linux. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/2/182-2, 1040 Vienna, Austria, October 2000.
- [2] Martin Delvai, Wolfgang Huber, Babak Rahbaran, and Andreas Steininger. SPEAR - Design-Entscheidungen für den "Scalable Processor for Embedded Applications in Real-Time Environments". In *Proc. Austrochip 2001, Vienna, Austria*, 2001.
- [3] Martin Delvai, Ulrike Eisenmann, and Wolfgang Huber. Modular construction system for embedded real-time applications. In *Austrochip Proceedings*, pages 103–109, Vienna, October 2002.
- [4] Wolfgang Huber. Peripherieanbindung an SPEAR Extension Modules. Research Report 71/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
- [5] Wolfgang Huber, Martin Delvai, Peter Puschner, and Andreas Steininger. Processor support for temporal predictability – the SPEAR design example. In *Proc. 15th Euromicro International Conference on Real-Time Systems*, July 2003.
- [6] Martin Delvai, Andreas Steininger, and Wolfgang Huber. Solving the fundamental problem of digital design – a systematic review of design methods. Research Report 88/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.
- [7] Karl Hendling, Thomas Losert, Wolfgang Huber, and Martin Jandl. Interference Minimizing Bandwidth Guaranteed On-Line Routing Algorithm for Traffic Engineering. In *Proceedings of the 12th IEEE International Conference on Networks (ICON 2004)*, volume 2, pages 497–503, Singapore, Singapore, November 16–19, 2004. ISBN 0-7803-8783-X.
- [8] Wolfgang Huber, Andreas Steininger, and Martin Delvai. Delay insensitive asynchronous pipeline implementation for code alternation logic. Research Report 85/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

- [9] Martin Jankela, Wolfgang Puffitsch, and Wolfgang Huber. Towards a rapid prototyping framework. *Workshop on Intelligent Solutions in Embedded Systems - WISES 2004*, June 2004.
- [10] Thomas Losert, Wolfgang Huber, Karl Hendling, and Martin Jandl. An Extensible Transport Framework for CORBA with Emphasis on Real-Time Capabilities. In *Proceedings of the IEEE International Conference on Computational Cybernetics (ICCC 2004)*, pages 155–161, Vienna, Austria, August 30, – September 1, 2004. ISBN 3-902463-01-5.
- [11] Wolfgang Puffitsch and Wolfgang Huber. Porting the GNU Compiler Collection to the SPEAR Microprocessor. Research Report 24/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.
- [12] Andreas Steininger, Martin Delvai, and Wolfgang Huber. Code Alternation Logic (CAL): A novel efficient design approach for delay-insensitive asynchronous circuits. Research Report 87/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.
- [13] Karl Hendling, Thomas Losert, Wolfgang Huber, and Martin Jandl. An Intelligent Interference Minimizing Routing Algorithm for Bandwidth Guaranteed Flows. In Wilfried Elmenreich, Tenreiro J. Machado, and Imre J. Rudas, editors, *Intelligent Systems at the Service of Mankind*, volume 2. UBooks, Augsburg, Germany, accepted.
- [14] Thomas Losert, Wolfgang Huber, Karl Hendling, and Martin Jandl. Utilizing CORBA for Hard Real-Time Systems. In Wilfried Elmenreich, Tenreiro J. Machado, and Imre J. Rudas, editors, *Intelligent Systems at the Service of Mankind*, volume 2. UBooks, Augsburg, Germany, accepted.

Curriculum Vitae

Wolfgang Huber

Edla 9

3261 Steinakirchen/Forst

Personal Data

Date of Birth: September 15th, 1974
Place of Birth: Wr. Neustadt
Citizenship: Austria

Education

1980 – 1985 Volksschule (*elementary school*)
Randegg
1985 – 1988 Mittelschule (*secondary school*)
Randegg
1988 – 1993 HTBLuVA St. Pölten
Fachrichtung Elektronik/Informatik
(*polytechnic - Electrical Engineering Department*)
St. Pölten
1993 – 2000 Technische Universität Wien – Informatik
(*Vienna University of Technology – Informatics*)
Academic degree: Diplomingenieur
(*comparable to Master of Science*)
2001 – 2005 Technische Universität Wien – Informatikmanagement
(*Vienna University of Technology –
Computer Science Management*)
Academic degree: Mag.rer.soc.oec
(*comparable to Master of Social and Economic Sciences*)
since February 2001 PhD Studies at the Vienna University of Technology

Working Experience

7/1993 – 2/1994 Military Service
St. Pölten (Austria)
1994 – 2001 Educator at Koplingheim St. Pölten
1995 – 1998 Software Development
KCC -Krammer Computer Consulting
Scheibbs (Austria)
2001 – 2005 Research & Teaching assistant at TU Vienna
Embedded Computing Systems Group