

MASTERARBEIT

**Self-Oscillation as a  
Time-Reference in  
Asynchronous Logic - the  
UART Example**

Ausgeführt am Institut für  
Technische Informatik, Embedded Computing Systems Group  
der Technische Universität Wien

unter der Anleitung von  
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger  
und  
Univ.Ass. Dipl.-Ing. Dr.techn. Martin Delvai

durch  
Werner Klein  
Westgasse 111  
2201 Gerasdorf b. Wien

Wien, 5. Februar 2008

---

## Kurzfassung

Seit vielen Jahren befinden sich synchrone Chip-Designs im Einsatz und haben sich bewährt. Durch Unterstützung von ausgereiften Softwarepaketen und maßgeschneiderter Zieltechnologie ist es möglich, den beträchtlichen Aufwand im Synthese- und Place&Route-Prozess für den Entwickler transparent zu gestalten. Aufgrund steigender Anforderungen an Geschwindigkeit und Energieverbrauch stößt das synchrone Chip-Design zunehmend an seine Grenzen. Signallaufzeiten und Wärmeentwicklung hindern den weiteren Lauf der Taktraterhöhung. Speziell die Taktverteilung ist ein maßgeblicher Problemfaktor. Asynchrone Chip-Designs bieten eine elegante Alternative und werden zunehmend als Forschungsthema aufgegriffen. Eine synchrone Taktverteilung entfällt und die Schaltgeschwindigkeit adaptiert sich an die schaltungstechnischen Gegebenheiten, gesteuert durch ein Handshake-Protokoll. Diese Arbeit behandelt den asynchronen Daten- und Kontrollfluss anhand einer konkreten Implementierung eines speziellen UARTs in “Four State Logic” (FSL). Der UART nutzt eine Rückkoppelung zur Erzeugung einer (mit Jitter behafteten) Oszillation. Diese dient als Zeitbasis. Der UART wird in Hinblick auf die Einhaltung einer genauen Übertragungsgeschwindigkeit geprüft.

Ein wesentlicher Bestandteil dieser Arbeit widmet sich der Simulation. Speziell zur Entwicklung von FSL-Schaltkreisen wurde ein Simulationsmodell erstellt, mit dessen Hilfe auf hoher Abstraktionsebene der Daten- und Kontrollfluss analysiert, sowie etwaige Deadlocks erkannt werden können. Weiters wird diskrete Event-Simulation behandelt und die Funktion anhand eines HDL<sup>1</sup>-Simulators erläutert. Die Simulation des oben genannten UARTS auf verschiedenen Abstraktionsebenen rundet den Teil der Simulation ab.

---

<sup>1</sup>Hardware Description Language

## Abstract

Synchronous chip designs have been in use for plenty of years and performed well. Fully developed software solutions and custom hardware targets spare the programmer most questions concerning synthesis and Place&Route. The underlying algorithms try to find an optimal solution. Increasing demands on speed and power consumption lead the synchronous chip design to its physical limitations. Signal propagation delay and heat development hinder further clock advancement. Clock distribution is a major problem. Asynchronous chip design solves most of the problems in a natural manner and offers an elegant design alternative with increasing research interest. No clock net is necessary and speed adapts to the circuit's conditions, controlled by a handshake protocol. This thesis covers the asynchronous control-, and data flow by means of the concrete implementation of a special UART in four state logic (FSL). The UART exploits a feedback loop to generate an oscillation (jitter-afflicted). The oscillation is used to derive a timebase. The performance of the UART is examined in terms of keeping a specific baud rate.

An essential part of this thesis is dedicated to simulation. A simulation model has been created to support FSL circuit development. By means of this model, control and data flow as well as possible deadlocks can be analysed at a high abstraction level. Moreover, discrete event simulation is covered and its function explained on the basis of an HDL<sup>2</sup> simulator. Simulation at different abstraction levels of the above-mentioned UART completes the simulation part.

---

<sup>2</sup>Hardware Description Language

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>State of the art</b>	<b>4</b>
2.1	Asynchronous logic . . . . .	6
2.2	Four state logic . . . . .	9
2.2.1	Coding scheme . . . . .	9
2.2.2	Combinational logic vs. registers . . . . .	11
2.2.3	Control/Data flow . . . . .	14
2.3	EUART architecture . . . . .	18
<b>3</b>	<b>Modelling of FSL circuits</b>	<b>23</b>
3.1	Motivation for FSL event simulation . . . . .	25
3.2	Selecting an appropriate simulation system . . . . .	26
3.3	Discrete event simulation . . . . .	26
3.4	Basic gates . . . . .	27
3.4.1	Register . . . . .	28
3.4.2	Combinational logic . . . . .	29
3.4.3	Phase inverter . . . . .	30
3.4.4	Simulation aids . . . . .	31
3.5	Running the simulation . . . . .	32
3.6	Example simulation model . . . . .	33
3.7	EUART simulation model . . . . .	33

---

3.8	Pitfalls . . . . .	38
<b>4</b>	<b>HDL circuit simulation</b>	<b>40</b>
4.1	HDL circuit simulation systems . . . . .	40
4.1.1	Simulator kernel . . . . .	41
4.1.2	Signal propagation . . . . .	42
4.1.3	Delta delay . . . . .	43
4.2	Simulation levels . . . . .	47
<b>5</b>	<b>Hardware implementation</b>	<b>49</b>
5.1	Notion of time . . . . .	49
5.2	Impact of FSL on the environment . . . . .	50
5.3	Impact on VHDL description . . . . .	51
5.3.1	Juxtaposition of synchronous and asynchronous design entry . . . . .	51
5.3.2	Coding guidelines . . . . .	55
5.3.3	Pitfalls . . . . .	63
<b>6</b>	<b>Design Flow</b>	<b>67</b>
6.1	Simulation . . . . .	68
6.2	Synthesis . . . . .	68
<b>7</b>	<b>Experimental results</b>	<b>70</b>
7.1	Simulation results . . . . .	70
7.2	Physical results . . . . .	73
7.2.1	Test set-up . . . . .	74
7.2.2	Measurement of the jitter . . . . .	75
7.2.3	Measurement of the long-term oscillation drift . . . . .	77
7.3	Simulation vs. physical results . . . . .	78
<b>8</b>	<b>Conclusion and outlook</b>	<b>79</b>

# List of Figures

2.1	Data moving from source to sink . . . . .	5
2.2	Bundled data approach using matching delays . . . . .	8
2.3	Huffman circuit . . . . .	8
2.4	Phase transition diagram . . . . .	10
2.5	Flow of data waves in FSL . . . . .	10
2.6	Register implementation . . . . .	13
2.7	Control and data flow of the pipeline . . . . .	14
2.8	Full pipeline . . . . .	15
2.9	Empty pipeline . . . . .	16
2.10	Pipeline with feedback paths . . . . .	16
2.11	Block diagram of the EUART . . . . .	19
3.1	Different sequences of data processing . . . . .	24
3.2	Simulation overview . . . . .	25
3.3	Structure of the “example model” . . . . .	34
3.4	Example model in AnyLOGIC . . . . .	34
3.5	Structure of the asynchronous EUART . . . . .	36
3.6	Asynchronous EUART modelled in AnyLOGIC . . . . .	37
3.7	Register in the “busdriver unit” storing three samples of the serial bus line . . . . .	38
3.8	Connecting an inverter . . . . .	39
4.1	Simulator kernel . . . . .	41

---

4.2	Simulation with and without delta delay . . . . .	44
4.3	Synthesised circuit of Listing 4.1 . . . . .	46
4.4	Delta cycles of circuit in Figure 4.3 . . . . .	46
5.1	Concept of design . . . . .	52
5.2	Dependencies between combinational processes . . . . .	63
5.3	Stacking FSL registers . . . . .	65
5.4	Direct feedback path . . . . .	66
7.1	Behavioural simulation of the “aEUART” - switching sequence	71
7.2	Behavioural simulation of the “aEUART” - sync. and receive .	72
7.3	Post-layout simulation of the “aEUART” - self-oscillation . . .	73
7.4	Test set-up . . . . .	74
7.5	Scope-view of the “aEUART” - self-oscillation . . . . .	76
7.6	Effect of jitter. . . . .	77
7.7	Effect of oscillation drift. . . . .	78

# List of Tables

2.1	FSL coding scheme . . . . .	9
2.2	Truth table of a 2-input OR in FSL . . . . .	12
3.1	Register connections of Figure 3.3 . . . . .	35

# Listings

4.1	VHDL example of signal assignments . . . . .	45
5.1	VHDL example of the “stable()” procedure (Paragraph “Simulation aids” in Section 5.3.1) . . . . .	54
5.2	VHDL example of handshake synchronisation (Paragraph “Handshake signals” in Section 5.3.1) . . . . .	55
5.3	VHDL example of combining multiple handshake outputs (Paragraph “Handshake signals” in Section 5.3.1) . . . . .	56
5.4	VHDL example of $\phi$ -detection . . . . .	58
5.5	VHDL example of the conversion of “std_logic” to “cal_logic” .	60
5.6	VHDL example of read back signals . . . . .	61
5.7	VHDL example of initialising FSL registers . . . . .	62

# Chapter 1

## Introduction

Synchronous logic design is a well-known and widespread design technique. High level description languages such as VHDL<sup>1</sup> or Verilog spare the programmer the physical implementation and its difficulties by introducing a clock signal. The clock signal provides a separation of the functionality of the circuit and its temporal behaviour. The designer can trust the processing of a logical operation to finish before its output is consumed upon the next active clock edge. The gap between design entry and implementation is solved by a well-engineered toolchain, resulting in a maximum design frequency.

Physical limitations seem to stop the competition for the highest clock rate. Clock rates about a couple of GHz imply a period below one nanosecond. With respect to speed of light, clock distribution on a silicon die is a great challenge and an upper boundary for the clock rate. The central assumption of synchronous designs is that all components get the active clock edge at the same time. In order to increase the clock rate, circuit designers put a lot of effort in workarounds to keep the illusion of a concurrent clock edge at different locations.

On the other hand, asynchronous designs solve many of contemporary problems in a natural manner. Asynchronous design methods date back to the

---

<sup>1</sup>Very High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

1950s, but compared to the market of synchronous designs nowadays, only a small segment is reserved for asynchronous logic. The mentioned problem of clock distribution is eliminated by using a handshake protocol instead, which is the main difference to synchronous design methods. Furthermore, getting rid of the clock eliminates the vulnerability to this single point of failure. In case of an erroneous oscillator, the whole synchronous design will stop working. Another advantage of asynchronous design is its robustness to varying delays, as caused by temperature drift. The asynchronous circuit works as fast as possible and speed accommodates to new conditions<sup>2</sup>. Switching activity is uncorrelated instead of being synchronised to an active clock edge. This is a welcome feature to prevent side-channel attacks (SCA) and reduce peak power consumption [16]. The latter one also decreases electromagnetic radiation. Later on we will introduce FSL<sup>3</sup>, which, apart from its main purpose, is a countermeasure for differential power analysis attacks, too. Thus, asynchronous logic seems to be well-suited for cryptographic applications.

Along with all these features we have to face a major drawback, coming up in asynchronous logic design: The notion of time. In synchronous logic designs we make use of the clock signal to derive a time base. Especially for time-triggered applications, it is important to act in a specific time slot. This reference clock is missing in asynchronous designs. Instead, the circuit has to measure time periods in relation to its self-oscillation speed to derive a time base. In contrast to the accurate clock signal in synchronous logic designs, this self-oscillation speed can vary because of varying circuit delays.

To address this topic, we will implement a UART<sup>4</sup> using delay-insensitive asynchronous logic. The challenge is to communicate at an exactly defined baud rate. In order to get a notion of time, the communication partner has to send a synchronisation pattern first. The asynchronous UART has to deal with oscillation jitter, which necessitates a more complex design. As

---

<sup>2</sup>Implying an appropriate asynchronous design technique.

<sup>3</sup>**F**our **S**tate **L**ogic

<sup>4</sup>**U**niversal **A**synchronous **R**eceiver **T**ransmitter

synchronous reference design we used a special enhanced UART (Section 2.3), optimised for usage with drifting clock sources.

The structure of the thesis is as follows: After this short introduction, Chapter 2 gives an overview of asynchronous logic and the fundamental design problem. Four state logic (FSL) is introduced, and different structures, such as pipelines, are discussed. Furthermore, the EUART<sup>5</sup> is being presented. FSL event simulation as well as discrete event simulation are covered in Chapter 3. The asynchronous EUART model is being discussed. Chapter 4 explains the HDL simulation working principle and different simulation levels. Chapter 5 and 6 are covering the hardware implementation, the design flow and the impact of asynchronous logic on the design and environment. Chapter 7 provides the experimental results. Finally, Chapter 8 presents the conclusion and outlook.

---

<sup>5</sup>(E)nhanced UART

# Chapter 2

## State of the art

Circuit design can basically be divided into the synchronous and the asynchronous approach. Whereas the synchronous approach deals with a periodic trigger (clock), the asynchronous counterpart is controlled by a handshake protocol between each data source and data sink. Both approaches aim at the same target: to ensure that all data is correctly processed by the circuit. This problem is called “fundamental design problem” and is discussed in detail in [3]. In order to highlight the differences between design styles, the following keywords are explained: validity, consistency and losslessness.

**Validity:** A signal is termed valid, if it is the stable result of a Boolean function performed on a consistent data word.

**Consistency:** A data word may consist of a set of valid values. A consistent input vector consists of valid signals that belong to the same context. For example, if all input signals but one already carry the new data whereas the last signal line is still carrying the old value (because of a delay), the vector is inconsistent.

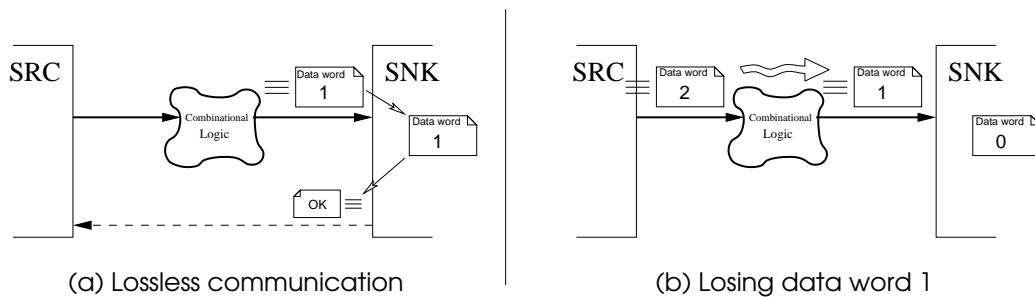


Figure 2.1: Data moving from source to sink

**Losslessness:** Lossless data flow means that no data wave must be lost on its way inside the circuit. In other words, a data word travelling from the source to the sink must not be lost. To guarantee such a behaviour, the sink has to trigger the source when it is ready for new data. This principle is referred to as “handshake” and shown in Figure 2.1(a) with an explicit handshake line from the sink to the source. The Figure shows a short pipeline structure, which will be discussed in more detail in Section 2.2.3. If there was a second data wave travelling to the sink, with the first one still on its way, there would be a risk of the second one to overwhelm and eliminate the first data wave, as depicted in Figure 2.1(b).

The synchronous logic design solves the problems of validity, consistency and losslessness in the time domain. The clock period considers all path and logic delays, thus implementation knowledge is necessary. After a specific period of time, all signals have to be valid and consistent to be consumed. There is a trade-off between performance and robustness. Hard timing enables higher data throughput but increases the risk of failing hardware.

In Section 2.1 an overview of asynchronous concepts to solve the fundamental design problem is given. Later on, we will introduce “Four State Logic”, short FSL. This concept is used for the implementation of the asynchronous EUART and will be the focus of this thesis.

## 2.1 Asynchronous logic

Whatever kind of design technique is used, the fundamental design problem has to be solved. This Section presents various asynchronous approaches and their way of tackling validity, consistency and losslessness. We distinguish two ways in which the handshake can be generated: explicitly or implicitly.

**Explicit handshake generation:** This type of asynchronous circuits uses a separate handshake signal to inform the source about the state of the sink. The sink can signalise a “busy” state, in which the source has to wait before issuing new data, or a “ready” state, in which the sink is ready to receive new data from the source. The explicit handshake generation ensures losslessness, which is one requirement for an error-free circuit.

Information about validity and consistency can be provided by an explicit delayed signal, for example. The delay (called matching delay) has to take the worst case of the combinational logic delay into account. The inputs of the sink are assumed to be valid and consistent when the signal through the matching delay has reached the sink. This approach is known as the “bundled data” approach and pictured in Figure 2.2. The bundled data approach moves the disadvantages of using time to establish validity and consistency (like synchronous designs do) to local regions. The circuit faces the same drawbacks as synchronous designs do, except that no large clock network exists. The matching delay has to be built out of hardware. Circuits of this kind are called “self-timed” (ST) circuits<sup>1</sup>.

Another possibility of ensuring validity and visualising consistency is to use signal encoding. This requires a dual rail encoding as given in the NCL<sup>2</sup> and FSL approach. NCL was developed by Theseus Logic, Inc. and extends Boolean logic by a third so-called “null” state, separating consecutive data

---

<sup>1</sup>Self-timed circuits contain regions of self-timed elements. Assumptions on delays are individually met for each region.

<sup>2</sup>Null Convention Logic

waves. The FSL approach is similar to the NCL example, except that the “null” state is eliminated by the use of four states in order to represent the logical values HIGH and LOW. Both NCL and FSL use the information domain to ensure validity. The fact that only one rail has to change from one valid code word to the next<sup>3</sup> ensures validity. A consistent data word consists of a set of equally encoded inputs in the case of FSL. In NCL, a data word is considered to be consistent, if all inputs receive a data wave after a preceding null wave.

“Transition signalling” constitutes another dual rail approach. In contrast to other approaches, the rails do not carry a state. Instead, a transition on rail “a” signals a new HIGH bit and a transition on rail “b” a new LOW bit. If no transitions occur, the state is comparable to the “null” state of NCL. The so-called “one-hot” encoding fulfils validity on the code level. A data word is rated to be consistent, if all inputs receive exactly one transition (event) since the last consistent data word. Circuits, such as necessary to build FSL, NCL and “transition signalling” designs are called “delay-insensitive” (DI) circuits<sup>4</sup> or, more precisely, “quasi-delay-insensitive” (QDI)<sup>5</sup>. Further details concerning FSL are presented in Section 2.2.

**Implicit handshake generation:** This type of asynchronous circuits does not provide an explicit handshake line. Instead, assumptions on the environment are made. The “fundamental mode” postulates that the circuit has to be in a stable state before one input signal is allowed to change [21]. As internal signals of the circuit are not visible to the environment, the longest delay of the circuit has to be calculated. The environment has to wait at least for this period, until it is allowed to change a single input line again. As representative, we mention the “Huffman approach”, pictured in Figure 2.3.

---

<sup>3</sup>Assuming an appropriate coding scheme

<sup>4</sup>Delay-insensitive circuits operate correctly regardless of the delays on its gates and wires.

<sup>5</sup>Quasi-delay-insensitive circuits are delay-insensitive, except that “isochronic forks” may be required.

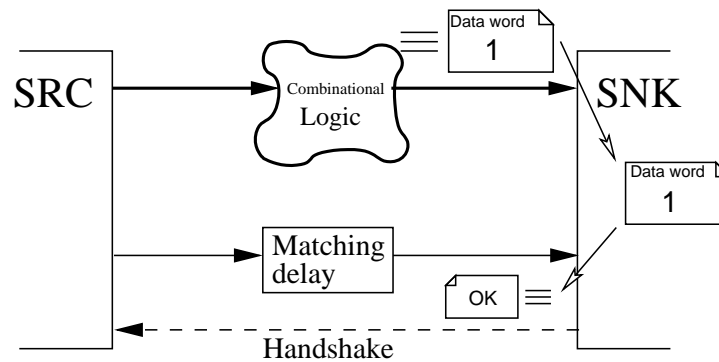


Figure 2.2: Bundled data approach using matching delays

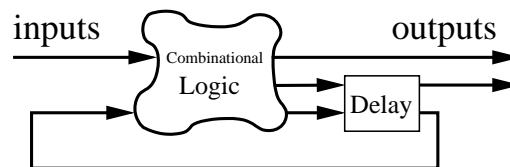


Figure 2.3: Huffman circuit

There are primary inputs, primary outputs and a feedback loop, which counts as input as well.

Due to the feedback, it is possible to save a state and implement an asynchronous state machine. Only one input signal is allowed to change at a time if the circuit is in a stable state. Afterwards, the following state is calculated and fed back through the delayed loop. Now the logic calculates new output according to the new state and settles in a stable state. During this period, the environment is not allowed to change any inputs. As only one signal is allowed to change at a time, the state encoding has to be chosen with care. Validity is solved in the information domain by using glitch-free functions and strong restrictions on the environment. The delay in the feedback loop ensures consistency by preventing new calculated output to disturb the input vector immediately. Huffman circuits are classified as “self-timed” circuits

because of the delayed feedback path.

## 2.2 Four state logic

FSL solves the fundamental design problem, mentioned at the beginning of this chapter, completely in the information domain. This means the circuit is delay-insensitive at gate level (not at transistor level). Neither path delays nor component delays impair the circuit's functionality. In contrast to NCL, FSL transfers information every cycle. This is accomplished with a coding scheme based on two code sets, as shown in the course of this chapter. We are going to use the term CAL (Code Alternation Logic) at some points in this thesis, which is just another abbreviation for FSL.

### 2.2.1 Coding scheme

FSL uses two code sets for coding each of the values HIGH and LOW in two different ways. The implementation consists of two signal rails for each data line to represent four states. Table 2.1 shows the coding scheme and the values of the two rails, further called "a" and "b". The logical information (HIGH or LOW) corresponds to rail "a". The code set (or so-called "phase") can easily be identified when both rails are combined by an XOR. For example,  $(a,b) = (1,0)$  carries the logical information HIGH and is coded in phase  $\phi 1$  because  $(1XOR0) = 1$ . To distinguish whether a signal is coded in  $\phi 0$  or  $\phi 1$ , we use upper and lower case letters for HIGH and LOW as used in Figure 2.4.

logic state	code $\phi 0$	code $\phi 1$
LOW	$(a,b)=(0,0)$	$(a,b)=(0,1)$
HIGH	$(a,b)=(1,1)$	$(a,b)=(1,0)$

Table 2.1: FSL coding scheme

The two code sets are used alternately for data transmission. Especially

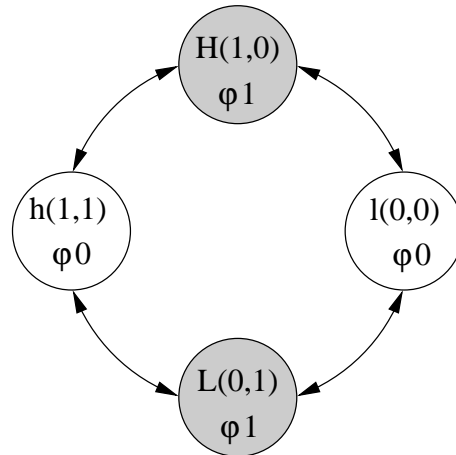


Figure 2.4: Phase transition diagram

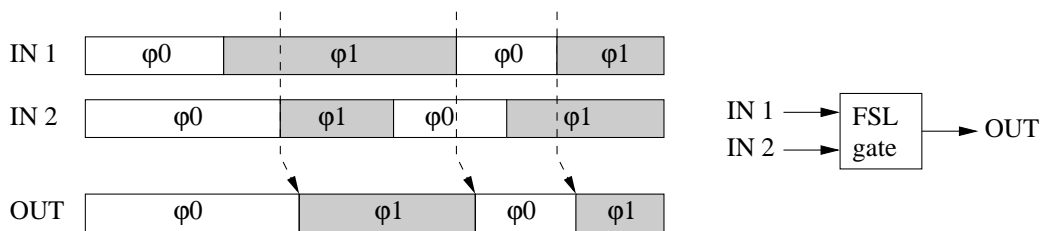


Figure 2.5: Flow of data waves in FSL

when switching code set, only one rail has to change its value. This very important property ensures continuous validity. A new context can be understood as issuing new data coded in the alternating phase (other code set). If all inputs appear in the same phase, they are in the same context and thus rated to be consistent. The possible phase transitions can be seen in Figure 2.4.

Figure 2.5 shows the flow of data waves in FSL. The data words can be easily synchronised due to the alternation of  $\phi_0$  and  $\phi_1$  waves.

### 2.2.2 Combinational logic vs. registers

In general, gates inside a logical circuit can be classified as sequential (registers, memories) and combinational gates (AND, OR, INV).

**Combinational gates:** Combinational gates do not participate in the handshake procedure. Thus we call them “transparent”. The gates compute the output value in the phase the consistent inputs are in. If they are not in the same phase, the output has to be held in the last state. In [13], the following important rules for FSL gates are summarized:

1. Data values of each signal must be coded in alternating phases.
2. The calculation is performed when all inputs are in the same phase.
3. In case the input signals are in different phases, the output remains in its last valid state.

The rules applied: A 2-input OR gate in FSL (each input consists of two rails in hardware) sees the input vector “HL”. Note the case sensitive letters representing the phase. As both inputs are in the same phase and therefore consistent, the output evaluates to 'H' (same phase as the input). If one of the inputs is fed with new data, the input vector becomes inconsistent (“Hl”, for example) and the OR gate remains in the last valid state 'H'. Once the second input is updated with new data, too (“hl”, for example), the input vector is consistent. Both inputs are in the same phase and the output evaluates to 'h'. As we can see, consecutive equal logical values are recognised by their different coding. Note that during all transitions only one rail of the FSL signals changes. Table 2.2 shows the truth table of a 2-input OR in FSL. The hold state symbolises a frozen output because of inconsistent inputs.

In comparison to standard logic gates, the gate count (logic elements) needed for FSL gates is significantly higher when mapping to the current

<b>Z</b>		<b>Input1</b>			
		h	l	H	L
<b>Input 2</b>	h	h	h	hold	hold
	l	h	l	hold	hold
	H	hold	hold	H	H
	L	hold	hold	H	L

Table 2.2: Truth table of a 2-input OR in FSL

FPGA generation. A 2-input gate in FSL in fact requires 4 inputs (two rails for each input) and a memory cell to hold the last state. Moreover, the current FPGA generation offers an unsuitable structure for asynchronous designs. Implementation details of an AND gate can be viewed in [3].

**Registers:** Registers participate in the handshake procedure. They are used for pipeline structures, as introduced in Section 2.2.3. Figure 2.6(a) presents the implementation of an FSL register. It contains a latch, the control logic (ctrl) to switch the latch transparent, a phase detector at the input as well as at the output data port, and an input (pass) as well as an output (c-done) port for the handshake signals. The latch inside the FSL register gets transparent, in other words, transports the input data to the output only if the following rules (switching conditions) apply:

1. The input phase differs from the output phase.
2. The phase of the downstream stage (register) equals the phase stored in this register's latch cells.

The input and output phase is examined by phase detectors. If a **consistent** data word is detected at the input data port, the phase information is extracted. Otherwise, the information of the last phase is kept. If the input phase differs from the output phase, obviously a new consistent data word, coded in the new phase, is waiting to be captured. Before the latch is set transparent, we have to ensure that the data word currently stored in

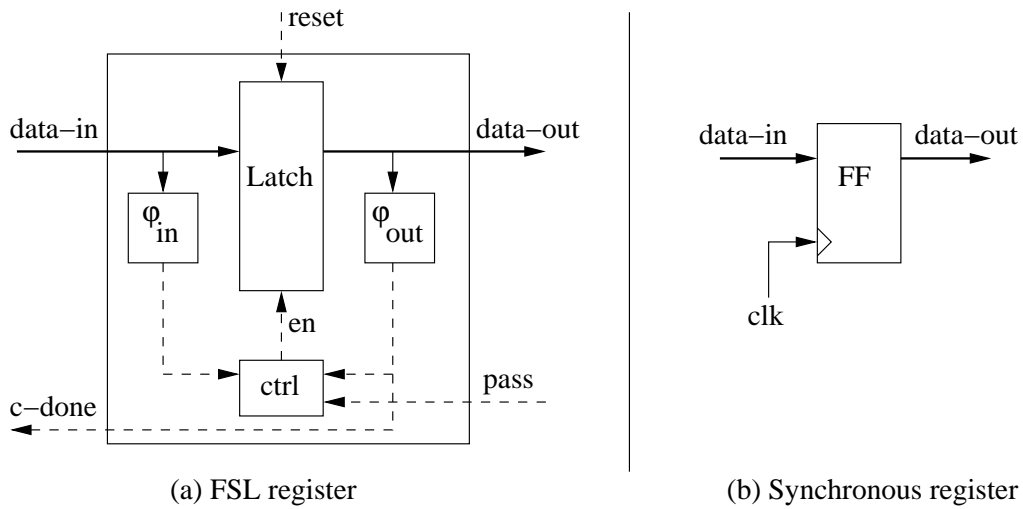


Figure 2.6: Register implementation

the latch (which is going to be overwritten) has passed the combinational logic  $g(x)$  and been consumed by the downstream register already. This information is provided by the feedback (handshake) signal. Confer Figure 2.7 representing the situation. The incoming feedback signal is called “pass”, the outgoing one “capture done” (c-done). The generation of the c-done signal is simply the output of the phase detector at the output data port. If the phase of the downstream stage equals the phase stored in this register’s latch, the latch is set transparent to capture the new data. Otherwise the latch remains closed. The above-mentioned rules are explained in Section 2.2.3 more practically. Information concerning delay assumptions at transistor level inside the FSL register can be gained from [13]. The synchronous counterpart of the FSL register is shown in Figure 2.6(b), which is triggered upon the rising/falling edge of the clock signal.

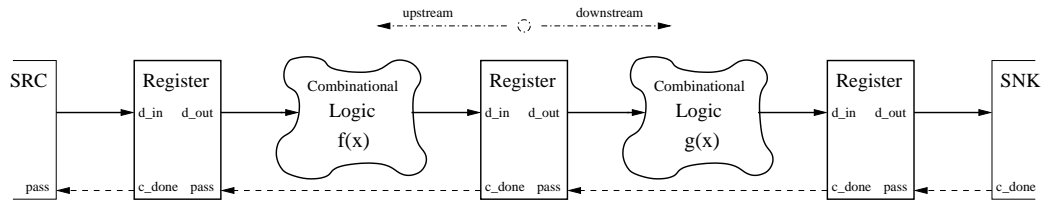


Figure 2.7: Control and data flow of the pipeline

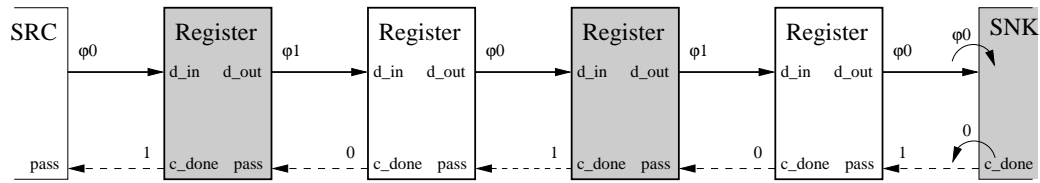
### 2.2.3 Control/Data flow

Depending on the structure of the circuit and the initial setup of the memory elements inside of the registers (“initialisation”), different sequences of events are possible. First of all, we are going to introduce the pipeline structure to address this topic. Forward- and feedback paths are explained afterwards, followed by phase inverter placement and register initialisation.

**Pipeline:** Registers are used for pipeline structures, as shown in Figure 2.7. They are involved in the handshake process to provide a lossless data flow in the circuit. The FSL register gets transparent if the rules (switching conditions) presented in Section 2.2.2 apply. We are going to explain these rules by means of Figure 2.7: The middle register stores a data word coded in one of the two phases  $\phi_0$  or  $\phi_1$ . This word can be seen at the output of the register. As soon as the downstream register has consumed the data word in question, the middle register is allowed to capture a new data word. This communication is done with an explicit handshake line, signalling the phase of the stored data word in the downstream register to the middle register. If there is a new data word issued by the left register now, the middle one will capture immediately.

As pipeline throughput and control flow differ depending on operating conditions such as full/empty pipelines or push/pull communication, we are going to differentiate these conditions now.

Let’s recall the coding scheme described in Section 2.2.1. When switching



Combinational logic between registers is not drawn to keep the figure clear.

Figure 2.8: Full pipeline

context, the new data has to be coded in the alternating phase. This mechanism enables to distinguish between consecutive data waves. Assuming that every pipe stage (register) carries the opposite phase of its neighbour's registers, as pointed out in Figure 2.8, this is called a **full** pipeline because every stage carries a new data wave, waiting to be captured by its downstream register. As soon as a data wave is consumed by the sink, the pipeline stages capture step by step, beginning on the right hand side. This is due to the fact that the register located in front of the sink is the first one allowed to switch after the sink has consumed its data. Therefore, the sink is connected with a handshake line, too. The principle is called a **pull** pipeline because the sink is in charge of the control flow (given the source is faster than the sink). If the pipeline is full, the source has to wait until a new data wave is accepted. In the **empty** pipeline, shown in Figure 2.9, all stages carry the same phase value, thus the same context. When the source issues a new data wave, the pipeline stages capture step by step, beginning on the left hand side. The data wave is transferred immediately to the sink through the pipe stages, since all registers have been waiting for new data. In this case, the source is in charge of the control flow (given the sink is faster than the source) and the principle is called a **push** pipeline.

**Forward / Feedback paths:** Many designs bring up a feedback path. State machines, for example, store their state in a register which is fed back to the logic again to control the data flow, as depicted in Figure 2.10. What

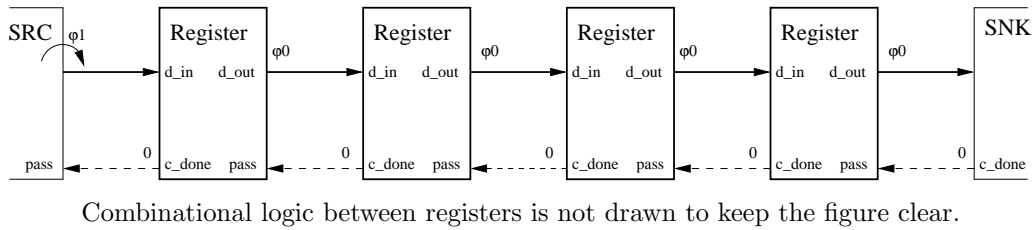


Figure 2.9: Empty pipeline

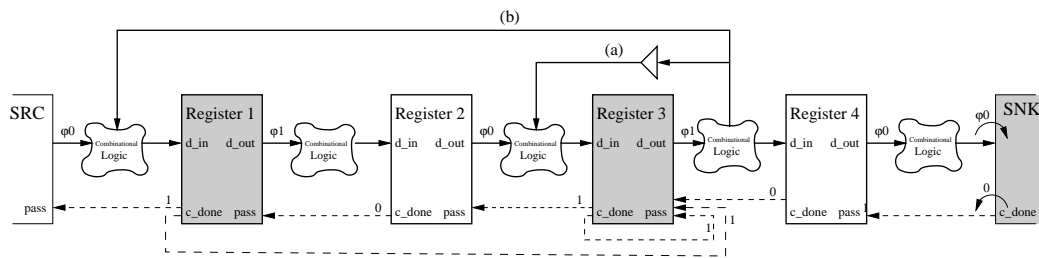


Figure 2.10: Pipeline with feedback paths

happens to the data of feedback path (a)? At the beginning of this chapter we pointed out consistency. FSL gates will evaluate new output if all input lines carry the same phase. As sketched in Figure 2.10, we have to place a phase inverter in feedback path (a) to establish consistent inputs at the register. Feedback path (b) does not need any phase inverter. At first sight, this causes an inconsistent input at register 1. Remember that the register chain starts to switch, beginning at the sink (on the right hand side) in case of a full pipeline. At the moment when register 3 captures the  $\phi_0$  wave from register 2, the input at register 1 becomes consistent. The placement of phase inverters is an error-prone task. Depending on the initialisation of the registers, the setup may look different. Therefore a simulation system has been developed within the scope of this thesis (see Chapter 3). An algorithm for inverter placement is discussed in the next Paragraph.

**Phase inverter placement:** Phase inverters are used to convert a data wave to the opposite coding set (opposite phase). This may be necessary in non-linear data paths when combining several data lines. Some general facts can be stated concerning inverter placement for empty initialised pipelines as explained in [3].

- In an empty initialised pipeline, a phase inverter has to be placed on all feedback paths.
- In an empty initialised pipeline, no phase inverter has to be placed on forward paths.

Whereas these two rules apply to empty pipelines, they do not apply to full one because in the latter the control flow is running in the opposite direction of the data flow. A generic algorithm for placing phase inverters is presented:

1. The steady state is considered: Starting from an arbitrary non-transparent node (register): Mark all outgoing edges with the same phase and follow them to the next non-transparent nodes. Mark these outputs with the alternative phase and continue following these edges. If an edge leads to a node that has been already visited, this edge may have to be phase-inverted to match the phase of the other input edges.
2. The dynamic behaviour is considered: A phase inverter is placed on all valid<sup>6</sup> feedback paths. The final circuit has to be initialised as assumed when applying the algorithm. Note that, together with a phase inverter on the data path, the handshake line has to be inverted, too<sup>7</sup>.

**Initialisation:** The FSL register contains an initial value, which defines the first output before any switching occurs. A set of connected registers needs well-considered initial values to avoid deadlocks. With an increasing amount

---

<sup>6</sup>A valid feedback path must contain at least two register nodes.

<sup>7</sup>Except for invalid feedback paths.

of registers and their interconnections, finding a working solution gets more complex. The initialisation values are closely linked with the placement of phase inverters.

## 2.3 EUART architecture

Within the scope of this thesis, an asynchronous E(nhanced) U(niversal) A(synchronous) R(eceiver) T(ransmitter) has been developed. Please note that the first occurrence of the term “asynchronous” denotes the asynchronous design approach, which is the integral part of this thesis. The second occurrence of “asynchronous” depicts just a property of the communication type. The description of the synchronous EUART, used as reference design for the asynchronous implementation, is subject to this Section. The initial version of the synchronous design was written by [7]. Later on, the UART has been redesigned to fit for real-time applications. The major improvements are periodic synchronisation, required by LIN<sup>8</sup> and TTP/A<sup>9</sup> protocols [15], and the facility to work with unprecise clock sources with a high drift rate, as it is the case with low cost RC oscillators. An oversampling mechanism has been added to reduce transmission errors. For more details see [10]. An overview of the enhanced UART is given in the following.

The EUART is realised as a generic extension module [5] and communicates with a processor over a very slim memory-mapped interface. The eight registers can be accessed via store and load instructions. These registers contain configuration data, status information and payload data. The central part of the EUART is the “control unit”. Its task is to coordinate the individual components which together form the functionality. The eight registers are part of the control unit. The control unit reads the values of the registers and generates the control signals for all other components. The “enhanced

---

<sup>8</sup>Local Interconnect Network [17]

<sup>9</sup>Time-Triggered Protocol for SAE Class A Applications [25]

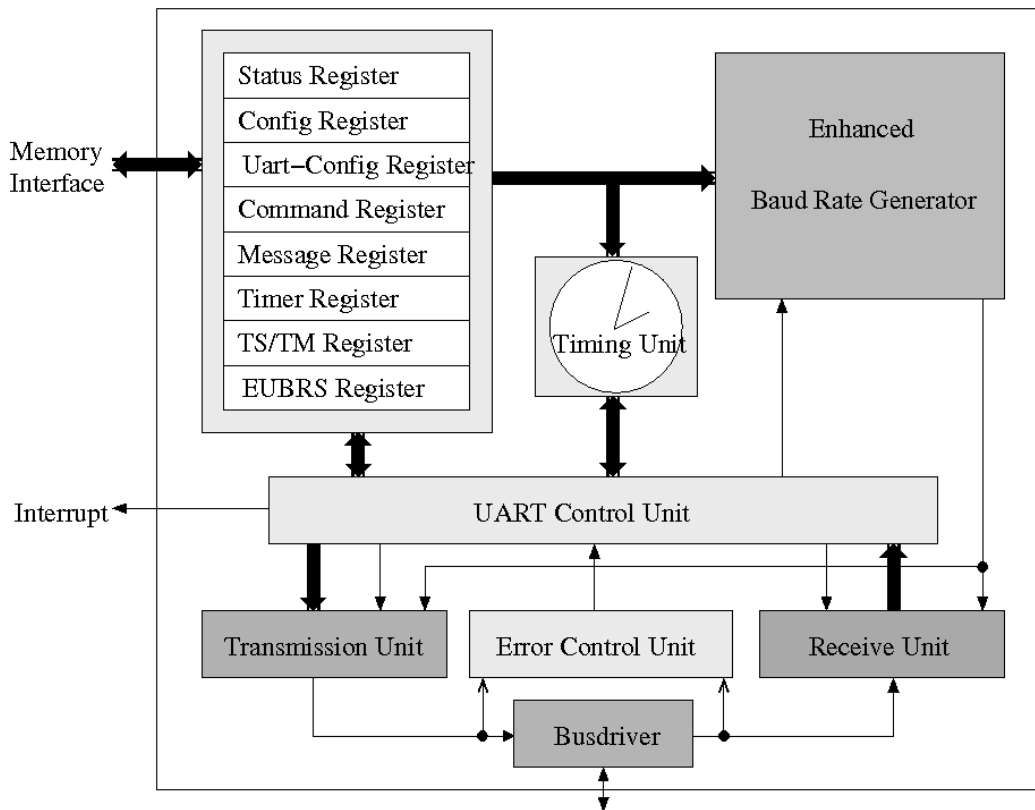


Figure 2.11: Block diagram of the EUART

baud rate generator” generates the ticks for the “receiver” and “transmitter” component. The “error control unit” checks the communication process and signals any error on the serial bus line. The “busdriver” contains a three level filter (3 bus samples constitute the recognised bus level) to obtain higher robustness against spike interferences. The “timing unit” contains timers used for time stamps or scheduled events.

The “message register” contains the received message. A message written to the “message register” is transmitted immediately. The current timer value is exported to the “timer register”. Time stamps and timer match values are maintained in the “TS/TM register”. The “uart-config register”

is used to define the message length, parity- and stop-bit settings. The combination of events and assigned actions in the “command register” enables the EUART to be configured in various ways. For example, “start a transmission” (action) at a specific “point in time” (event). The combination of a timer and a UART leads to a powerful module for real-time applications.

In general, UARTS need a special oscillator frequency to achieve transfer speeds such as 19200 bit/second. The baud rate depends on the frequency and the baud rate setting (UBRS). Following [4], the baud rate is calculated by:

$$BaudRate = \frac{f}{C_1 \cdot (UBRS + C_2)}$$

$C_1$  and  $C_2$  are integers, depending on the implementation. Typically,  $C_2$  is set to 1 and  $C_1$  represents the number of samples per bit cell. The dividend  $f$  depicts the oscillator frequency. In case of a specially calibrated oscillator (4.9152MHz instead of 5.00 MHz), UBRS would be a whole number. But one of the key points of this EUART is to work with cheap oscillators, which is why UBRS is a fractional value. When approximating UBRS by a whole number, an arithmetic error occurs, which leads to an imprecise baud rate. Therefore the “EUBRS register”<sup>10</sup> is partitioned into an integer part (first 12 bits) and a fractional part (last 4 bits). Thus, the length of one bit is computed using the following formula:

$$t_{bit} = \frac{EUBRS}{16} \cdot \frac{1}{f_{clk}} \cdot \frac{1}{2}$$

The “EUBRS register” is interpreted by the “enhanced baud rate generator”. The upper 12 digits are interpreted as the integer part. The lower four bits (fractional part) are used for the extrapolation mechanism. This mechanism considers the fractional part by suppressing one tick of the sample rate

---

<sup>10</sup>This is the baud rate setting register of the enhanced UART

counter, according to the strategy in [7].

Cheap oscillators are subject to high drift rates resulting in a baud rate drift. If multiple nodes are participating in the communication this error gets worse because of individual oscillators. To overcome the baud rate drift, this EUART, apart from a synchronisation mechanism, features continuous resynchronisation.

The synchronisation mechanism is an essential part of the EUART design and verified in [10]. A short summary is given in the following: The EUART captures the timing of the bus by receiving synchronisation patterns from a master node (TTP/A, LIN). A valid synchronisation pattern consists of a sequence of zeros and ones. A new node<sup>11</sup> listening to the bus has to detect the synchronisation pattern by analysing the time elapsed between transitions. Therefore a timer is set to zero with the first falling edge and counts the length of the bit cell. This value is compared to the value of the next bit cell and so on. If eight equidistant transitions have been recognised, the pattern is accepted and used to set the baud rate. Otherwise, the procedure has to be restarted. With respect to oscillation drift, the first and the last bit cell of a perfect synchronisation pattern may be measured with one tick difference, causing the synchronisation to fail permanently. Therefore, a “small” deviation must be tolerated. This tolerance value will be specified shortly. To avoid malicious synchronisation spanning multiple messages, the parity check has to be set accordingly, and the tolerance has to be set below an upper boundary: For symmetric time slots (messages of an even number of bits), the case in which one half of the bits are ones and the second half are zeros has to be prevented. This can be achieved by selecting the parity check appropriate:

$$\frac{Msg}{2} \begin{cases} even & \rightarrow \text{Even parity} \\ odd & \rightarrow \text{Odd parity} \end{cases}$$

---

<sup>11</sup>In this context, a “node” is understood as a communication partner in the time-triggered network, interfacing the bus through the EUART module.

For asymmetric time slots (messages of an odd number of bits), sequences of  $k$  zeros (ones) and  $k+1$  ones (zeros) must not be considered as synchronisation patterns. This can be achieved by setting an upper tolerance limit of 16,6%, assuming time slots to consist of 13 bit cells, as in the case of TTP/A. The lower tolerance limit is set to 4% to tolerate at least one tick deviations between two bit cells. The upper limit avoids malicious synchronisation to data traffic. A tolerance of 6.25% has been chosen for the implementation, since this value is easy to implement in hardware. The corresponding formulas are explained in [10].

Due to automatic synchronisation and resynchronisation the EUART is well-suited for porting to an asynchronous implementation. Instead of an imprecise RC oscillator, we have an imprecise oscillation origin from the asynchronous design. The eliminated arithmetic error mentioned above allows even more tolerance in varying oscillation. The asynchronous EUART design is mentioned in the next chapters along with the addressed topics.

# Chapter 3

## Modelling of FSL circuits

The conventional design approach of digital systems provides two functional steps: First the system architecture has to be defined in consideration of high level issues such as performance, throughput, functional units and framework. Furthermore, basic components are identified and specified, accompanied by specifying their interaction. During this design phase, decisions concerning bus structures and component behaviour are made. The outcome is a component-based description of the overall system, with each component being defined by its interface and behaviour.

In a second step, these components are implemented by using an HDL<sup>1</sup>. In synchronous designs, a component is driven by a clock signal which controls the internal processing sequence. Due to well-explored synchronous design techniques, the implementation of the component specification happens in a straight-forward manner in most cases. In contrast to that, asynchronous designs are event-based (which makes a clock obsolete), coming up with an other challenge that has to be considered during implementation: The processing sequence of data is important, since different sequences will lead to different results. For example, the result of the adder in Figure 3.1 (a) moves forward through the register chain by one step per cycle, whereas in

---

<sup>1</sup>Hardware Description Language

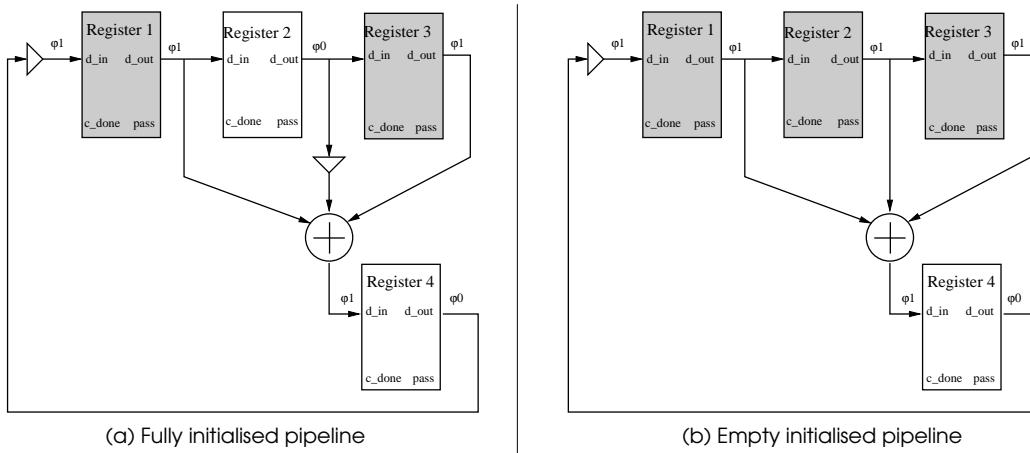


Figure 3.1: Different sequences of data processing

(b), the result is shifted through the registers 1,2 and 3 at once.

Moreover, deadlocks may occur if unrelated data is linked together or data may be lost if any handshake line is missing. Even worse, a circuit that seems to work, can fail in certain conditions because of an incorrect design. Therefore, register locations, phase inverter placement and handshake lines have to be evaluated to support the implementation process. Within this thesis, a simulation environment for FSL circuits has been created to bridge the gap between the specification at system level and the asynchronous implementation. Section 3.6 includes a small circuit, which the reader is invited to examine to get an idea of the complexity of asynchronous circuits. Figure 3.2 shows the different simulation levels during system design, with increasing grain from top to bottom. FSL event simulation is situated between high-level system simulation and component-based HDL simulation.

	Simulation level	Modelling language	Simulation system
Increasing grain ↓	System-level	---	System Studio
	FSL event-level	JAVA	AnyLogic
	Behavioural-level	{ HDL	Synplify, Quartus, Design Compiler
	Prelayout gate-level		
	Postlayout gate-level		
	Switch-level		
Transistor-level			

Figure 3.2: Simulation overview

### 3.1 Motivation for FSL event simulation

With the growing utilisation of registers and cross-linking signals, the system gets more complex. The initialisation of registers and placement of phase inverters is a error-prone task. Moreover, source-sink<sup>2</sup> pairs have to be identified and connected by a handshake line. This simulation step will not replace HDL circuit simulation. It is intended as an additional simulation to support the asynchronous design process. The level of abstraction hides the functional part (combinational logic) of the design and focuses on the interconnection of registers. The benefits of FSL event simulation are a better overview of data-flow and the handshake process, as well as a verification of the phase inverter placement. For example, the aEUART design consists of six register blocks in total 388 bits wide. They are cross-linked to each other, resulting in a high grade of interconnection and complexity. Identifying source-sink pairs and investigation of a working setup is the objective of FSL event simulation.

<sup>2</sup>A pair of linked registers is called “source-sink” pair.

## 3.2 Selecting an appropriate simulation system

There are a lot of different simulation systems available. Starting with the model requirements, we are able to choose a suitable software product: The FSL system can be modelled by a sequence of events, changing discrete state variables. The data flow / handshake information can be represented by data packets, travelling over the network within an arbitrary delay. Packets arriving at a component cause events and may change their inner state. According to some processing rules, further output is generated. All state variables are discrete and finite. Therefore we are going to choose a discrete event simulation system; in Section 3.3 we are going to decide in favour of a specific software system.

## 3.3 Discrete event simulation

Discrete event simulation may be written in any language such as JAVA or C++. There are some common mechanisms used for discrete event simulation: A variety of free and commercial libraries offer support for event scheduling, random number generation, statistical results, graphical representation and others. This is the most flexible and most time-consuming way to create a model. For quicker model building, a set of GUI<sup>3</sup>-based simulation systems are available on the market. Due to drag&drop facilities and predefined library elements, the model creation process is simplified and reduced to parameter settings . Whatever design method is used, discrete event simulation includes an event list to keep track of the next taken action. Events are scheduled to occur at a specific point in time and may be added to the event list dynamically with progression of simulation. This happens if a specific event triggers a new event: A “coin inserted” event will change

---

<sup>3</sup>Graphical User Interface

the internal state of a music box to “music on” and schedule an event “stop music” to be executed after an amount of time. The event list is processed sequentially and simulation time may jump to the next event if real-time mode is not desired. Initial events are scheduled during the start-up of the simulation. The simulation runs as long as the event list is not empty or any other stop condition is fulfilled. The function of such a simulator is discussed by means of a discrete event simulator for HDL models in Chapter 4. For FSL event simulation, our aim is to enhance a GUI-based simulation system with an FSL library to enable the user to construct arbitrary models in a quick and easy manner. The product AnyLOGIC [26] has been selected, since it offers a GUI, drag&drop and animation facilities. The FSL library is written in JAVA, which is the language supported by AnyLOGIC. In Section 3.7 we present the aEUART simulation model, built by means of the FSL library.

### 3.4 Basic gates


Basically, we need to implement a library which provides the basic gates of FSL. They have already been discussed in Section 2.2.2 and are the following:

- Combinational gates (AND, OR, INV)
- Register
- Phase inverter

Since we are not interested in the semantics of data transferred in the circuit, the combinational gates are modelled in one combinational object to represent any logical function of the circuit. The focus of interest is the phase information that is transported. This abstraction helps to keep FSL event simulation as simple as possible. In addition to the basic gates, we are going to implement a source as well as a sink object and some handy simulation aids. Before discussing each basic gate separately, some common information concerning the gates and the modelling in AnyLOGIC [26] shall be provided:

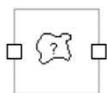
AnyLOGIC uses communication lines to exchange messages between so-called ActiveObjects. The FSL simulation library contains ActiveObjects like the register, the phase inverter and any other element in the library, just called “object” from now on. There is exactly one input port on the left hand side and one output port on the right hand side of every object. In fact, the ports are bidirectional, as will be explained later. All messages pass through these ports. A state-chart is used to process incoming messages and to update the animation which reflects the current state of the object. Outgoing messages are travelling along the communication lines, connected to the output port. A message is just another object used to exchange information between objects. Regardless of the message type (object), all messages use the same communication line between two objects. For the modelling of delay insensitivity, random delays are used. In the following, the basic gates are presented.

### 3.4.1 Register

 The function of the register is discussed in Section 2.2.2. The simulation model offers an automatic handshake routing. This ensures easy and clean routing. So-called handshake messages are sent along the incoming data path (communication line) to the upstream registers. Therefore every connected upstream (source) register is automatically linked with the handshake. An arbitrary amount of registers can be connected to the input and output port. When talking of an input/output port, this is always related to the data path. Of course, the handshake messages are travelling in the opposite direction, and the ports are bidirectional. The output port of the register delivers a data message containing the incoming phase when the switching condition is fulfilled (see Section 2.2.2). At the same time, a handshake message is released to inform the upstream registers about the capture event.

The register administrates a list of the connected upstream and downstream registers. This list is automatically initialised at the start-up of the simulation run by a registration sequence. We have to know all connected neighbour registers to judge the consistency of the data inputs and to verify whether all handshake messages arrived at the output port (losslessness)<sup>4</sup>. If all the handshakes are equal, the handshake value is processed. This synchronisation has to be done explicitly for the implementation in hardware (see Listing 5.2). The list items are printed to a logging file providing the designer with an identification of the source-sink pairs, which is valuable information for subsequent HDL implementation. When instantiating the register, an initial value ('h','l','H','L') can be set. In fact, there is no difference in selecting 'h' or 'H' / 'l' or 'L', since only the phase information is important. The simulation results are equal regardless of whether HIGH or LOW values are used. The register is called “cal\_register” in the simulation library.

### 3.4.2 Combinational logic

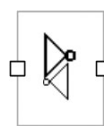


This object is filed under “cal\_transparent” in the simulation library and represents the combinational logic in the design, as explained in Section 2.2.2. The cal\_transparent object calculates new output in the phase of the inputs, if **all** of them are in the same phase (consistent). We have to know the upstream (source) registers to check this rule, like in the case of the register (see Section 3.4.1). Incoming handshake messages are just forwarded to support the handshake management of the register. The parameter “initial\_value” has no effect on this object. There is a modified version called “cal\_transparent\_zero\_delay” in the simulation library, which does not model the combinational delay.

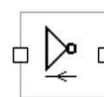
<sup>4</sup>In hardware, this issue does not arise, since a fixed count of data inputs and handshake lines is examined.

### 3.4.3 Phase inverter

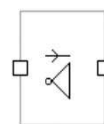
As soon as forward/feedback paths are used, it may become necessary to convert the phase. Moreover, a phase inversion is generally linked with a handshake inversion. But sometimes it is necessary to invert the phase information and keep the handshake untouched or vice versa. As the simulation model uses a single communication line for data and handshake messages, all combinations of inversion are implemented:



**Phase inverter with handshake inversion:** This object inverts the phase of data messages and the value of handshake messages routed through it. In many situations, this is the desired function. The corresponding object in the simulation library is called “cal\_phase\_inverter\_handshake\_inverter”.



**Phase inverter with handshake forward:** If there is a feedback path from the register output to the register input of the same register, it is necessary to convert the phase, but not the handshake signal. In this case, the object in question will do the job. Figure 3.8 pictures the situation. A data message, coming from the register’s output, is being phase-inverted and fed back to the combinational logic (transparent node). The handshake message is travelling in the opposite direction without being modified. The corresponding object in the simulation library is called “cal\_phase\_inverter\_handshake\_forward”.

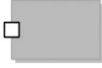


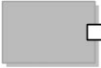
**Phase forwarding with handshake inversion:** This object completes the possibilities of inversion combinations by inverting the handshake

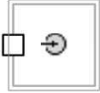
only. Data messages are passing without being modified. The corresponding object in the simulation library is called “cal\_phase\_forward\_handshake\_inverter”.

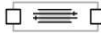
### 3.4.4 Simulation aids

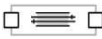
The following objects complete the basic gates to provide a useful library.


**Sink:**  If we take a look at the pipeline in Figure 2.8, we see that new data waves are issued at the beginning of the pipeline (source) and are consumed at its end (sink). The object in question represents the sink. It has one input port on the left hand side to receive data messages. An incoming data message is always accepted and acknowledged by the corresponding handshake message. Messages are deferred with a random delay. All upstream (source) registers are automatically stored in a list for the same reason as for the register (consistency issue). The library name of this object is “cal\_sink”.

**Source:**  The source issues the first data wave defined by the parameter “initial\_value”. New data waves are generated adequately to the incoming handshake message. An incoming handshake message always leads to a new data wave. Messages are deferred with a random delay. All downstream (sink) registers are automatically stored in a list, for the same reason as for the register (losslessness issue). This object is filed under “cal\_source” in the simulation library.

**monitor:**  This object serves the debugging purpose only. Connected to a port, it is logging all outgoing messages.

**Delay (exponentially distributed):**  This object is used to model time delays by deferring messages. Any message entering this object through its input port is delayed and then forwarded to the output port. The delay is exponentially distributed. The parameter can be set when instantiating this object. It is used in the `cal.transparent` object to model the combinational delay, for example. The library name is “`serial_channel_distr`”.

**Delay (constant):**  Like the “Delay” object mentioned above, this object defers messages, but with a constant delay. The delay can be set when instantiating this object. Its name in the library is “`serial_channel_fixed_delay`”. The delay objects share the same symbol.

**Register with separate handshake:**  This object equals the register object, except for the fact that an additional handshake port is added. This port enables the separate routing of the handshake line.

## 3.5 Running the simulation

The simulation process starts with a registration sequence. During this sequence, all registers are sending a “hello” message containing its unique ID through the upstream (input) and downstream (output) port. If this message arrives at a register, the ID is added to the appropriate list. The source and sink registers are automatically identified like that, which is a valuable information for the subsequent implementation. The registration sequence necessarily gets to know the connected neighbour of each register. During the simulation, the registers and transparent nodes can be watched when capturing a new data wave by means of an event counter and colouring. If the circuit activity stops, there may be inconsistent input data or a wrong input phase somewhere. In this case, the bug can be found by analysing the

colouring of the phase waves and event counter values or by examining the log files.

### 3.6 Example simulation model

The “example model” in Figure 3.3 is considered as a starting point in FSL circuit building. Table 3.1 lists the upstream and downstream registers from the point of view of a dedicated register, as they are connected in Figure 3.3. The “sync” component is used to combine multiple handshake lines. If all incoming handshake lines are equal, the corresponding handshake message is forwarded to the register. A circle at the input depicts a negation of the handshake value. This component equals the behaviour of the implementation in Listing 5.2. Figure 3.4 represents the same circuit modelled in AnyLOGIC. The two registers “\_cal\_reg\_a” and “\_cal\_reg\_b” on the right consist of an explicit handshake port (c\_done) due to the fact that no possible constellation can be achieved out of the phase inverter combinations. The “sync” component is part of the register and therefore invisible. The sequence in which the registers fire is left to be discovered by the interested reader. Note that combinational logic keeps the old output value until a consistent input word leads to new output. To verify your results, take a look at the flipped text.

Sequence:  
Reg a and Reg b are allowed to fire first.  
Reg c fires after Reg a has fired.  
Reg b is allowed to switch again after Reg a has fired.

### 3.7 EUART simulation model

Figure 3.5 illustrates the individual components of the asynchronous EUART and their interconnections. Each component except for the “busdriver” contains a register and an internal feedback loop. In fact, the “busdriver” also contains a register, but it is not connected to the aEUART pipeline. Therefore, it does not affect the control flow and is omitted in Figure 3.5.

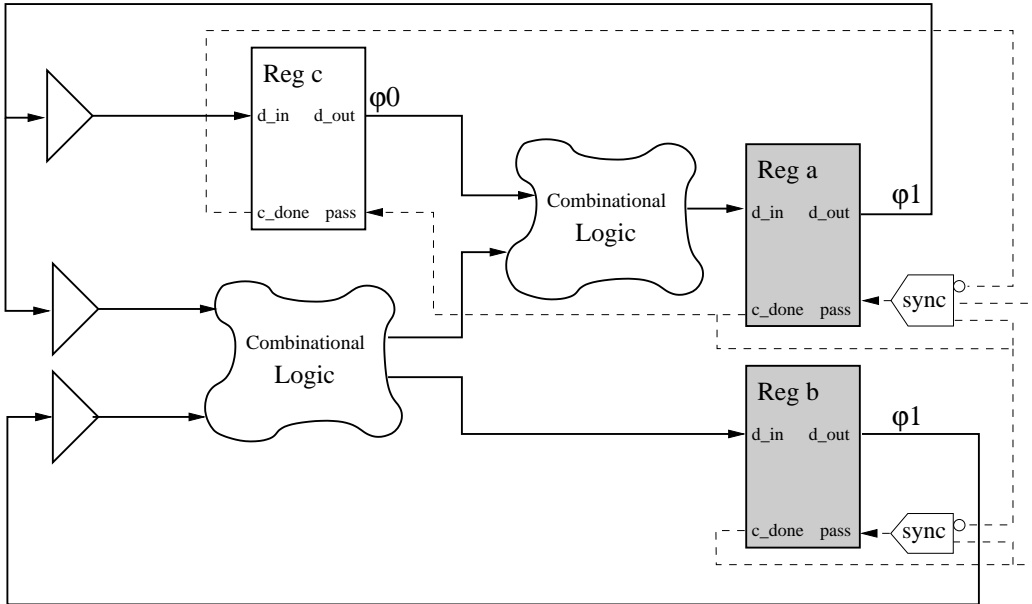


Figure 3.3: Structure of the “example model”

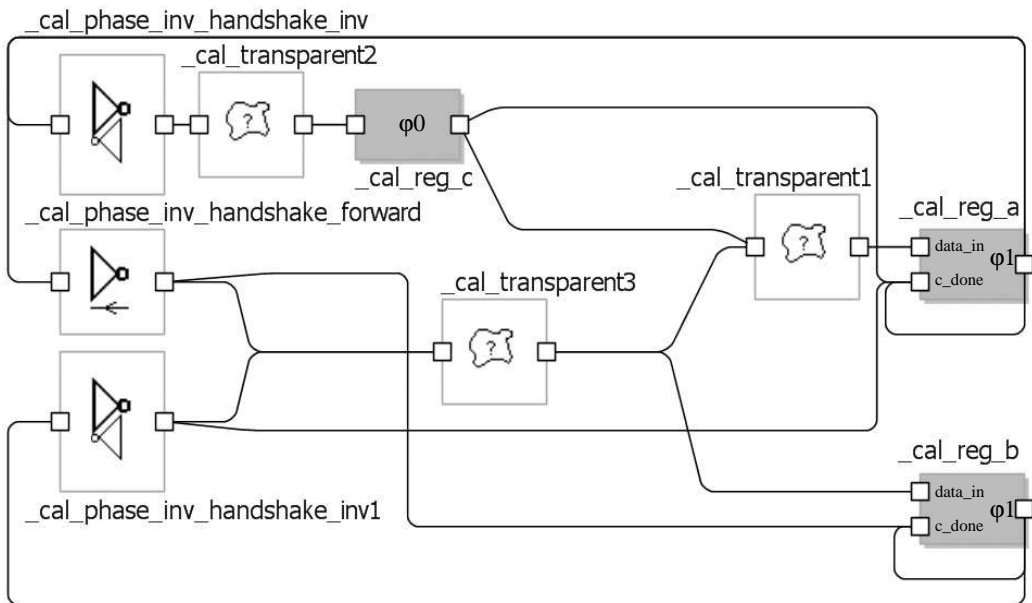


Figure 3.4: Example model in AnyLOGIC

point of view	upstream registers	downstream registers
Reg c	Reg a	Reg a
Reg a	Reg c Reg a Reg b	Reg c Reg a Reg b
Reg b	Reg a Reg b	Reg a Reg b

Table 3.1: Register connections of Figure 3.3

This stand-alone register, illustrated in Figure 3.7, is used to store the last three samples of the serial bus line. The “bus\_in” signal has its origin in a standard logic signal that is permanently converted to FSL, suitable for the register. The handshake line is fed back to the register itself. This leads to permanent switching activity, hence the bus is sampled as fast as possible. On the basis of these samples, short interferences are filtered out.

Figure 3.6 shows the aEUART circuit modelled in AnyLOGIC. Registers and the combinational logic are hidden inside the components. The components are denoted with the initial phase value  $(\phi_0, \phi_1)$  of the contained register. The components consist of input (left) and output (right) ports as defined in their HDL entity description. As we are not interested in multiple links between two components, some of the components’ ports are left open.

Starting with the placed inverters and register initialisation given in Figure 3.6, we are now going to examine the switching sequence, which is a result of the simulation run: The “transmitter” and “uartctr” are the only two components with consistent inputs. But the “uartctr” inputs are in the wrong (old) phase to trigger the register, whereas the “transmitter” inputs are present in the right (new) phase. This causes the register in the “transmitter” component to be the first one to fire. The new “transmitter” output changes the input of the “timing”-, “errctr”- and receiver component to a consistent data word. They fire next. As soon as the “receiver” has fired, its output changes the input of the “ebr\_generator” to a consistent data word.

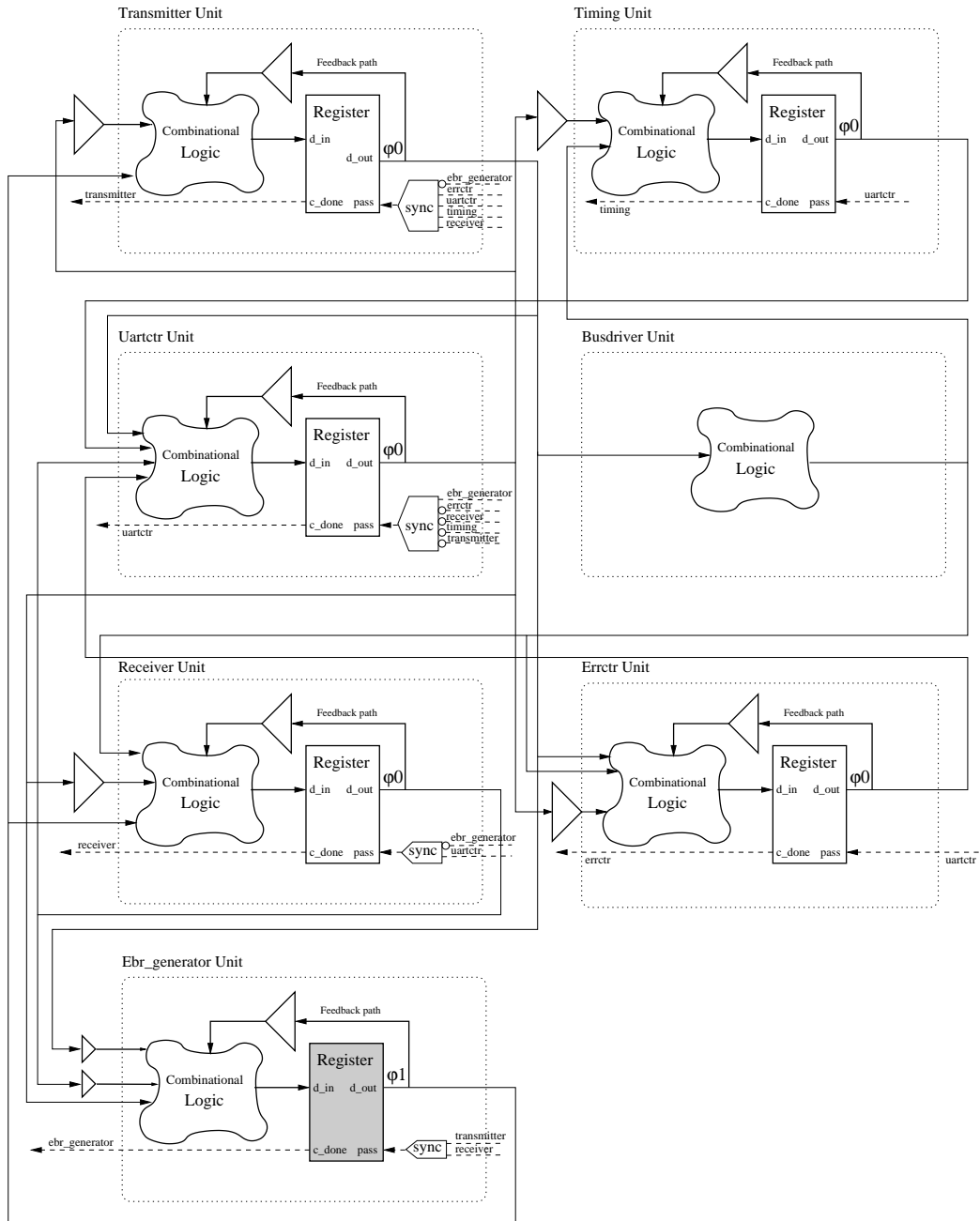


Figure 3.5: Structure of the asynchronous EUART

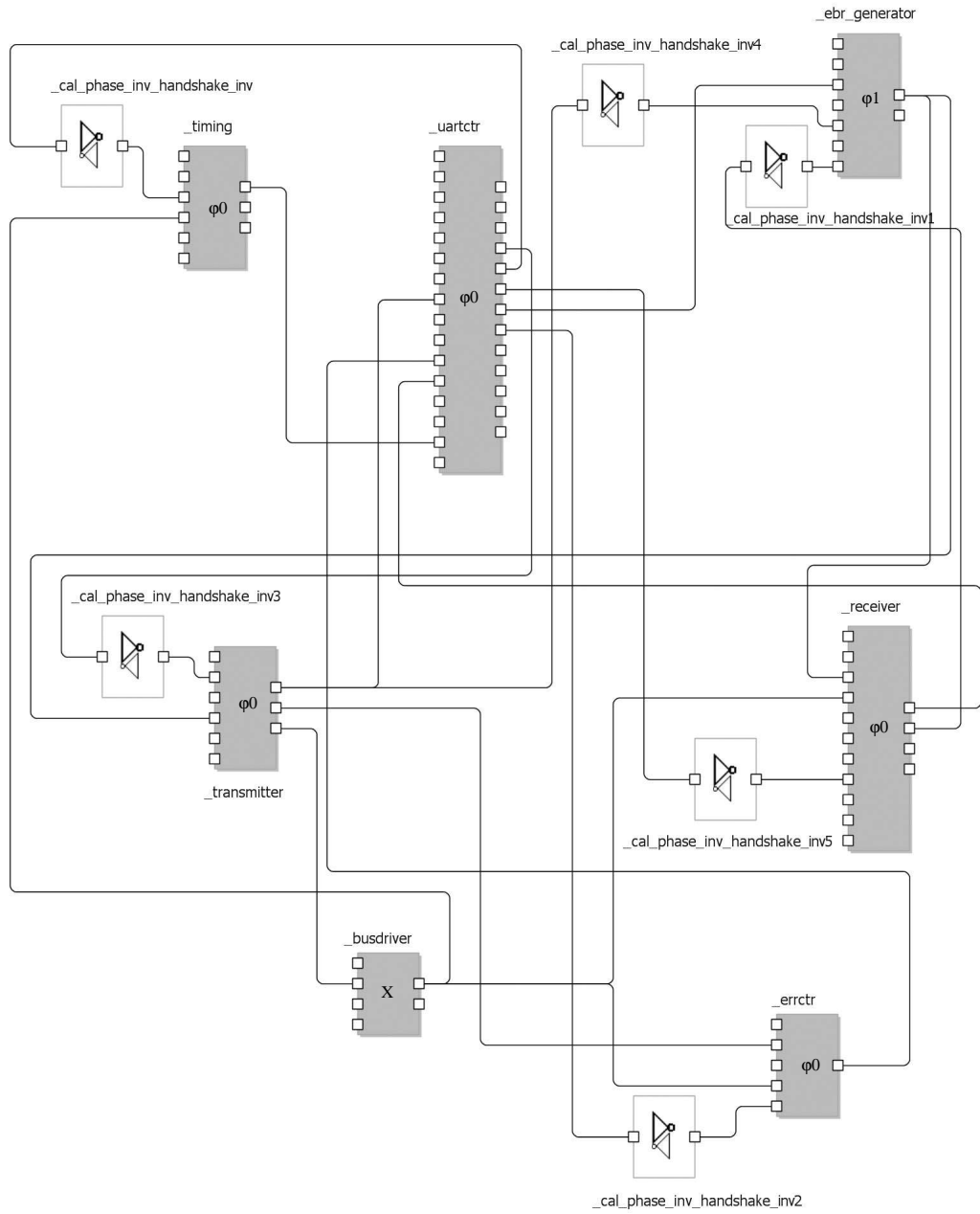


Figure 3.6: Asynchronous EUART modelled in AnyLOGIC

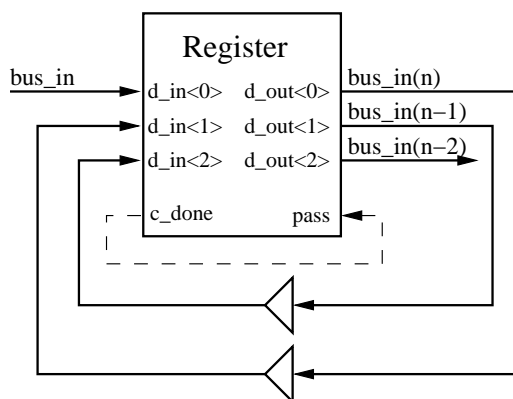


Figure 3.7: Register in the “busdriver unit” storing three samples of the serial bus line

The “ebr\_generator” fires and finally completes the “uartctr” component input with the last missing  $\phi_1$  value to establish consistency. The “uartctr” is the last component in the sequence, before a new cycle starts at the transmitter component. Sequence: {transmitter}  $\rightarrow$  {timing,errctr,receiver}  $\rightarrow$  {ebr\_generator}  $\rightarrow$  {uartctr}.

### 3.8 Pitfalls

AnyLogic [26] uses messages to communicate between objects. When a message arrives at a port, the message can be referenced to by a variable called “msg”. Note that this is just a reference. Be sure to make a copy of the message if you intend to store it for later purpose and if the message, referenced by the “msg” pointer, is modified somewhere else. Especially when forwarding the incoming message to other objects, all of them are working on the same copy (“reference forwarding”).

Take care when placing phase/handshake inverters. From an optical point of view, they should be placed in front of following objects. This ensures an intuitive correct connection (input on the left hand side). Confer Figure 3.8,

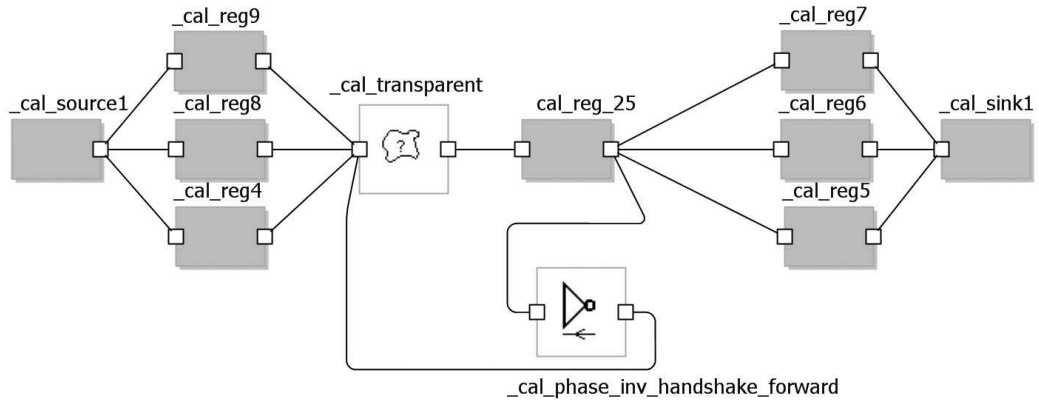


Figure 3.8: Connecting an inverter

which shows the inverter that is not positioned in front of the following object.

# Chapter 4

## HDL circuit simulation

This chapter covers HDL circuit simulation at different levels of grain by means of the asynchronous EUART design. We start at behavioural simulation, continue with the levels depicted in Figure 3.2 and end up with gate-level simulation, which is the last meaningful simulation step towards an FPGA design before downloading for physical verification. Prior to the discussion of the different levels in Section 4.2, the function of an HDL circuit simulator is explained in the following Section.

### 4.1 HDL circuit simulation systems

HDL circuit simulation systems are discrete event-based, as described in Section 3.3. The model is defined in the VHDL (or Verilog) language and analysed/executed by the simulation system. In the following, we are going to describe the simulator kernel and discard the “text editor”, the “schematic editor”, the “waveform viewer”, etc., which constitute a complete simulation system [1].

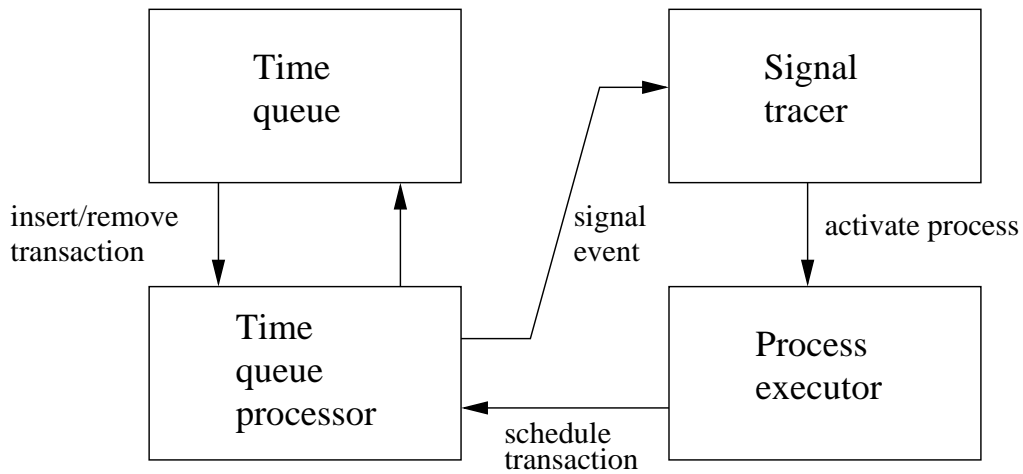


Figure 4.1: Simulator kernel

### 4.1.1 Simulator kernel

The kernel is illustrated in Figure 4.1, which consists of four major parts: Time queue, time queue processor, signal tracer and process executor.

The kernel operates on a network of processes with input and output signals. A process is executed if one of the input signals changes in value. As a consequence new output is calculated. Some signal changes do not lead to new process output immediately, as in the case of sampled signals in a sequential process. Such signals are sampled upon a clock event, for example. Their evaluation at other instants would be a waste of time. Hence it is sufficient to execute a sequential process at clock events only. On the other hand, a combinational process has to be executed every time one of the input signals changes. The sensitivity list is used to determine signals triggering the process in case of a value change. The time queue processor keeps track of so-called transactions. Transactions are stored in the time queue in chronological order. A transaction contains a signal name, the value it is assigned to and the time when this assignment is scheduled. If a transaction represents a signal change (the new value is different from the current one),

it is called a signal event. Signal events are forwarded to the signal tracer, which marks all processes for execution that are concerned by the signal event. The marked processes are executed by the process executor and new resulting time queue entries are forwarded to the time queue processor to be inserted into the time queue. A simplified simulation cycle consists of the following steps, preceded by an initialisation phase where all variables and signals are set to their default values:

1. Increment simulation time to the time of the next entry (transaction) in the time queue. If there are no time queue entries, stop simulation.
2. Read and remove all transactions that are scheduled at the current time instant from the queue. Activate the corresponding processes for transactions representing signal events.
3. Execute all activated processes. If any, schedule new transactions. Continue with step 1.

As soon as we have introduced delay models, an advanced simulation cycle will be presented.

### 4.1.2 Signal propagation

Signal propagation is natural and of physical origin. For accurate modelling of hardware, VHDL supports two types of delay:

- Inertial delay
- Transport delay

Inertial delay models the effect of capacitance on signal changes. Signal propagation takes place if the input persists at a given level for a given period of time. Glitches are filtered out if they are shorter than the given time. Inertial delay is implied unless the key word “transport” is used.

Transport delay propagates all signal changes, even short spikes. The VHDL examples look as follows:

```
Y <= SigA after 3 ns; --inertial delay
```

```
Y <=transport SigA after 3 ns; --transport delay
```

If no time delay is given, zero delay is assumed:

```
Y <= SigA; --same as "SigA after 0 ns"
```

For simulation purposes, all zero delay assignments are replaced by so called “delta delays”, which will be explained in Section 4.1.3. Note that delay models are available for signals only, as they may be considered synthesised directly in hardware and are associated with one or more drivers [20]. Variables do not support delay models, since they provide containers for values used in the computation of signal values. Variable assignments are instantaneous and indicated by the “:=” symbol:

```
Y := VarA; --instantaneous variable assignment.
```

### 4.1.3 Delta delay

HDL allows the designer to describe systems at various levels of abstraction. As such, timing and delay information may not always be included in an HDL description. Delta delay is the default propagation delay, if no delay is explicitly prescribed. Therefore, HDL signal assignments do not take place immediately. Instead, they are scheduled for the next delta cycle, respectively. A delta delay value is a non-zero time interval, selected by the HDL compiler, so that an indefinite number of such intervals may be accommodated within a simulation time step [11]. No sum of delta delays can advance simulation time. The delta delay supports the model of concurrent process execution in VHDL: The execution order of processes during simulation does not affect the simulation output. All active processes can be executed in the same simulation cycle. Without the concept of delta cycles, simulation results may differ depending on the process execution order. Figure 4.2 illus-

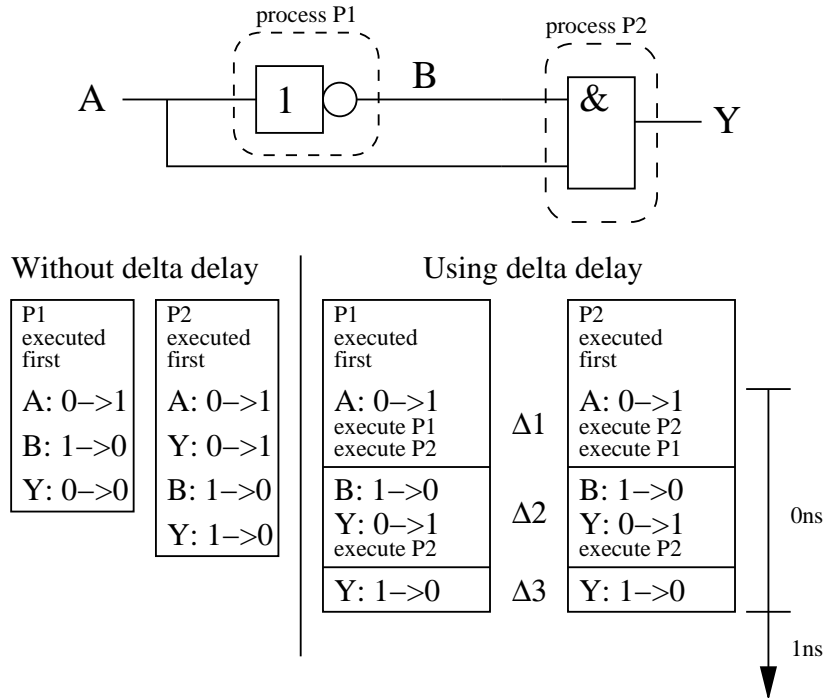


Figure 4.2: Simulation with and without delta delay

trates the simulation results of a simple circuit with and without the usage of delta delays. The simulation, without using delta delays and executing process “P1” first, discards the spike on output “Y”. If the concept of delta delays is used, both simulation runs are equal, regardless of which process is executed first.

We are now going to extend the simplified simulation cycle presented in the course of Section 4.1.1.

1. Increment simulation time to the time of the next entry (transaction) in the time queue. If there are no time queue entries, stop simulation.
2. Read and remove all transactions that are scheduled at the current time instant or delta-delayed entries from the queue. Activate corresponding processes (in any order) for transactions representing signal events.

---

```
LIBRARY IEEE;
USE IEEE.STD_Logic_1164.all;
USE IEEE.Numeric_STD.all;

entity signal_assignment is
  port(A,B,C,D : in std_logic;
       Y       : out std_logic);
end entity signal_assignment;

architecture behaviour of signal_assignment is
  signal Sig1, Sig2: std_logic;
begin
  Sig1 <= A and B and C;

  P: process (Sig1, Sig2, D)
  begin
    Y <= Sig2 or D;
    Sig2 <= not Sig1;
  end process P;
end behaviour;
```

---

Listing 4.1: VHDL example of signal assignments

3. Execute all activated processes. If any, schedule new transactions Some of them may be delta-delayed transactions. If so, continue with step 2; otherwise, go to step 1.

Note that there may be many delta cycles related to the same simulation time. Every time a signal is assigned (with zero- or omitted delay statement), the assigned value is available (updated) in the next delta simulation cycle though within the same simulation time. As process execution order does not matter, processes can also be executed on different processors to speed up the simulation. A complete example for studying delta cycles is given in Listing 4.1, the synthesised circuit in Figure 4.3 and the delta cycle computation in Figure 4.4.

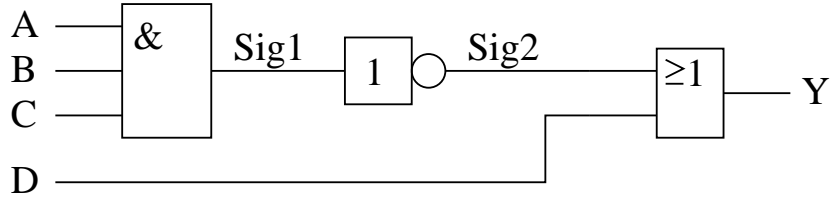


Figure 4.3: Synthesised circuit of Listing 4.1

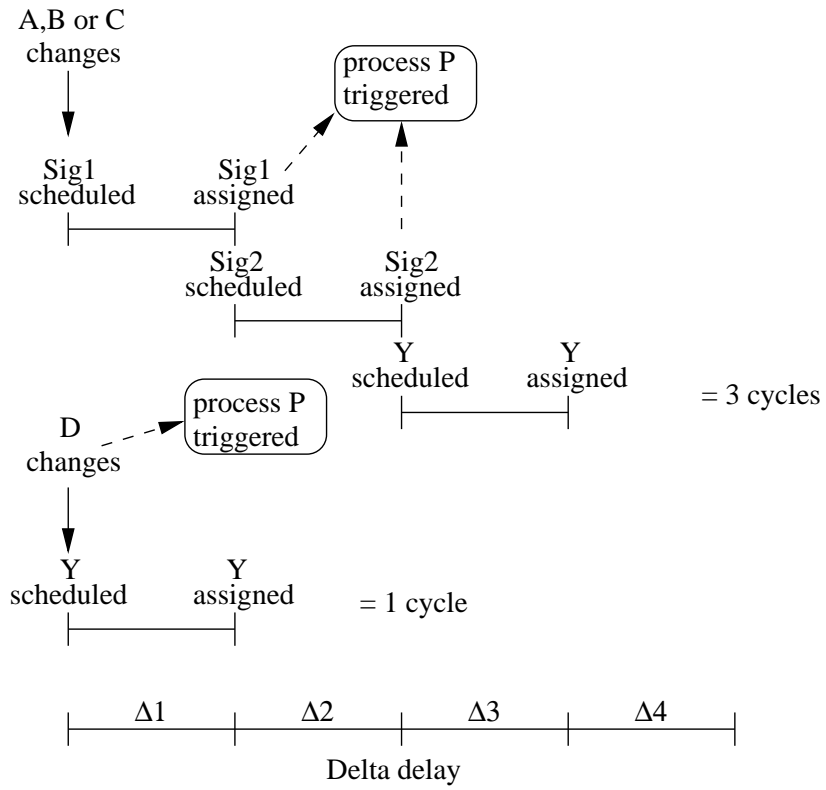


Figure 4.4: Delta cycles of circuit in Figure 4.3

## 4.2 Simulation levels

Simulation is used to verify the specification by stimulating the design and examining the reaction. The stimulus is written in a testbench, which is used through all simulation steps. A positive simulation result does not imply that the design is error-free, but only that the tested functionality worked properly. There may still be undetected errors. Figure 3.2 lists the levels of HDL simulation [22], which we are going to explain now with additional information concerning FSL:

**Behavioural:** Behavioural simulation is used to verify the functionality of the design and testbench. No timing information is considered. Moreover, all HDL constructs can be used, since the circuit is not synthesised yet. The HDL source is elaborated and converted to an internal simulator representation. Behavioural simulation offers the fastest simulation speed because the source code is directly simulated. In the simulation of standard logic designs, a process is suspended until a signal event triggers its execution. In FSL designs, a process has to be triggered if a signal event occurs **and** all inputs are consistent. Since we are using a standard simulation system, we have to implement such a behaviour. This is done by leaving the sensitivity list empty, and by the so called “stable()” function. The “stable()” function waits until all process inputs are consistent. For detailed information see Paragraph “Simulation aids” of Section 5.3.1.

**Prelayout gate-level:** Prelayout gate-level simulation is used to verify the synthesised circuit at gate level. The source code has been interpreted and translated to standard gates by the synthesis tool. All gates have the same “unit delay”, as their precise delays are unknown at this point in time. Therefore, exact timing information cannot be expected for this simulation step. The afore mentioned “stable()” function is not synthesise-able. The consistency check is implemented in the gates, which means that the FSL

design has been translated to special FSL gates, instead of standard gates. FSL gates contain a memory element to store the last output in case of inconsistent input. Since this memory element cannot be initialised, undefined values may propagate through the circuit if the inputs are not consistent or undefined. To prevent such situations, combinational logic should be fed by registers which provide initial values.

**Postlayout gate-level:** Postlayout gate-level simulation is used to verify the timing of the design. The standard gates have been mapped to technology-dependent gates of the FPGA vendor. A static timing analysis calculates the gate delays, depending on parameters such as temperature and voltage. The “unit delays” are replaced with the exact timing values by a back-annotation process. Even the wire delays are known since they have been calculated after the Place&Route process. Note that static timing analysis cannot determine a maximum system frequency for asynchronous designs because of a missing clock signal. In the case of the asynchronous EUART, the Place&Route process has been performed by the “Altera Quartus II” tool. The Altera APEX20KC FPGA has been selected as target device.

**Switch-level:** Switch-level simulation takes into account voltage levels instead of logical values. Gates are replaced by transistors, which are modelled to be simple switches. Switch-level and the following transistor-level simulation are not used for FPGA-targeted designs.

**Transistor-level:** Transistor-level is the lowest simulation level. The transistor is modelled according to its physical behaviour, including parasitic effects. Simulation speed at this level is much slower than on the level of behavioural simulation.

# Chapter 5

## Hardware implementation

### 5.1 Notion of time

The synchronous design approach basically relies on a periodic trigger to tackle the fundamental design problem described in Section 2. The benefit of an accurate trigger is the possibility to derive a timebase. By means of a known oscillator frequency, the synchronous design is able to communicate at a specific baud rate, or even generate time-scheduled events. On the other hand, asynchronous designs are independent of a clock source, at the cost of a missing timebase. There is no relation to real time.

What can be done is to derive a time base, based on the speed of an internal oscillation. This oscillation can be achieved with a feedback path (loop). The precision of the oscillation depends on the jitter of the loop. The variation may be caused by different execution times depending on data and operation as well as temperature drift, which leads to different gate delays. Two different solutions have been taken into account to generate such an oscillation for the aEUART:

1. Use a separate dedicated feedback circuit.
2. Use the oscillation of the entire circuit.

The first solution amounts to a simple circuit containing a register with a feedback path. The oscillation is quite fast and stable due to the short data path with predictable delay. Difficulties may arise when integrating the oscillation circuit in the aEUART because two different oscillation speeds are combined (oscillation circuit vs. aEUART oscillation) in a single design. Moreover, additional hardware and glue code would be required to dissolve phase conflicts.

Within the scope of the second solution, the oscillation of the entire aEUART is used to derive a time base. This solution has been chosen for the implementation, its considerable advantage being easy integration, since the present synchronous EUART can be adapted in a straight forward manner and no additional gates are required. One disadvantage of this solution is a long data path that leads to greater jitter and slower oscillation. Oscillation speed is not critical with respect to relatively slow serial baud rates. The jitter (imprecise oscillation) has to be tolerated by the circuit. The long-term drift rate of the oscillation has to be handled by means of re-synchronisation as explained in Section 2.3 .

In order to behave according to a specific timing, an external reference signal is necessary. The duration can be internally measured by a counter incremented every oscillation tick. The resulting counter value forms the relation between internal ticks and time durations. The aEUART gathers these reference signals from the serial bus line driving the TTP/A or LIN protocol with synchronisation patterns. The synchronisation process of the EUART is described in Section 2.3.

## 5.2 Impact of FSL on the environment

The aEUART is built as an extension module, connected via a small memory mapped interface, as described in Section 2.3. This interface contains FSL coded data and leads through the handshake line to allow the host to partici-

pate in the handshake process of the aEUART module. Usually, the aEUART module is connected to the asynchronous SPEAR processor [6]. Together, the processor and the aEUART constitute one asynchronous system.

For verification purposes, the aEUART is used as a stand-alone application. Therefore, a synthesise-able wrapper has been written to interface the aEUART. The input signals have to be provided as FSL signals in the alternated phase to the preceding phase. Thus, we only have to inspect the handshake signal coming from the aEUART which tells us what kind of phase was captured last, in order to generate data in the new phase. As soon as the inputs are captured, the handshake value changes and all aEUART input signals are coded in the new phase, regardless if there is new semantic meaning or not. The wrapper always provides the right phase to the aEUART and can be seen as a non-blocking source.

## 5.3 Impact on VHDL description

The asynchronous design entry presents us with a slightly different coding style. The two different coding sets have to be handled within the source code. In addition to the data path, the handshake lines must be connected, phase inverter placed and finally the registers initialised in the right phase to get the circuit working. At first sight this looks like a big overhead next to synchronous circuits but due to the coding guidelines in Section 5.3.2, together with the FSL event simulation, it gets a manageable task.

### 5.3.1 Juxtaposition of synchronous and asynchronous design entry

In the following, some major differences

**Clock:** The well known clock event is a key part of synchronous designs. The prepared data is captured by the register upon the rising edge of a

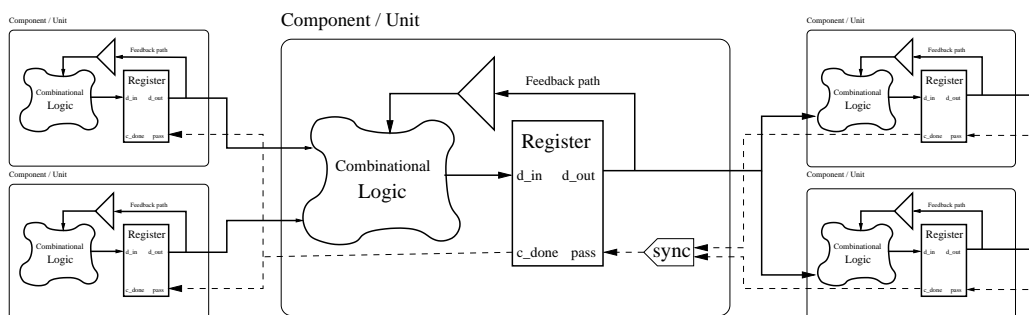


Figure 5.1: Concept of design

periodic input signal. The key part of asynchronous design is constituted by a handshake protocol. The register captures data according to a number of rules explained in the Paragraph “Registers” in Section 2.2.2.

**Feedback of signals:** In Section 2.2.2, placement of phase inverters has been discussed. However, we discuss the phase inverter in combination with state machines, as they are used many times. Most designs make use of a “Mealy” state machine, which means that output is generated as a function of the inputs and the current state. Reading back the current state stored in the register leads to a feedback path, that starts from the register’s output and goes back to the combinational logic (see Figure 5.1). The current state has to be phase-inverted (by the inverter in the feedback path) when being read for any conditional decisions. It may be tempting to directly use the state (and any other signal) at the register output in the logic cloud in front of the register, like in synchronous designs. Such a procedure would combine signals of different phase encoding and lock the circuit.

**Arithmetic expressions:** The behaviour of combinational logic in FSL, such as AND, OR and INV gates is implemented in the FSL library. Their behaviour is particularly adapted to process consistent input data or to otherwise hold the old output. At the current version of the library, arithmetic

symbols like “+” or “-” are not supported. To avoid standard logic implementations in the synthesis process, the adder has to be built out of basic gates in the FSL library. Under these circumstances, a ripple carry adder has to be instantiated every time an addition or a subtraction is required. If a subtraction is desired, the “two’s complement” can be used.

**Simulation aids:** As stated in Section 2.2.2, combinational logic evaluates new output if the inputs are consistent. This behaviour is implemented in the FSL library. In the case of behavioural simulation, no FSL gates are used, except for the register, because it is explicitly instantiated. In order to allow behavioural simulation of FSL circuits and high level design entry by using “if”, “case”,... statements, we have to ensure that the simulator executes these statements only if the processed data is consistent. This is what the function “stable(input\_signals)” is used for. The function is put at the beginning of a VHDL process. The vector “input\_signals” keeps a concatenation of all process input signals. The sensitivity list has to be empty. The process is executed immediately and suspended at the wait statement contained in the “stable()” function. As soon as consistency is established, the simulator continues process execution right behind the stable() function. The process is looped from the “begin” to the “end” statement. The source code in Listing 5.1 provides an example. Note that the “stable()” function matters behavioural simulation only. It can not be synthesised and there is no sense to do so; the consistency check is implemented in the FSL gates.

**Handshake signals:** The FSL approach uses handshake signals to achieve data flow control. These signal lines must be routed between the registers. This, to synchronous designers unfamiliar, step, has to accompany the design cycle of data flow, since every downstream stage needs to communicate its readiness to the upstream register. Potential inversions of the handshake lines may be necessary. Furthermore, the handshake lines have to be and-ed together with hysteresis in the case of multiple downstream registers. As

---

```
USE work.cal.all;
USE work.cal_beh.all;

architecture behaviour of sample_ent is

signal all_process_inputs : cal_logic_vector(5 downto 0);
signal input1 : cal_logic_vector(1 downto 0);
signal input2 : cal_logic;
signal input3 : cal_logic_vector(2 downto 0);
signal myoutput : cal_logic;

begin --behaviour

    all_process_inputs <=  input1    --#2
                        & input2    --#1
                        & input3    --#3

    combinational: process( )
    begin
        --synopsys synthesis_off
        stable(all_process_inputs); -- wait until vector is consistent
        --synopsys synthesis_on

        if ((input1 = 'HL' and input2 = 'H') or input3 = 'LHL') then
            myoutput <= 'H';
        elsif ((input1 = 'hl' and input2 = 'h') or input3 = 'lhl') then
            myoutput <= 'h';
        end if;
    end process combinational;

end behaviour;
```

---

Listing 5.1: VHDL example of the “stable()” procedure (Paragraph “Simulation aids” in Section 5.3.1)

---

```
USE work.cal.all;
USE work.cal_beh.all;

architecture behaviour of receiver_ent is

begin --behaviour

    handshake_sync: process(c_done_uartctrl_unit , c_done_ebr_generator_unit_conv)
    begin
        -- Synchronising the c_done signals from multiple downstream
        -- registers (2 in our case)
        if c_done_uartctrl_unit = '1' and c_done_ebr_generator_unit_conv = '1' then
            pass_receiver_unit <= '1';
        end if;

        if c_done_uartctrl_unit = '0' and c_done_ebr_generator_unit_conv = '0' then
            pass_receiver_unit <= '0';
        end if;
    end process handshake_sync;

end behaviour;
```

---

Listing 5.2: VHDL example of handshake synchronisation (Paragraph “Handshake signals” in Section 5.3.1)

soon as all downstream registers have consumed the data, a new data wave can be issued. Otherwise losslessness is not provided anymore. An example of the combination of multiple handshake lines is presented in Listing 5.2.

As we use multiple registers to achieve one huge register of the desired data width, the outgoing handshake lines of the registers have to be and-ed together with hysteresis too, in order to match the abstraction of one big register. The coding example is given in Listing 5.3.

### 5.3.2 Coding guidelines

There are many different ways of describing the same thing. The following guidelines have been extracted from the experience collected during the implementation process of the asynchronous EUART. These guidelines are

---

```
USE work.cal.all;
USE work.cal_beh.all;

architecture behaviour of receiver_ent is

begin --behaviour

    handshake: process (c_done1, c_done2, c_done3)
    begin

        -- Synchronising c_done signals of a huge FSL register consisting
        -- of multiple FSL register instances
        if c_done1 = '1' and c_done2 = '1' and c_done3 = '1' then
            c_done <= '1';
        end if;

        if c_done1 = '0' and c_done2 = '0' and c_done3 = '0' then
            c_done <= '0';
        end if;
    end process handshake;

end behaviour;
```

---

Listing 5.3: VHDL example of combining multiple handshake outputs (Paragraph “Handshake signals” in Section 5.3.1)

intended to serve as a solid basis to keep the source code readable and consistent in terms of structure. There are some common VHDL parts which are most likely necessary in every design component. The guidelines fit the theoretical concept shown in Figure 5.1.

**$\phi$ -detection:** All entity inputs of a design component are used to detect the current input phase. If the inputs are consistent the phase information is stored in the signal “phi\_value”. Otherwise, the signal keeps the old phase value.

In Listing 5.4, the function “fi\_det(all\_entity\_inputs,phi\_value)” demands a signal or a vector as the first parameter and a “phi\_value” which contains the current phase. If the first parameter contains inconsistent signals, the second parameter is returned. Otherwise, the phase value of the vector/signal is returned. As used in the aEUART implementation, the signal used to keep the return value (“phi”) is used as the second parameter of the function too. This construction is used to keep the old phase information as long as the inputs are inconsistent. It is very practical to write the signal widths in comment when concatenating signals. This helps to figure out the vector width in case of length mismatch.

The second process reads the “phi\_value” signal and assigns the corresponding value to CONST0 and CONST1. These can be used for short, whenever assigning a logical zero or a logical one. For any other values the “to\_cal\_logic(value,phi\_value)” function (presented in the next Paragraph) is appropriated, to convert a standard logic value to FSL logic in the matching phase of the current inputs.

**Converting standard logic values:** The HDL description of the behaviour of a design in FSL takes getting used to, since the behavioural description has to cover both of the two phase values (code sets). One possibility would be to double the code and replace the 'H' and 'L' values with the opposite code set, namely 'h' and 'l' (as done in Listing 5.1). However this

---

```
USE work.cal.all;
USE work.cal_beh.all;

architecture behaviour of demo_ent is

signal phi_value : cal_ctrl;
signal all_entity_inputs : cal_logic_vector(5 downto 0);
signal CONST1, CONST0 : cal_logic;

begin --behaviour

all_entity_inputs <=  input1    --#2
                    & input2    --#1
                    & input3    --#3

    phi_evaluation: process (all_entity_inputs , phi_value)
    begin
        phi_value <= fi_det(all_entity_inputs , phi_value);
    end process phi_evaluation;

    phi_assign: process (phi_value)
    begin
        if phi_value = '1' then
            CONST1 <= 'H';
            CONST0 <= 'L';
        elsif phi_value = '0' then
            CONST1 <= 'h';
            CONST0 <= 'l';
        end if;
    end process phi_assign;

end behaviour;
```

---

Listing 5.4: VHDL example of  $\phi$ -detection

would lead to cumbersome code and source code changes would be an error-prone task. Instead, one should use standard logic values and let a function translate into FSL values in the current phase, as presented in Listing 5.5. The function is called “to\_cal\_logic()” and uses two parameters. The first one is the standard logic value to be converted and the second one the phase value to be used for the coding.

The case statement requires constant values in the different branches. Therefore, the states are manually coded with constants in both phase values. Combining the “to\_cal\_logic()” function with the  $\phi$ -detection presented in the Paragraph before, enables the logic to handle both code sets while programming just one code line.

**Read back signals and registers:** Most of the time, it is necessary to read back signals going through the register. Therefore all signals used within the logic cloud in front of the register that come from this register output must be phase-inverted to avoid a phase mismatch. Fortunately, this conversion can be done comfortably with the help of a vector in one statement, preventing single signal inversions distributed in the source code. The synthesis tool should be configured to eliminate unused signals. Otherwise logic would be wasted for unused inverted signals. Listing 5.6 shows an example.

**Initialising FSL registers:** During reset, registers provide the output with initial values. This is the same behaviour as the one synchronous designs have. The initial value is defined by a generic parameter. Due to the fact that the “Synopsis Design-Compiler” [23] allows generics of integer type only, the register is currently limited in width to 31 bits<sup>1</sup>. The conversion from “standard logic” to “integer” is best done by using conversion functions such as shown in Listing 5.7.

---

<sup>1</sup>The MSB carries the sign and therefore is not used.

---

```

USE work.cal.all;
USE work.cal_beh.all;

architecture behaviour of uartctrl_ent is

  constant ST_SYNC : std_logic := '0';
  constant ST_READY : std_logic := '1';

  constant ST_SYNC_I0 : cal_logic := '1';
  constant ST_READY_I0 : cal_logic := 'h';

  constant ST_SYNC_I1 : cal_logic := 'L';
  constant ST_READY_I1 : cal_logic := 'H';

  —for coding more than two states, use cal_logic_vector data type

  signal all_entity_inputs : cal_logic_vector(5 downto 0);

  begin —behaviour

    all_process_inputs <= actual_state_conv
                          &skip_flag
                          &dummy_counter;

    combinational: process ( )
    begin

      —synopsys synthesis_off
      stable(all_process_inputs); — wait until vector is consistent
      —synopsys synthesis_on

      case actual_state_conv is — the phase inverted actual_state
        when ST_SYNC_I0 | ST_SYNC_I1 => —match both phase values
          TimCtr_next      <= to_cal_logic(TIMERSYNC, phi_value);
          actual_state_next <= to_cal_logic(ST_READY, phi_value);
          go_next          <= CONST0; —short cut for to_cal_logic('0', phi_value)
        when ST_READY_I0 | ST_READY_I1 =>
          actual_state_next <= to_cal_logic(ST_SYNC, phi_value);
          if (skip_flag = CONST1) or (dummy_counter = to_cal_logic(counter, phi_valu
          ready_flag_next  <= CONST1;
        when others =>
          actual_state_next <= to_cal_logic(ST_SYNC, phi_value);
        end case;
      end process combinational;

  end behaviour;

```

---

Listing 5.5: VHDL example of the conversion of “std\_logic” to “cal\_logic”

---

```

USE work.cal.all;
USE work.cal_beh.all;

architecture behaviour of test_ent is

signal reg, reg_conv, reg_next : cal_logic_vector(96 downto 0);

begin --behaviour

    combinational: process( )
    begin

        --synopsys synthesis_off
        stable(all_process_inputs); -- wait until vector is consistent
        --synopsys synthesis_on

        if actual_state_conv = to_cal_logic(ST_SYNC, phi_value) then
            next_state <= to_cal_logic(ST_READY, phi_value);
        end if;
    end process combinational;



---


-- register inputs



---


    reg_next(15 downto 0) <= RecMsgNxt;           --#16
    reg_next(31 downto 16) <= IORegEubrsNxt;      --#16
    reg_next(47 downto 32) <= IORegTIMER_Nxt;    --#16
    reg_next(63 downto 48) <= IORegTsTmNxt;      --#16
    reg_next(64) <= next_state;                    --#1



---


-- register outputs



---


    reg_conv <= phi_conv(reg); --because the register output is fed back
                                     --to the combinational input logic
                                     --that is present in the new phase

    RecMsg_conv <= reg_conv(15 downto 0);
    RecMsg      <= reg(15 downto 0);

    IORegEubrs_conv <= reg_conv(31 downto 16);
    IORegEubrs     <= reg(31 downto 16);

    IORegTIMER_conv <= reg_conv(47 downto 32);
    IORegTIMER     <= reg(47 downto 32);

    IORegTsTm_conv <= reg_conv(63 downto 48);
    IORegTsTm     <= reg(63 downto 48);

    actual_state_conv <= reg_conv(64);
    actual_state     <= reg(64);

end behaviour;

```

---

Listing 5.6: VHDL example of read back signals

---

```

USE work.cal.all;
USE work.cal_beh.all;

architecture behaviour of uartctrl_ent is

signal reg, reg_conv, reg_next : cal_logic_vector(96 downto 0);
constant init_value : cal_logic_vector(61 downto 0) :=

'L'           --TrCtr #1
&(not EXT_ACT_I1) --IOEna #1
&"LLLLLLLLLLLLLLLL" --IOReg(EXTCONFIG) #16
&"LLLLLLLLLLLLLLLL" --IOReg(DATA0) #16
&'L'         --ConfWrite #1
&'L'         --enaflag #1
&'L'         --EbrCtr #1
&NOACTION_I1 --RecCtr #2
&'L'         --Go #1
&'L'         --ClrConfWrite #1
&'L'         --StartRec #1
&TIMERRESET_I1 --TimCtr #3
&ST_READY_I1  --actual_state #1
&"LLLLLHLLLLLLLLHL" --IOReg(EXTSTATUS)#16

constant init_value_int_1 :=
  conv_integer(unsigned(to_std_logic(init_value(30 downto 0))));
constant init_value_int_2 :=
  conv_integer(unsigned(to_std_logic(init_value(61 downto 31))));

begin --behaviour

  cal_reg1: cal_reg_with_init
    generic map (
      w           => 31,
      phi_value  => 0,           -- 0 = h / l , 1 = H / L
      init_value => init_value_int_1)
    port map (
      d           => reg_next(30 downto 0),
      q           => reg(30 downto 0),
      c_done     => c_done1,
      pass       => pass,
      reset      => reset);

  cal_reg2: cal_reg_with_init
    generic map (
      w           => 31,
      phi_value  => 0,           -- 0 = h / l , 1 = H / L
      init_value => init_value_int_2)
    port map (
      d           => reg_next(61 downto 31),
      q           => reg(61 downto 31),
      c_done     => c_done2,
      pass       => pass,
      reset      => reset);

```

---

Listing 5.7: VHDL example of initialising FSL registers

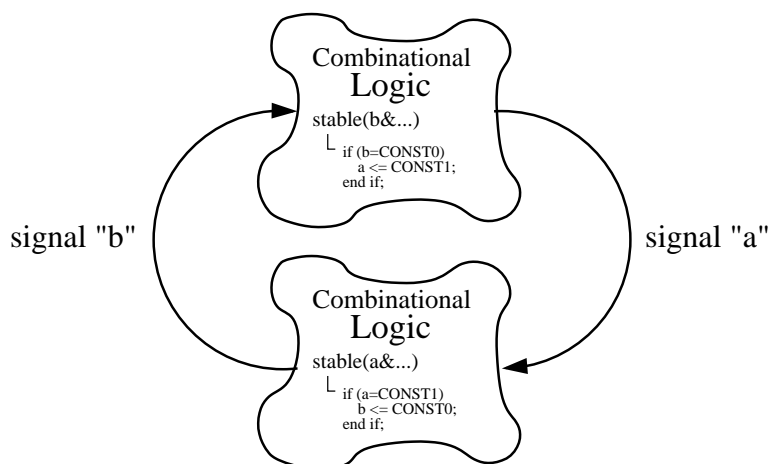


Figure 5.2: Dependencies between combinational processes

**Inter-process communication:** In Figure 5.2, two combinational logic processes communicate by means of the signals “a” and “b”. Each process is modelled in VHDL, beginning with the “stable()” function which is explained in Section 5.3.1. Signal “a” and “b” are not fed by a register and therefore not defined during simulation reset. As a consequence, the parameter of the “stable()” function is inconsistent for both processes and none of them is executed. Furthermore, signal “a” and “b” do not get defined because they are set inside the processes which are never executed. The simulation fails. The synthesised hardware will drive unpredictable values for signal “a” and “b”, either in phase  $\phi_0$  or  $\phi_1$ . There is a risk of locking up the circuit due to inconsistent data. In order to avoid this, such signals should be routed across a register providing initial values.

### 5.3.3 Pitfalls

**Stacking FSL registers:** Due to restrictions in the initialisation (see Paragraph “Initialising FSL registers” in Section 5.3.2) of the current implementation, an FSL register can dispose of a maximum width of 31 bits. Often

more bits are necessary and therefore multiple register instances are used in parallel for the abstraction of one huge FSL register. In general this should be fine with the circuit. However, special care has to be taken when feedback paths of one register instance are directly (except for the necessary phase inverter) coupled back to the input of a register which is connected in parallel. In this particular case, the individual registers may lock up one another due to individual switching instants. Illustrated in Figure 5.3: Most of the signals coming from the registers' outputs are linked back to the logic cloud. One of the signals is linked directly to Reg1b just to delay the signal by one data wave, for example. Each register evaluates the switching condition independently. Given that Reg1b fulfils the switching condition and captures the input data, the inputs at Reg1a are still inconsistent because of some slow data paths in the logic cloud. In the meantime, the direct feedback path is propagating a new phase value to Reg1a, right before having fulfilled the switching condition. By now, the slow data paths coming from the logic cloud have caught up and are consistent with the other inputs, except for the direct feedback path, present in the new phase. The circuit will stop to work. The question arising in this context is whether this could also happen to just one FSL register. The root of the problem is the independent evaluation of the switching conditions. The time elapsed until both registers have captured the inputs is the unknown critical delay, within which direct feedback paths can lock up the circuit. We are raising the topic of direct feedback paths, because signals fed back to the logic cloud are not affected. This is due to the fact that they are blocked by the logic gates for the reason of inconsistency until the upstream register is triggered to deliver new data. This, however, happens only after both registers in Figure 5.3 have captured the input data. The same theory can be applied to one FSL register shown in Figure 5.4, with one significant difference though: The critical delay exists from the point of time the register gets transparent until all signals have been latched and the inner loop closes the gate. Over this period of time,

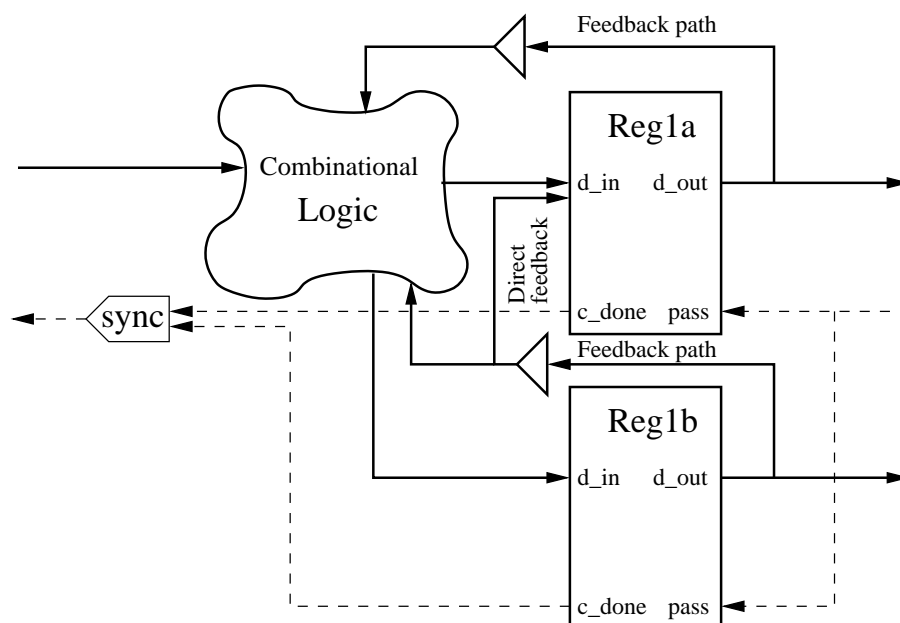


Figure 5.3: Stacking FSL registers

the inputs have to stay consistent. As the inner loop can be matched in order to be faster than the outer loop, there is no risk for single registers. One potential workaround for stacking FSL registers is to evaluate the phase of all register instances together. If the switching condition is fulfilled, the capture procedure begins at all instances concurrently, similarly to a single FSL register.

**Signals and the “stable()” function:** Signals are updated after some delay. Sometimes it is necessary to use the following constructs in a VHDL “process” block:

```
fooxt <= CONST1;
```

```
varA <= foonxt;
```

The simulation engine assigns “CONST1” to the signal “fooxt” in a first process run. In a second run, the value is propagated to the signal “varA”.

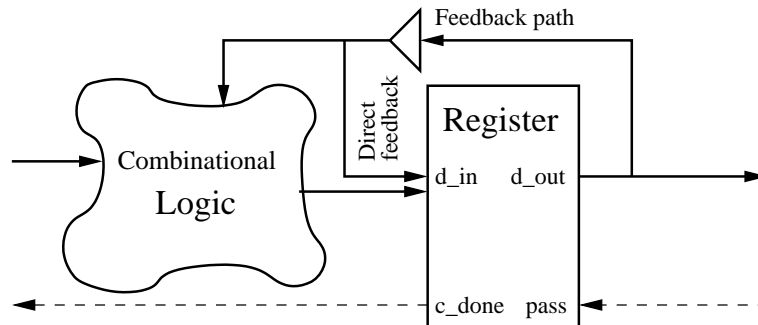


Figure 5.4: Direct feedback path

The “*stable()*” procedure triggers the process just once and then waits until one of the inputs changes its value. This leads, for example, to an undefined value of “varA” that blocks the connected circuit behind. One solution to this problem consists in using a variable instead of a signal for “foontxt”, if possible. A workaround for simulation purposes would be to put a “**wait for 1 ns;**” behind the “foontxt <= CONST1;” statement to force the simulation engine to update the signals.

# Chapter 6

## Design Flow

This Chapter describes the design flow used for the simulation and synthesis of the asynchronous EUART design. The design flow is based on Synopsis [23] software running on a SunOS platform. In order to operate with FSL gates, a set of “awk” and “sed” scripts performing modifications on the design files are required. The toolchain has been developed during the aSPEAR<sup>1</sup> project [13][3].

After the design entry has been made, the design flow basically consists of behavioural simulation, the first synthesis step, functional simulation, the second synthesis step, pre-layout simulation, Place&Route and, finally, post-layout simulation. The structure of the design flow uses a strict folder system to keep the source files and the corresponding generated files organised. This allows for an automatised simulation and synthesis process. An introduction to the design flow and folder system can be obtained from [12]. In the following, we are going to introduce the (makefile) targets provided by the toolchain. A target is composed of the component name and the action to be performed, separated by a dot. The top-level component of the aEUART design is called “top\_ent”, which is a synthesise-able wrapper for the aEUART

---

<sup>1</sup>asynchronous Scalable Processor for Embedded Applications in Real-Time Environments

design. The available targets are:

- top\_ent.beh
- top\_ent.cal2rail
- top\_ent.func
- top\_ent.railsyn
- top\_ent.pre
- top\_ent.post

They are discussed in the following two Sections.

## 6.1 Simulation

In order to perform the behavioural simulation, the target “top\_ent.beh” has to be called inside the “sim” directory (“make top\_ent.beh”). The behavioural description of FSL can be found in the folder “cel\_beh” which is used for behavioural simulation. The functional simulation uses the file “top\_ent\_rail.vhd” generated in the course of the first synthesis step. As the type of the ports has been modified (dual rail encoding), an automatically inserted layer is in charge of the mapping. Functional simulation is performed by calling the target “top\_ent.func”. The target “top\_ent.pre” performs the pre-layout simulation. The second synthesis process has to proceed the pre-layout simulation, to feed the simulator with the generated file “top\_ent\_rail\_pre.vhd”, containing the target technology.

## 6.2 Synthesis

In contrast to synchronous designs, the FSL design flow needs two synthesis steps. This is due to the FSL coding scheme. Standard logic designs make use

---

of two-valued single rails to represent the values of HIGH and LOW. The FSL design utilises four states to represent these two values. The first synthesis step is executed by calling the target “top\_ent.cal2rail” in the folder “calsyn”, resulting in a generic hardware including four state logic. Each single four state logic rail is replaced by two dually coded rails. In the following second synthesis step, the design is mapped to the target technology. In our case, this is the “Altera APEX20KC” FPGA. The second synthesis step is executed by calling the target “make top\_ent.railsyn” in the folder “syn”. “Place and Route” is taken out by the “Altera Quartus II” software package.

# Chapter 7

## Experimental results

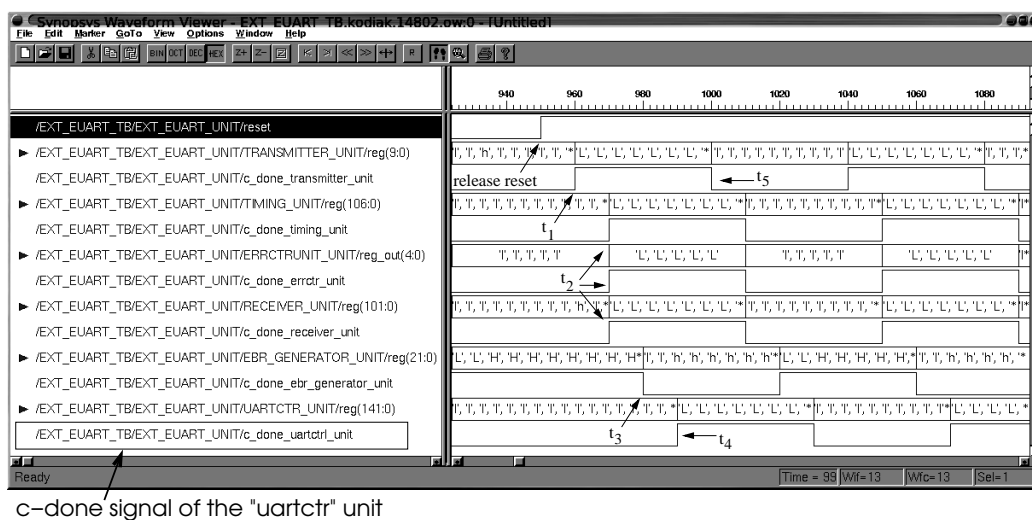
The synchronous EUART has been ported to an asynchronous implementation, in consideration of the discussed topics in Chapter 5. The pipeline registers have been located and initialised according to Section 3.7. The design has been synthesised by the mentioned design flow (Chapter 6) and the generated netlist has been fed to “Quartus II”, which is the Place&Route front-end of the “Apex20KC” FPGA. Whereas the synchronous EUART version requires only 699 LEs<sup>1</sup> [4], the asynchronous version occupies 21235 LEs, and thus about 30 times as much.

### 7.1 Simulation results

Figure 7.1 shows the behavioural simulation of the asynchronous EUART. The switching sequence of the registers can be observed as simulated with the help of FSL event simulation (see Section 3.7): The transmitter unit takes over data at  $t_1$ . Subsequently, the timing-,errctr- and receiver unit accept new data at instant  $t_2$ . The ebr\_generator unit depending on the receiver unit, it captures next at  $t_3$ . Finally, the uartctr unit captures at instant  $t_4$  and a new cycle starts over with the transmitter unit at  $t_5$ .

---

<sup>1</sup>Logic Element is the smallest programmable unit within Altera’s FPGA



c-done signal of the "uartctr" unit

Figure 7.1: Behavioural simulation of the “aEUART” - switching sequence

A complete synchronise- and receive procedure of the aEUART is shown in Figure 7.2. First the aEUART detects a valid synchronisation pattern. The length of a bit cell is counted in relation to the switching speed of the whole aEUART (as we decided to use the whole circuit oscillation to derive a time base in Section 5.1) and the EUBRS value is derived. The baud rate generator (ebr\_generator) generates the timing, based on the EUBRS value in order to receive the message on the serial bus. When the stop-bit is detected, the message is transferred to the message register of the I/O interface. Both Figures (7.1 and 7.2) state the results of behavioural simulation. Thus the timing information is not essential.

The post-layout simulation, including the timing information of the Place&Route process, is presented in Figure 7.3. The shown “c-done” signal of the “uartctr” unit states the switching activity of the register chain. In between every transition pair of the waveform, a complete capture sequence of the registers occurs. For example, the “uartctr” unit captures a  $\phi_1$  data wave at the first

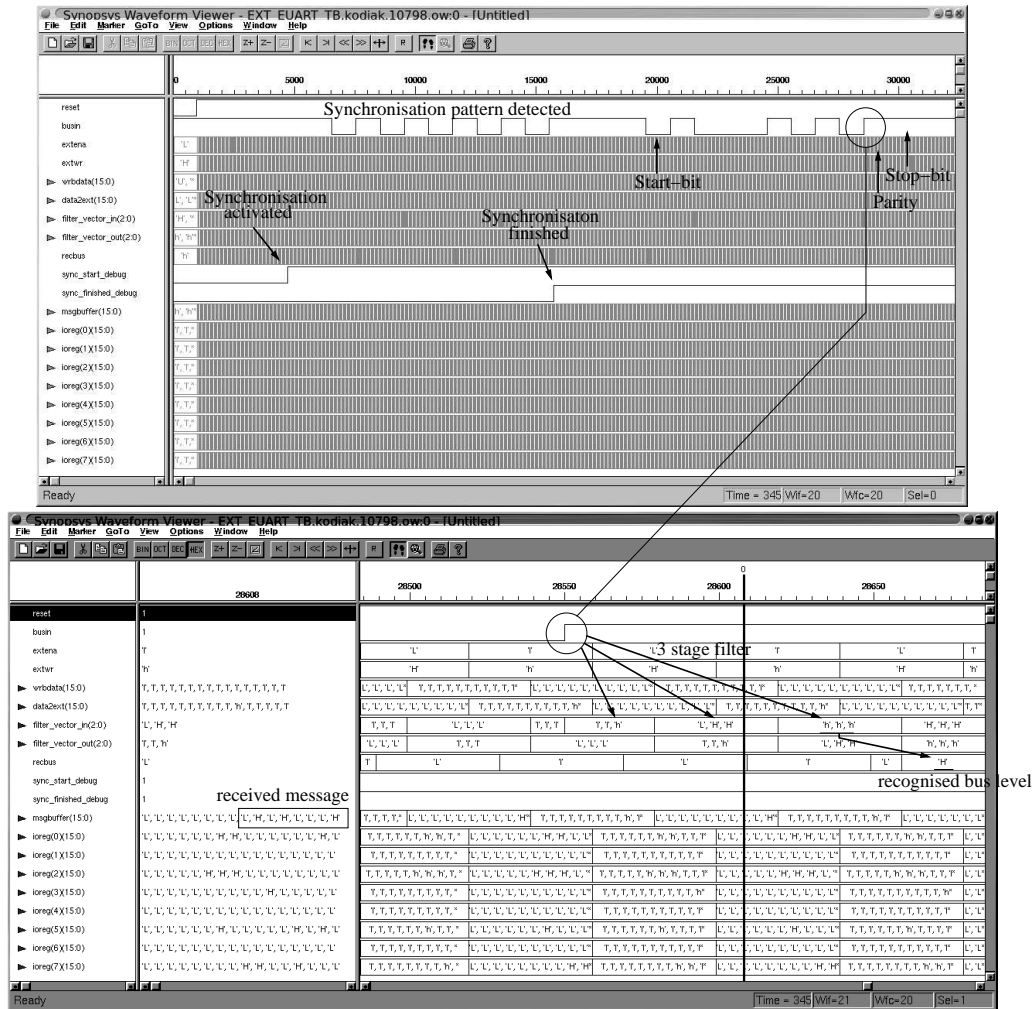


Figure 7.2: Behavioural simulation of the “aEUART” - sync. and receive

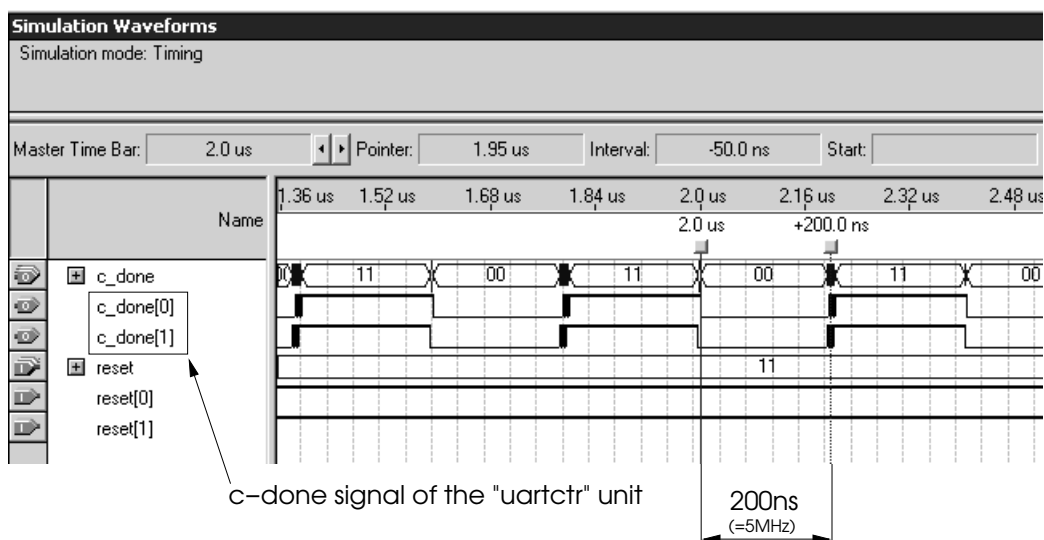


Figure 7.3: Post-layout simulation of the “aEUART” - self-oscillation

ruler (2us) and acknowledges by setting “c\_done” to “00”<sup>2</sup>. Subsequently, the transmitter-, timing-, errctr-, receiver- and ebr\_generator unit capture new data (not visible in the post-layout simulation). After 200ns it is again the turn of the “uartctr” unit to capture new data. Thus the aEUART works with a speed comparable to a synchronous clock with 5MHz.

## 7.2 Physical results

The function of the asynchronous EUART has been verified via functional simulation. The target of the physical verification is to determine the qualification of the aEUART for the TTP/A and LIN architecture. We are interested in the jitter as well as in the long-term drift of the self-oscillation. Both effects are undesired and may break up communication. Therefore a test set-up has been installed as described in the next Section. The results

<sup>2</sup>The handshake signal is two-valued (‘0’,‘1’). However, the implementation uses two rails for the handshake signal, both carrying the same logical value.

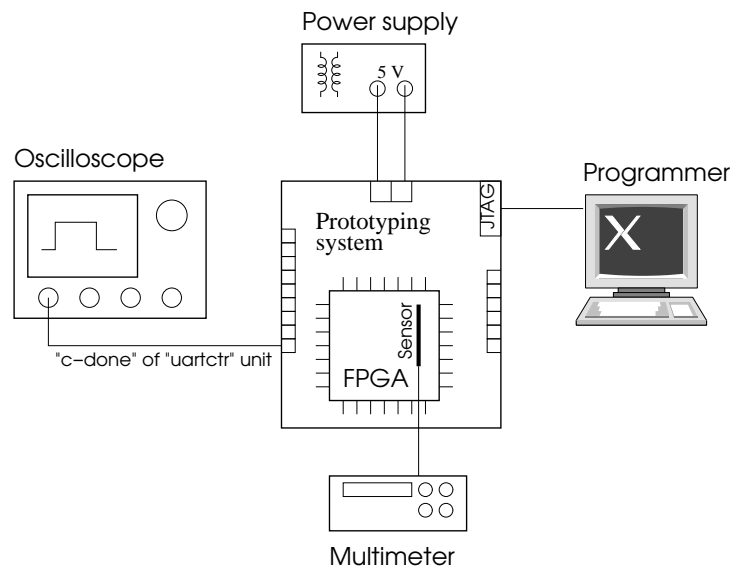


Figure 7.4: Test set-up

are presented thereafter.

### 7.2.1 Test set-up

As shown in Figure 7.4, the test set-up consists of the following components:

- El Camino “DIGILAB megAPEX” prototyping system [8] based on the APEX20KC FPGA by Altera.
- Tektronix TDS3034 oscilloscope
- Agilent 34411A multimeter with 2-wire heat sensor.
- Laboratory power supply

The aEUART design has been downloaded to the non-volatile memory of the prototyping board for a fast configuration process of the FPGA. A 2-wire temperature sensor has been attached to the surface of the FPGA to record the package temperature.

### 7.2.2 Measurement of the jitter

“Jitter” is the distance between the shortest and longest period of a clock signal. In the asynchronous EUART design, a clock cycle consists of the sequence of registers, each capturing data exactly once. In order to measure the self-oscillation we examine the handshake line (“c-done”) of a dedicated register. As internal signals of the FPGA are difficult to access, focus is kept on the handshake line of the “uartctr” unit.

Figure 7.5 shows the “c-done” signal of the “uartctr” unit as measured by the oscilloscope. The period varies within the range of 152 and 158ns, connoting a jitter of 3,78% at an FPGA package temperature of 33°C. The average frequency amounts to 6,45MHz, which in fact is faster than the post-layout simulation states.

The jitter directly affects the receive/transmit process. In contrast to synchronous designs, the oscillation frequency is a result of data path delays, depending on the actually performed operation. Thus we assume the following worst case scenario in order to see the effect of jitter: The aEUART synchronises to the serial bus while its oscillation is stable at 6,329MHz (period of 158ns). Afterwards, the receive mode is activated and the oscillation frequency switches to 6,578MHz (period of 152ns). The length of a bit cell is shortened by 3,78%. Given a message length of 10 bits, the total offset accumulates to 37,8%. The aEUART communication works fine as long as the offset stays below 50% (half a bit cell), as shown in Figure 7.6. The over-sampling mechanism correctly recognises the bit cells, since the majority of samples overlaps with the corresponding bit cell on the bus (assuming error-free transmission). The maximum count of bits  $n$  per message is limited by the following relation:  $jitter[\%] * n < 50\%$

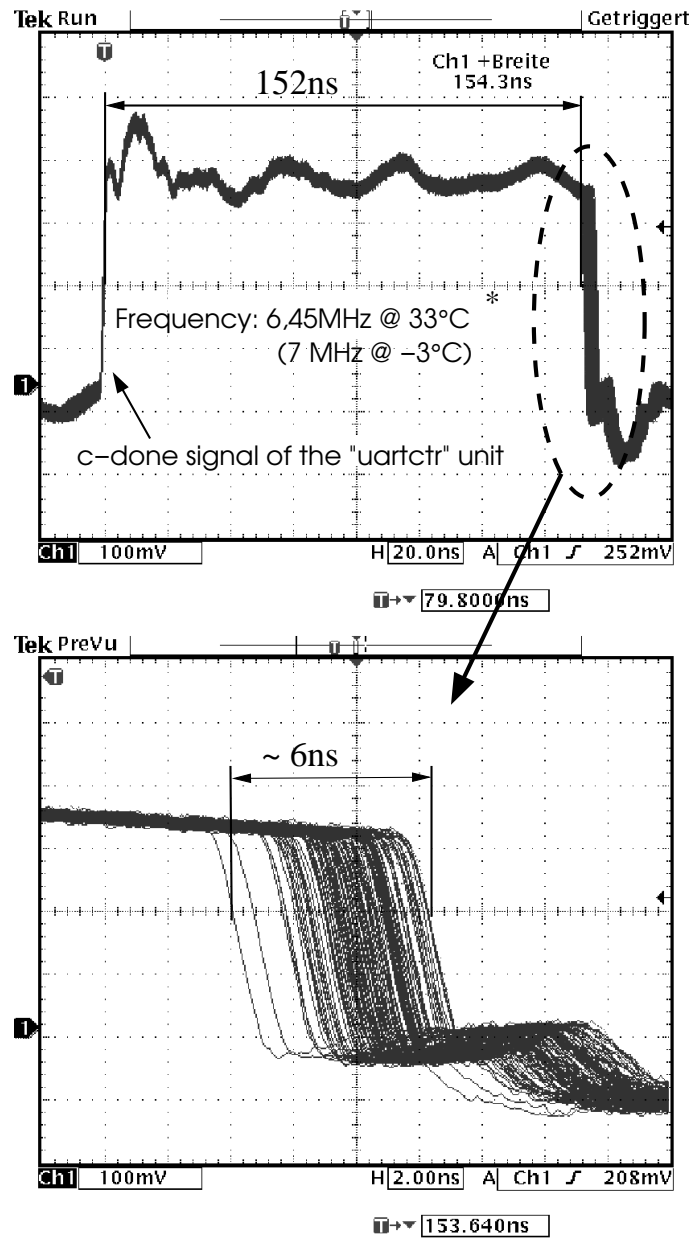


Figure 7.5: Scope-view of the “aEUART” - self-oscillation

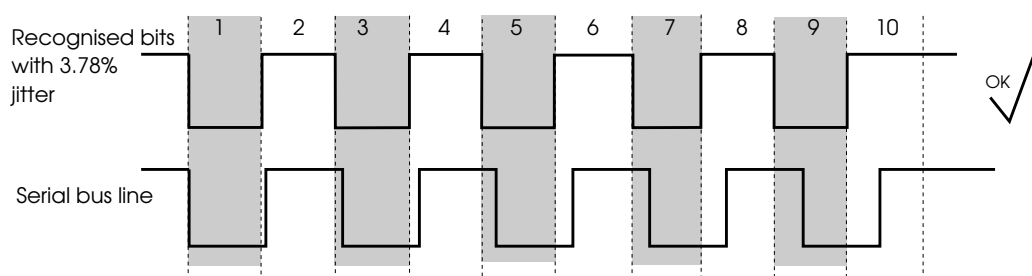


Figure 7.6: Effect of jitter.

### 7.2.3 Measurement of the long-term oscillation drift

Cheap oscillators possess a high drift rate, resulting in a baud rate drift and finally in a communication error. Temperature drift is one reason for oscillation drift. In order to see how the aEUART reacts to oscillation drift, the prototyping system is unplugged from the power source and put into a freezer. After the cooling process, timing measurements are performed immediately after power-up. The package temperature amounts to  $-3^{\circ}\text{C}$ . The period varies within the range of 140 and 146ns. Again, the jitter equals 6ns. The cold has caused an oscillation drift ranging from 6,45MHz up to 7MHz (7,74%). Thus the length of a bit cell is shortened by 7.74%. Communication fails, since the offset of the bit cells exceeds 50% by the seventh bit of a message, as illustrated in Figure 7.7. Therefore, the long-term oscillation drift is compensated by the resynchronisation technique described in Section 2.3.

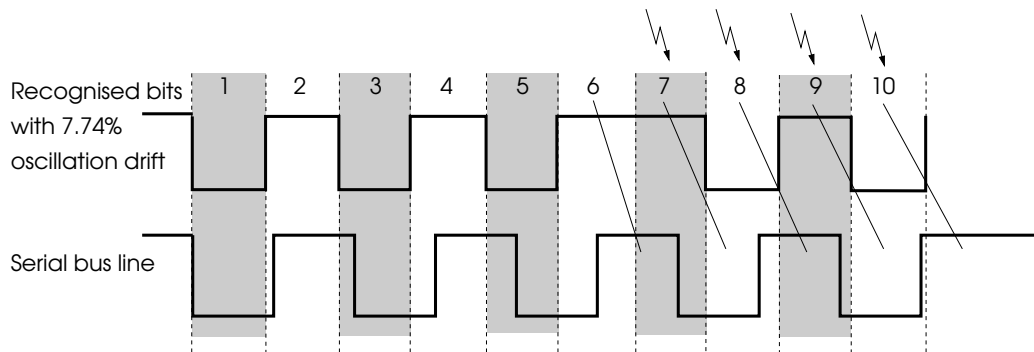


Figure 7.7: Effect of oscillation drift.

### 7.3 Simulation vs. physical results

The post-layout simulation states an oscillation frequency of 5MHz. The physical oscillation frequency amounts to 6,45MHz, which equals an error of 22%. This is due to the worst-case timing assumption of the simulator. Furthermore, the real hardware conditions differ from the timing model used in the simulation.

# Chapter 8

## Conclusion and outlook

Asynchronous logic is a promising design technique, solving most of the problems occurring in synchronous designs in a natural manner. The implementation process of the asynchronous EUART demonstrated the high complexity of asynchronous circuits. The designer needs a good understanding of the control and data flow. Introducing FSL event simulation turned out to be an essential part of the design process to overcome design complexity; considerations related to the FSL control/data flow are separated from the actual component implementation and verified by simulation. The current FPGA technology seems to be quite unsuited for asynchronous designs, leading to a multiple of the logic implied by its synchronous counterexample. Moreover, software systems for asynchronous circuits are still at an early stage of development. Possibly, future synthesis software will support the asynchronous design paradigm the same way nowadays' tools support the synchronous design flow. The asynchronous EUART serves as a starting point for the "ARTS"<sup>1</sup> project [9]. The main goal of the ARTS project is to investigate whether asynchronous logic is suitable for the use in (hard) real-time systems. The practical part of the work consists of an asynchronous TTP-controller.

Information concerning ongoing activities within the scope of the asyn-

---

<sup>1</sup>Asynchronous Logic in **Real-Time Systems**

chronous EUART can be consulted at the aEUART project homepage [14]:

**<http://trac.ecs.tuwien.ac.at/EUART>**

# Bibliography

- [1] J. R. Armstrong and F. G. Gray. *VHDL Design Representation and Synthesis*. Prentice Hall PTR, 2000.
- [2] A. Davis and S. M. Nowick. An Introduction to Asynchronous Circuit Design. Technical Report UUCS-97-013, Dept. of Computer Science, University of Utah, Sep 1997.
- [3] M. Delvai. *Design of an Asynchronous Processor Based on Code Alternation Logic - Treatment of Non-Linear Data Paths*. PhD thesis, Vienna University of Technology, January 2005.
- [4] M. Delvai, U. Eisenmann, and W. Elmenreich. Intelligent UART Module for Real-Time Applications. In *Proceedings of the First Workshop on Intelligent Solutions for Embedded Systems*, pages 177–185, Vienna, Austria, June 2003.
- [5] M. Delvai, U. Eisenmann, and W. Huber. Modular Construction System for Embedded Real-Time Applications. In *Austrochip Proceedings*, pages 103–109, Vienna, Oct. 2002.
- [6] M. Delvai, G. Fuchs, T. Handl, W. Huber, and A. Steininger. Design of an Asynchronous Microprocessor with Four-State Logic. *Austrochip 2005*, Oct. 2005.
- [7] U. Eisenmann. Universal Asynchronous Receiver Transmitter. Master’s thesis.
- [8] El Camino. <http://www.elcamino.de/prod.html>.
- [9] M. Furringer. The ARTS project homepage. <http://trac.ecs.tuwien.ac.at/ARTS>.

- 
- [10] R. Gallo, M. Delvai, W. Elmenreich, and A. Steininger. Revision and Verification of an Enhanced UART. <http://ti.tuwien.ac.at/ecs/research/publications>.
- [11] S. Ghosh. In Search of the Origin of VHDL's Delta Delays. In *International Symposium on Quality Electronic Design*, page 310, 2002.
- [12] T. Handl and W. Huber. Der CAL DesignFlow, Eine Einführung. Technical report, Vienna University of Technology.
- [13] W. Huber. *Design of an Asynchronous Processor Based on Code Alternation Logic - Exploration of Delay Insensitivity*. PhD thesis, Vienna University of Technology, May 2005.
- [14] W. Klein. The asynchronous EUART project homepage. <http://trac.ecs.tuwien.ac.at/EUART>.
- [15] H. Kopetz, W. Elmenreich, and C. Mack. A comparison of LIN and TTP/A. In *3rd IEEE International Workshop on Factory Communication Systems (WFCS 2000)*, 6.-8. September 2000, Porto, Portugal, pages 99–107, Sep. 2000.
- [16] H. Li. Security evaluation at design time for cryptographic hardware. Technical report, University of Cambridge, April 2006.
- [17] LIN. <http://www.lin-subbus.org>. Local Interconnect Network.
- [18] A. Peeters. The 'Asynchronous' Bibliography Homepage. <http://www.win.tue.nl/async-bib/async.html>.
- [19] A. Peeters. The 'Asynchronous' Bibliography (BIB<sub>T</sub>E<sub>X</sub>) database file `async.bib`. Corresponding e-mail address: `async-bib@win.tue.nl`.
- [20] D. J. Smith. *HDL Chip Design*. Doone Publications, 1996.
- [21] J. Sparsø and S. Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.
- [22] A. Steininger. Der Design-Flow eines ASIC. <http://ti.tuwien.ac.at>.
- [23] Synopsis. <http://www.synopsys.com>. Synthesis and Simulation Software.
- [24] The Embedded Computing Systems Group. <http://ti.tuwien.ac.at>.
- [25] TTP/A. <http://www.vmars.tuwien.ac.at/tpa/>. Time-Triggered Protocol for SAE Class A Applications.
- [26] XJ Technologies. <http://www.xjtek.org>. Simulation Software and Services.