FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Description Methods for Asynchronous Circuits — A Comparison

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Robert Najvirt
Matrikelnummer 0526813

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Dipl.-Ing. Jakob Lechner

Wien, 27.9.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Robert Najvirt
Gessayova 25
851 03 Bratislava

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

(Ort, Datum) (Unterschrift Verfasser)

Abstract

The advances of silicon manufacturing technology make it possible to integrate billion-transistor systems on a single die but the price to pay is higher parameter variability, resulting in problems with reliability, difficult clock distribution, high power consumption and more. Proponents of asynchronous circuits claim them to be a possible solution to most of these problems and they are indeed becoming an increasingly interesting design choice.

The aim of this work is to describe the most established description methods for asynchronous circuits, compare them in respect to a number of parameters and list their advantages and disadvantages. To further illustrate the differences, a simple ring topology network interface controller is described with all the considered methods and the implications of their use for the description are indicated. Next, to demonstrate the use of each description method in its typical field of application, selected parts of the faculty developed SPEAR2 processor are implemented with the methods.

As a result of this work a comparison of seven different description methods is now available that, unlike the existing literature so far, views these methods from the same angle, namely a carefully chosen set of criteria. This comparison, along with the selected demonstrative examples, could serve prospective designers of asynchronous circuits when choosing a description method for a project. At the same time, the extraction of the underlying concepts of the considered methods along with their comparison is also valuable for a didactic presentation of asynchronous design.

Kurzfassung

Die Fortschritte in den Siliziumherstellungsverfahren ermöglichen es, Systeme mit Milliarden von Transistoren auf einem einzigen Chip zu integrieren. Die Nachteile einer solchen Miniaturisierung sind höhere Parametervariabilität, die zu Problemen mit Zuverlässigkeit führt, erschwerte Taktverteilung, hoher Energieverbrauch und andere. Befürworter von asynchronen Schaltungen behaupten, diese seien eine mögliche Lösung von den meisten dieser Probleme und sie sind tatsächlich eine zunehmend interessante Designmöglichkeit.

Das Ziel dieser Arbeit ist es, die etabliertesten Beschreibungsmethoden für asynchrone Schaltungen zu beschreiben, sie anhand einer Anzahl von Parametern zu vergleichen und ihre Vorteile und Nachteile zu benennen. Um die Unterschiede besser zu verdeutlichen, wurde ein einfacher Ringnetzwerkkontroller mit allen den betrachteten Methoden beschrieben und auf die Konsequenz derer Benutzung für die Beschreibung hingewiesen. Um die Benutzung von Beschreibungsmethoden in ihren typischen Anwendungsgebieten zu zeigen, wurden zusätzlich gewählte Teile von dem auf der Fakultät entwickelten SPEAR2 Prozessor mit den Methoden beschrieben.

Als Ergebnis dieser Arbeit ist jetzt ein Vergleich von sieben Beschreibungsmethoden vorhanden, die diese Methoden, im Gegensatz zu der bestehenden Literatur, vom selben Blickwinkel betrachtet, und zwar durch sorgfältig ausgewählte Beurteilungskriterien. Dieser Vergleich mit den gewählten anschaulichen Beispielen kann einerseits zukünftigen Entwicklern von asynchronen Schaltungen bei der Wahl einer Beschreibungsmethode für ein Projekt von Nutzen sein. Gleichzeitig aber kann sie durch die Erfassung der grundlegenden Konzepte der betrachteten Methoden zusammen mit deren Vergleich wertvoll für didaktische Präsentationen von asynchronem Design sein.

Table of Contents

Introduction

Despite the fact that asynchronous circuits have been an active research area for decades, their commercial application remains marginal. Amongst other factors, this is because there is a multitude of description method each having different advantages and disadvantages and each being bound to a different tool. The choice, which description method to learn and which tool to invest in is even more difficult due to the uncertain future of both, description methods and tools.

However, with the advances in silicon manufacturing technology, making it possible to integrate billion-transistor systems on a single die for the price of higher parameter variability, fully synchronous systems are becoming increasingly inefficient and difficult to design. As a result, asynchronous circuits and systems are becoming an increasingly interesting design choice and impressive circuits from both, academia and industry demonstrate the potential of this design style.

The objective of this work is to give an overview of established description methods for asynchronous circuits, compare them in respect to a number of parameters and list their advantages and disadvantages.

Hopefully, it will be able to serve prospective designers of asynchronous circuits when choosing, which description method to learn. In addition, the findings could be a valuable source of information for a didactic presentation of asynchronous design.

The work is structured as follows:

As some readers might not be fully familiar with asynchronous design, Chapter 1 introduces some of its basic concepts, the understanding of which is assumed in the argumentation in the analysis, comparison and discussion. The experienced reader may skip this chapter.

Chapter 2 introduces the considered description methods and describes them to the extent that the reader would understand the approach and the underlying concepts.

To further clarify the use of the methods and visualise the fundamental differences between them a simple circuit is described with all the considered methods in Chapter 3 as a comparative example. A short discussion about the implications and possibilities of the methods for that particular circuit follows.

Chapter 4 gives a summary of properties of description methods, comparing them directly.

A second, design example for each of the methods to support the arguments from the analysis, comparison and discussion is presented in Chapter 5. Parts of a processor core will be chosen to show the use of a method in its suggested application domain.

Chapter 1

Asynchronous Circuits

This chapter serves as an introduction to asynchronous circuit design for those, who are not familiar with the concepts. First, motivation for choosing asynchronous circuits is given. Then, basic concepts such as timing models, signal coding, handshaking and typical building blocks will be described. Finally, advantages and disadvantages of asynchronous circuits as compared to synchronous ones will be presented. The information presented in this chapter is considered prerequisite knowledge in the rest of this work.

1.1 Why Asynchronous?

This whole work is about description methods for asynchronous circuits. But why to use asynchronous design in the first place? Forshaw and Hahn wrote in their paper [1] from 1990: "With the advent of more sophisticated ASIC libraries and CAD tools it becomes increasingly important for the circuit designer to adhere to a synchronous design methodology."

No doubt, the rising complexity of integrated circuits required methods for a faster and more reliable design process. In the synchronous design style, all flip-flops in the circuit are clocked with the same central clock, the period of which is carefully set to allow all signals traverse even the longest combinatorial path between two flip-flops (the critical path) and possible hazards to resolve. This allows to model temporal behaviour separately from functionality. The advantages of this approach are excellent tool support, efficient testability and simplicity of design. As those are legitimate arguments for using it, modern highly integrated chips have been designed almost exclusively using the synchronous methodology.

Disadvantages of synchronous design according to Forshaw and Hahn include higher power dissipation and the necessity of clock distribution. This were acceptable problems compared to the advantages they brought.

In the present, 20 years after the paper has been published, the on-chip transistor count has increased by three orders of magnitude. Keeping the synchronous timing assumption valid in such highly integrated systems is becoming a painful problem and is always connected with large performance, power, and area overheads.

As can be seen in [2], only approximately 45% of the clock period represent the actual computation. The rest includes safety buffers for correct operation under worst case conditions such as high temperature and low voltage, accounting for imperfect clock distribution

(clock skew — the difference in the arrival time of the clock edge at various locations of the chip) and combinatorial path imbalance (waiting for the critical path to complete even if it was not utilised in the current computation). With deep submicron technology scaling, the inherently present parameter variability increases and further degrades the worst case conditions for certain fault probabilities.

Distributing the high fan-out clock signal over the whole chip area requires a considerable length of wires which results in a high parasitic capacitance. Driving a high frequency signal over this network and still retaining sufficiently steep edges to lower clock skew requires strong drivers. In high performance microprocessors, clock distribution consumes up to 40% of the total power [3]. The high dissipation causes problems with power delivery as well as thermal design.

The area overhead caused by the clock distribution, although usually in the range of 25% can go up to 45% for certain applications [4].

Moreover, the clock wires also act like antennas and radiate the high-frequency clock signal. This electromagnetic radiation introduces noise which, in signal processing applications, can alias into the processed frequency band and degrade the quality of such applications.

Proponents of asynchronous design claim it is the solution for these problems and evidence their claims with manufactured asynchronous circuits with impressive results (see Section 1.3 for examples).

1.2 Asynchronous Circuits

Asynchronous circuits, also referred to as *clockless* or *self-timed*, are those in which no global synchronisation by means of a clock signal takes place. Sources of communication synchronise with the destinations locally for each transfer.

In this section, the timing models used in asynchronous circuit design, the different possibilities of implementation of asynchronous communication and typical components asynchronous circuits contain will be described.

1.2.1 Timing models

The assumption that all paths' delays globally share the same bound is abandoned. Instead, one of the timing assumptions supporting the asynchronous methodology is used by which the resulting circuit can be classified. The timing models frequently used in literature are:

Delay Insensitive Circuits that do not depend on any timing assumption for correct operation are called delay insensitive (DI). In this model, both gate and wire delays can be arbitrary (but finite). These circuits are insusceptible to parameter changes, be it operating conditions as temperature and voltage, migration to new process technology or rerouting. However, as Martin showed in [5], this class is very limited and virtually all practical circuits fall out of this class.

Quasi Delay Insensitive The least restrictive universally applicable model is called quasi delay insensitive (QDI). Its only assumption is that signals in certain wire forks arrive at their destination 'simultaneously'. More precisely, the difference in latencies of

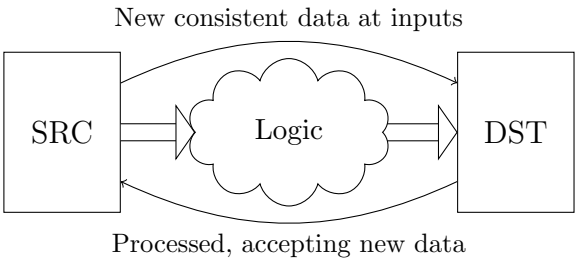


Figure 1.1: Handshaking signals

the fork branches must be smaller than the gate delay of the following logic. Such forks are called *isochronic*. Only a fraction of wire forks in a circuit have to be isochronic — no timing assumption is made about the other wires or the logic. The QDI class of circuits is Turing-complete, as has been shown in [6].

Speed Independent The timing model which assumes that interconnect delays are negligible compared to gate delays is called speed independent (SI). In contrast to QDI, in SI circuits all wire forks are isochronic by definition.

There are also variations of this class in which the circuit is partitioned into smaller regions. The rather unrealistic assumption of negligible interconnect delay within a circuit is relaxed to hold only inside those so called *equipotential regions*. This model is also referred to as *self timed* [7].

Bounded Delay Circuits that fall into the bounded delay (BD) class, also referred to as *matched delay* or simply *self-timed* require that both logic and interconnect delays are bounded much like in synchronous design. However, each logic path is considered individually whereas in synchronous design one bound has to apply system-wide.

1.2.2 Asynchronous communication

A communication link in asynchronous circuits is called a *channel* and is composed of two main parts, the wires carrying the actual data and means of synchronisation between the sender and the receiver. The synchronisation is called *handshaking* and is generally organised as an alternating sequence of a *request* for communication and the response to the request, called *acknowledgement*. One or both of those events can be coincident with data transfer.

Although a channel is a direct link, the same construct with combinatorial logic added to the data path is used to implement pipeline stages. A schema of a channel with optional logic is depicted in Figure 1.1.

The connection points for a channel, which are mostly located directly at the input to or the output from state holding elements, are also often called *ports*. This is the rule for external ports that are intended to connect a circuit to its environment.

Channels can also be categorised by the number of communicating parties. A *narrowcast* channel connects exactly one sender to one receiver. A *multicast* channel on the other hand connects one sender with multiple receivers. The handshaking in multicast channels must ensure that all receivers have processed previous data before the sender can transmit

a new value. Having multiple senders is not possible, since a simultaneous transmission to the same channel would lead to undefined results and may even cause damage of the output drivers.

Another parameter of a channel is the direction of the request relative to the direction of the data exchanged. This determines, whether the sender or the receiver initiate the communication. If the request comes from the sender, the channel is referred to as a *push* channel. In a *pull* channel, on the other hand, the receiver requests the transmission. If data are exchanged in both directions, coincident with both, the request and the acknowledgement, the channel is referred to as a *biput* channel. A port that is on the requesting side of a channel is called *active*, while a port waiting for requests to acknowledge is called *passive*. This attribute is important for interconnecting black-box circuits, since active ports can only be connected to passive ports and vice versa.

Channels can also be used purely for synchronisation. They consist only of handshaking signals and are referred to as *dataless* or as *synchronisation channels*. Controlling access to a shared resource might be an example of the application of such channels (See also Chapter 3).

Using the request on a channel as a trigger for various actions, mostly however communication on other channels, and delaying the acknowledgement of the request until all actions complete is called *handshake enclosure*. A typical example for such behaviour is a sequencing element, which upon a request on its input port sequentially performs full handshake cycles on its output ports and only after the last output completes its handshake, the element finishes by acknowledging its input request.

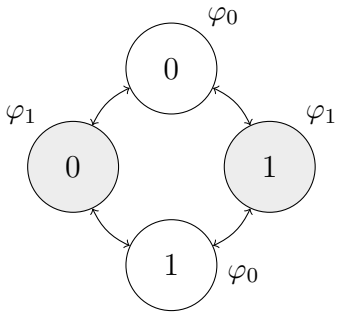
Signal coding

Since most of the timing models for asynchronous circuits allow arbitrary gate delay, it is not possible to determine when consistent data are present at the end of a logic path, without observing the data itself. Unfortunately, the usual data coding based on signal levels, where a positive voltage indicates a '1' and the grounding of a signal indicates a '0', cannot be used because of this fact. As an example, it is not possible to determine whether the current level on a wire is the new value to be received or whether transition is still to come without assuming completion and therefore bounding the time allowed for stabilisation of the signal.

A more complex signal coding is therefore required for most of the timing models described above, which not only allows to distinguish between a '1' and a '0', but also provides some separation of successive values. Since this implies the encoding of more than two states. This is only possible with more than one wire per bit. A coding which uses two wires per bit is called *dual rail*.

The act of observing data to determine when new consistent data is present is called *completion detection*. The circuit required for completion detection is dependent on the coding of the observed data. The most common codings are listed below:

Transition signalling Transition signalling (also referred to as *rail transition* coding) is the most simple dual rail coding for understanding; for hardware implementation, it is one of the most complicated. Each bit of data is encoded as a transition on one of two wires. One wire, called the '0-wire' for example, is used for logic zeros while the other, the



(a) State machine

	0	1
φ_0	00	11
φ_1	01	10

(b) Exemplary state assignment

Figure 1.2: The LEDR encoding scheme

'1-wire', is used for logic ones. Since each transmitted value corresponds to one transition, subsequent values can be accurately separated in time. For completion detection, the current states of the wires have to be compared to their states from when the last value input was processed. Thus, a state holding circuit is required for completion detection.

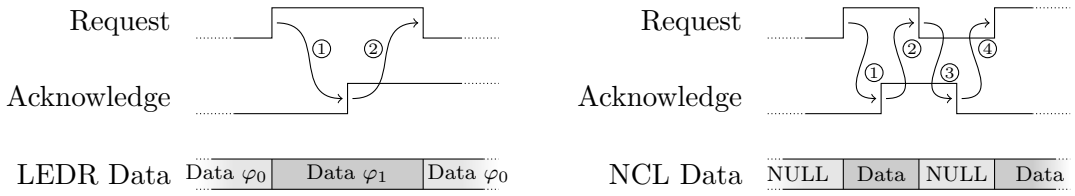
Null convention logic (NCL) In NCL, the two wires represent three different states. Apart from the two logic values to be encoded, a *NULL* is added. Usually, *NULL* is represented with both wires being at a low level and the logic values with exactly one of the wires being high. The fourth possible state (both wires being high) is not allowed.

For communication, alternating *phases* are used: A data phase where the data value is transfered followed by a *NULL* phase which serves as a separator between two subsequent values.

Level-encoded two phase dual rail scheme (LEDR) LEDR also uses phases to separate subsequent values like NCL, but uses all four possible states that can be encoded with two wires. Each of the two phases, either referred to as φ_0 and φ_1 or *odd* and *even*, can be represented by two of the four states available, one for each logic value. The state assignment is such that the change from one phase to another requires exactly one transition, no matter what values each phase encodes. Refer to Figure 1.2 for clarification.

N-of-m coding In n-of-m coding, m wires are used while transitions occur on exactly n of those wires for each data transfer. This gives $\binom{m}{n}$ possible combinations. Usually, the highest power of two that fits that number (be it 2^k) limits the amount of useful codes, as all possible values that can be encoded with k bits will be distributed over these codes.

As can be seen, n-of-m coding is not a dual rail scheme where each bit is separately encoded in two wires. Instead, the value is distributed over multiple lines without a direct correspondence between the value and its code. This makes this scheme more difficult to implement and unsuitable for computation as even simple operations require complicated combinatorial functions. With n-of-m coding, it is however possible to communicate data values with less transitions than the number of bits they contain and/or use less wires than the double of the amount of bits, which gives it a power and area advantage. It is mostly used for long interconnects or network-on-chip.



(a) Two phase handshaking

(b) Four phase handshaking

Figure 1.3: Handshaking protocols

A subset of n-of-m coding is the *one-hot coding*, where n is always equal 1. As an example, the most popular 1-of-4 coding uses 4 wires with exactly one having a transition at each transfer. This gives $\binom{4}{1} = 4$ possibilities and therefore 2 bits can be transferred at a time. This scheme uses the same amount of wires to encode an even number of bits as the above codes, however, in comparison to them needs only half the number of transitions to transfer a value.

An overview of different n-of-m codes as well as their implications on area and performance can be found in [8].

Handshake protocols

The simplest way to implement an asynchronous channel is to augment a standard data bus by two wires, a request and an acknowledgement wire. There are, however, two possibilities how to use those wires to signalise the handshaking events. Either a transition is used for each event, this corresponds to the *two phase handshaking protocol*, or one of the logic levels is used to signalise a pending transition, thus requiring two transitions for each event (one to bring the wire back to the idle state). The latter possibility is referred to as the *four phase handshaking protocol*.

For clarification, Figure 1.3 shows the two protocols and the causal relationships between events. In the figure, a push channel is assumed and the transitions on explicit request and acknowledge wires are shown in the top two lines of the figure.

Figure 1.3a shows the two phase protocol with the two causal relationships. First the sender generates a request by a transition on the request wire. After observing the request and processing the data, the receiver acknowledges the request by a transition on the acknowledge wire (1). Only after the sender receives the acknowledgement, it can output a new value (2).

In Figure 1.3b, the four phase protocol is depicted. Again, the sender starts with a request which is acknowledged by the receiver upon reception (1). However, when the acknowledge event reaches the sender, the request wire must be returned back to its idle state (2). After observing this, the receiver resets the acknowledge wire, too (3). Only after this cycle is complete, the next value can be transmitted by a new request (4).

The approach of using a standard, one wire per bit data bus augmented by handshaking wires is called *bundled data*. Clearly, for correct operation, the request signal is required to arrive at the receiver later than the data after traversing possible logic. This is implemented by estimating the worst case timing of the data path and delaying the request signal accordingly. Since the correct operation of the resulting circuit is depending on a timing

assumption bounding both gate and wire delays, only circuits of the bounded delay class can make use of such channels.

To design DI, QDI, or SI circuits, dual rail or more complicated signal codings supporting completion detection must be used. Since completion detection will indicate when consistent new data are present at the inputs of the destination, it can be used instead of the request signal. Note that while in push channels the data replace the request wire, in pull channels it is the acknowledge wire which is replaced by completion detection of the data. Biput channels use completion detection for all handshaking events. The codings can also be classified by equivalence to handshaking protocols. NCL has a default state (*NULL*) to which it has to return, which corresponds to the four phase handshaking protocol. With transition signalling and LEDR, each transition represents a new value, which corresponds to the two phase handshaking protocol. N-of-m codes can be used in both variants, with each n transitions denoting a new value (two phase) or with an idle state of the wires separating two values (four phase). In Figure 1.3, the equivalence of events on the request wire of a push channel and the completion detection of the data is shown using LEDR phases in the two phase handshaking protocol and NCL data validity in the four phase protocol.

Note that hybrid protocols are also possible where data use a four phase protocol while the acknowledge uses the two phase alternative or vice versa.

1.2.3 Metastability

In synchronous systems, all registers are controlled with the same clock signal resulting in virtually simultaneous registration of data. This common clock event allows to assume that in each cycle, data arrive at the registration points at the same time in the whole circuit. As there is no such synchrony in asynchronous systems, no two events can be considered to occur at the same time[1]. The behaviour of a circuit is thus not only dependent on communicated data values, but also on the order of event occurrences, which also includes whether an event has occurred between other events or not.

Where parts of a circuit are not explicitly synchronised using handshaking on dedicated channels, there is no causal or temporal relationship between events from those parts. Such events can occur in any order, but also as close to each other that they could be considered simultaneous, and they are called *concurrent*. When a circuit is fed with concurrent events and its behaviour is dependent on the order of their occurrences, the choice between two virtually simultaneous events cannot be made unambiguously and the circuit can become *metastable*.

Metastability is a state of a bistable device where the output is stuck between the two stable states. The circuit will eventually resolve the metastable state and the output will turn into one of the stable states. However, no upper bound can be given on the resolution time [9] which means, that if a circuit prone to metastability is given limited time to resolve before reading its output (such as in synchronous or bounded delay asynchronous circuits), there will always remain a certain probability that it will fail. The correct operation of asynchronous circuits that do not bound gate delay is not affected by the resolution time of metastability, however the undefined signal level still has to be avoided.

[1] This is true at a functional level. In the implementation, some signal arrivals can be assumed simultaneous (e.g. isochronic fork in QDI circuits).

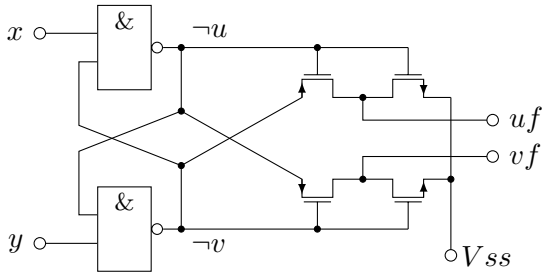


Figure 1.4: An implementation of an arbiter

In properly designed synchronous systems, the only source of metastability are asynchronous external signals and clock domain crossings that can violate the setup and hold times of the input flip flops. In asynchronous systems, in addition to the sampling of external signal levels, it is the ordering of any two concurrent events, such as the choice of requests when accessing a shared resource.

1.2.4 Typical components

There are a few building blocks, that solve some of the typical problems in asynchronous circuits, some of which will be described in this section. Where not stated otherwise, push channels are assumed in the descriptions. In the figures that show building blocks with exemplary input/output behaviour represented as tokens, the tokens are ordered by the time of occurrence with the earliest being at the right hand side.

The two probably most discussed functional blocks in literature are the arbiter and the synchroniser [10] which both implement functions that require the handling of potential metastability. The **arbiter** performs the task of choosing one of requests when these can occur simultaneously. When both requests are input at the same time, the arbiter makes a nondeterministic choice and delays the other request, until the chosen request is removed. An arbiter is called *fair*, if no request can be delayed infinitely. This means that a fair arbiter cannot choose one input over the other infinitely many times.

Figure 1.4 shows the implementation of a basic arbiter from [11], often referred to as a *mutual exclusion element* in literature. In the figure, x and y are the inputs, $\neg u$ and $\neg v$ are inverted, unstable, mutually exclusive outputs denoting which input has been selected and uf with vf are the stable outputs. The main functionality is implemented with the two logic gates. This part is called the *bare arbiter* (also, unstable arbiter) and produces inverted outputs susceptible to metastability. The four transistors form the *metastability filter*, an analog circuit preventing metastability to be driven at the outputs.

The **synchroniser** is used to sample a signal's value at a given point. It has two inputs, one accepting requests from the control circuit, the other for the sampled signal itself. In contrast to the arbiter, which requires requests not to be withdrawn until they are selected, the signal input of the synchroniser is allowed to change at any time. When the synchroniser receives a request, it outputs the current value of the signal and holds this value until the next request arrives. While this function can be implemented with a simple D flip-flop for signals being stable during the sample period, with signals that can potentially make a transition at a time very close to that of the request, the circuit becomes prone to metastability and a metastability filter as in Figure 1.4 has to be used.

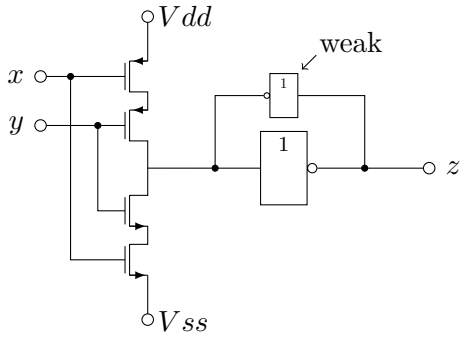


Figure 1.5: An implementation of a Muller C-element

Again, no upper bound on the decision time for ambiguous signal states can be given and such decision is nondeterministic.

The **Muller C-element** [12], or Muller C-gate, is another building block frequently used in asynchronous circuits. It performs the function of an AND gate for transitions. In Boolean logic, the AND gate indicates when both inputs are at a high level but gives no information about the states of the inputs when at least one is low. Similarly, an OR gate only indicates when both signals are low. To form the conjunction of transitions, each transition on the output must indicate transitions on both inputs. The Muller C-element is a state holding operator that serves this purpose. When both input levels are equal, the output has the same signal level. When the inputs differ, the output holds its state until both inputs are equal again. Figure 1.5 shows one possible implementation of a Muller C-element.

While the above blocks work directly with signals on wires, the following blocks, motivated by [13], use channels for communication and have a higher abstraction level, since they only correspond with a hardware implementation when the channels and the handshaking protocols used are defined.

A **fork** splits one channel into multiple branches. The forward path can be implemented by simply forking all channel wires without any additional components. However, in the opposite direction, the signals must be joined so that the input request is acknowledged only after acknowledges on all output channels are received. This can be implemented with Muller C-elements.

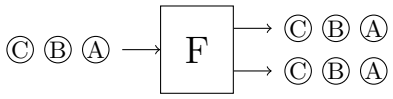


Figure 1.6: A fork block with an exemplary input/output

The **join** block has inverse functionality to that of a fork. When requests from all input channels are received, a request on the output channel is generated. Again, this functionality can be implemented with Muller C-elements. The reverse path, that of the acknowledgements, can be implemented by forking the acknowledge wire to all input channels.

A **merge** can be described as a self-selecting multiplexer. Requests from input channels are selected one-by-one and passed to the output channel. Once the handshake on the

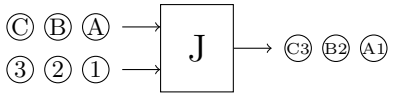


Figure 1.7: A join block with an exemplary input/output

output channel is completed, the incoming request is also acknowledged and the next request is selected. If requests on the inputs can occur simultaneously, the merge block must contain an arbiter to resolve possible conflicts. Where the input requests are guaranteed by the environment not to occur simultaneously, the merge block can be implemented with a simpler and faster circuit.

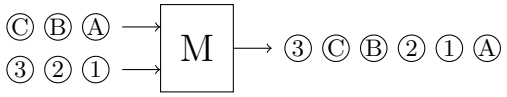


Figure 1.8: A merge block with an exemplary input/output

1.3 Properties of Asynchronous Circuits

In this section, the advantages of asynchronous circuits as well as the most notable disadvantages will be listed.

Average case performance The clock period of a synchronous circuit has to be carefully set to make sure the longest register-to-register path in the whole clock domain has enough time to produce valid and stable outputs before the next clocking edge even when it is not used in the current computation. Therefore also rarely used blocks have to be optimised not to slow down the whole design and optimising other, frequently used blocks to be faster than the clock period does not bring any benefit.

An asynchronous circuit on the other hand waits for the data to become valid before registering them, instead of waiting for a global clock signal. This is either done by dual rail data encoding or a delayed request signal. In both alternatives, the actual register-to-register delay will depend on the current computation. Therefore when computing on short paths with a few logic gates only, the circuit is much faster than when computing on long paths with many gates. As a result, rarely used functions can be left unoptimised and still have only marginal effects on the overall performance while even slight optimisations on frequently used paths may have considerable effects, corresponding to Amdahl's law [14].

Of course, in a pipeline the throughput of consecutive stages is linked and if one stage has a constantly limited throughput, the design will not benefit from optimising the following stages to be faster. However, if one stage is very rarely performing a slow computation, this will only delay the pipeline for this occurrence and not for the whole computation and may be left unoptimised.

Lower power In standard synchronous circuits, the global clock signal is driving all state holding elements with a minimal skew which requires strong drivers and a large

distribution tree. The long wires along with the huge fan-out increase the power required for every transition. The clock, being global has to switch even when only a little part of the circuit requires it, e.g. when most of the pipeline is stalled because of a branch misprediction.

A typical technique to address this issue in synchronous design is clock gating. The circuit is divided into clock islands in which the clock signal can be turned off. This way, unused parts of the design do not consume dynamic power. However, the partition of the circuit can be done only in a coarse-grain manner and unnecessary switching can only be reduced, not avoided.

In asynchronous circuits, there is no clock signal, and switching only occurs in parts of the circuit currently performing a computation. All other parts are idle and consume only leakage power.

However, data encoding for DI, QDI and SI circuits can introduce some additional switching as consecutive values have to be distinguishable by transitions on the wires. As an example, NCL coding requires two transitions for every transferred bit, LEDR and transition signalling require one and in 1-of-4 coding, two bits will be transferred with every transition. Nevertheless, the dynamic power saving in circuits using such codes compared to their synchronous counterparts is still considerable.

Adaptation to physical variations The problems with variations as described in the beginning of this chapter, can be solved by using DI, QDI or SI circuits. Those are inherently free from predefined timing assumptions and operate as fast as the current environment allows. When a transient delay fault slows down the computation, the circuit will simply wait for it to finish. No statistical analysis is necessary to adjust the clock period to tolerate faults with a certain failure rate.

This also applies for operating temperature and the power supply voltage. In synchronous design, the operating conditions such as temperature range as well as the power supply voltage range are specified, and the clock period is set for the circuit to work under these conditions. However, the boundaries of these safe operation intervals are worst case conditions and most of the time (in an average environment) the circuit could safely perform better. A DI, QDI or SI asynchronous circuit adjusts to the environment and simply becomes slower with rising temperature.

Technology migration and modularity As a consequence of the above paragraph, migration of an existing asynchronous design to a new technology (e.g. from 90 to 45 nm) does not require any changes to the design (assuming delay insensitive design or similar). This also greatly simplifies modular design — to use a module a simple interconnection of data and handshaking signals is sufficient no matter which technology the module was designed for and which is actually used. No timing requirements have to be fulfilled to ensure correct operation.

Lower electromagnetic emission As the clocking edge in synchronous circuits is distributed over a large wire network and triggers a considerable amount of switching activity, the electromagnetic emissions of such circuits are concentrated around the clock frequency and its harmonics and, as a result, have high peaks at those frequencies.

The emission spectrum of asynchronous circuits is more evenly distributed over the frequency domain with much lower peaks as synchronisation occurs only between communication partners and not globally. This has advantages for example in devices with radio receivers which receive less disturbance from the digital signal processing unit and so achieve a better signal to noise ratio, possibly eliminating the need to switch off the processor during the reception of a message.

Failure-free arbitration The timing assumption in the synchronous approach, where each signal is allowed to change at a certain time window can only apply within a circuit. However, most circuits have to process external events such as interrupts which are inherently asynchronous. Such signals have to be synchronised before use to avoid timing faults such as metastability. Also, in clock domain crossings, signals have to be synchronised to the local clock.

The synchronisation process, for example using a synchroniser as described in the previous section, can take an arbitrary time to resolve. In synchronous circuits, the maximum resolution time is limited to allow correct operation with a certain probability. However, the longer the resolution time, the longer the delay introduced by the synchronisation process. The reliability of the circuit is therefore traded for its performance. Asynchronous circuits allow failure free synchronisation as no time bounds are applied (delay insensitivity or similar assumed), while not introducing additional delay to the average case.

There are three most notable disadvantages directly resulting from the requirements of asynchronous circuits.

Larger area Most asynchronous timing models require dual rail or other advanced signal coding schemes that allow the separation of successive values. This is usually associated with a 100% wiring overhead which undoubtedly increases the chip area occupied by the circuit.

No masking of hazards A considerable advantage of timing assumptions in synchronous and bounded delay circuits is the fact that all activity of combinatorial logic is ignored in the time between two clock edges. This allows all hazards to be resolved before a value is read. In DI, QDI and SI circuits, where each transition represents an event, hazards must be avoided with techniques such as masking with additional gates and small scale relative timing constraints similar to that of an isochronic fork.

Lack of tools In synchronous design, a vast majority of circuits are described in either VHDL or Verilog. These are supported by a wide range of high quality tools capable of code generation, synthesising to netlists for a multitude of back end technologies, simulation and much more. The tool sets available for asynchronous design, with a few exceptions, originate from academia and use own description methods as input. A commercially wide accepted design flow with a choice of high quality tools that can interchangeably be used for various steps is, however, yet missing.

Testability In the testing process of synchronous circuits, the global clock can be halted which causes the state of the whole circuit to be locked in state holding elements. By

added shift registers, the state can be read and/or altered to perform predefined tests. The testing of asynchronous circuits is more difficult because they cannot be simply globally halted, redundant logic used for masking hazards can also mask faults, and some faults, usually timing related, only become observable in certain operating conditions.

Design complexity The design of asynchronous circuits is arguably more difficult than that of synchronous ones. This is due to the need of local synchronisation in every path, analysis of the design to show the absence of deadlocks, the elimination of hazards in logic paths and the identification and meeting of (relative) timing constraints at the lowest design level. However, the complex design of a low skew, multi gigahertz clock distribution network for large chip areas in current technology, which is not needed in asynchronous circuits, mitigates the advantages of synchronous design.

Selected references for implementations of asynchronous circuits include the "Caltech asynchronous microprocessor" [15, 16], the series of "AMULET" processors [17, 18, 19], the "MiniMIPS" processor [20], the asynchronous 80C51 microcontroller [21], the ultra-low power "Lutonium" processor [22] and an asynchronous FPGA architecture [23, 24].

Chapter 2

Description Methods

In the beginning of this chapter, the typical problems in asynchronous design are extracted from the previous chapter, also serving as criteria for the later comparison. The following parts introduce the most popular description methods for asynchronous design, explain them and list some of their properties. Though the descriptions of the methods in this chapter do not fully show all their properties and capabilities, they are meant to be sufficient to give a good understanding of the methods and their underlying concepts.

2.1 Problems in asynchronous circuit design

The following list shows the typical problems in asynchronous circuit design extracted from Chapter 1.

Timing Even though the choice of timing models is more related to the circuit synthesis, the timing assumptions inherent to a description method are also very important to evaluate. Moreover, only methods that allow the specification of time bounds can be used to describe circuits in the bounded delay class.

Communication The communication in asynchronous circuits has many variants. Parameters such as the employed handshaking protocol, signal coding, the activeness and passiveness of ports must all match for successful communication between two circuits. The possibilities to chose these parameters in the description methods will be evaluated.

Arbitration One of the most important functional block in asynchronous circuits is the arbiter. While it prevents failure due to metastability, its unnecessary use where requests are known not to occur simultaneously decreases the performance and increases the area requirements of a circuit. Description methods thus should allow the description of mutual exclusion with or without the use of an arbiter. This will be evaluated using a merge block with optional arbitration. Also note that the metastability filter as part of the basic arbiter is an analog circuit that cannot be described using other building blocks such as the bare arbiter.

Concurrency Since an asynchronous circuit is a highly concurrent system with explicit synchronisation required for every communication, it is important to be able to describe it as such. Description methods should allow the description of concurrent functional block and the synchronisation between them.

Sequence Every description also contains events that are required to be sequential. For this purpose, a description method should provide a convenient way to define causal order between events.

Level/Event sensitivity A description method is said to be level sensitive, when the behaviour of circuits is described using signal levels. An event sensitive description method on the other hand, uses events which can range from transitions on wires to complete synchronised communications for the description of the circuit's behaviour.

Level of abstraction Similarly to synchronous circuits, a higher level of abstraction simplifies the design of complex circuits but hides some implementation details that might be important for simpler circuits. Each description method will be analysed for the abstractions it uses and the effects they have on the expressiveness of the method.

Modularity and parametrisation A key concept allowing feasible descriptions of complex circuits is modularity, which allows commonly used circuit blocks to be described as modules and then instantiated wherever they should be used. A further improvement is the possibility to describe parametrised modules which can be assigned parameters at the time of instantiation, allowing one module to describe several similar circuits thus improving it's applicability. It will be evaluated whether the description methods allow previously described blocks to be reused within another description.

2.2 Production Rule Sets

As the first and at the same time the least abstract description method, the *Production Rule Sets* (PRS), originating from the *Caltech asynchronous synthesis tools* (CAST) [25] as an intermediate representation, will be described. The description is a summary of [11] and [26].

Production rules are based on Dijkstra's *guarded commands* [27] and have a very similar notation and semantics. A *production rule* (PR) is a construct of the form $G \mapsto S$ where G is a Boolean expression called the *guard* of the production rule and S is a single or a comma separated list of assignments. The assignments can only be as simple as setting variables to a high or low logic level, in the sequel only referred to as *true* and *false*. The notation of an assignment which sets variable x to *true* is $x\uparrow$, for setting x to *false* it is $x\downarrow$. The guard can be any Boolean expression with one or more variables, however, some limitations to allow direct mapping to CMOS circuits and their correct operation apply.

The guards are restricted to be *stable*, which means that if the evaluation of the guard changes from *false* to *true*, it must remain at this value until the execution of the assignment. Two PRs that assign different values to the same variable are called

complementary and implement an *operator*. It is required, that for the guards G_1, G_2 of complementary PRs

$$G_1 \mapsto x\uparrow$$
$$G_2 \mapsto x\downarrow$$

$\neg G_1 \vee \neg G_2$ always holds. In other words, it is not allowed for two PRs to simultaneously assign different values to the same variable. This property is called *non-interference*. Additionally, if $G_1 \vee G_2$ holds at all times, the variable x is always held at a logic level and the implemented operator is called *combinational*. If, however, $\neg G_1 \wedge \neg G_2$ can hold at any time, x will by definition hold its last value in that time and the resulting operator is called *state holding*.

The concurrent composition of multiple production rules is called a *production rule set* (PRS). The behaviour of circuits described as blocks of interconnected PRSs is equivalent to the set union of all the PRs they contain with PRs specifying the connecting wires, in the form of identity functions, added to the set. In case two different PRs $G_1 \mapsto S$ and $G_2 \mapsto S$ have the same assignment at their right hand side, they are replaced by the new rule $G_1 \vee G_2 \mapsto S$.

The correspondence of operators described in PRs and the mapping to their CMOS implementations is straight-forward, possibly even direct. In a typical CMOS circuit, where *true* is represented by a positive voltage while *false* is represented by pulling a wire to ground, an operator is implemented by a P-stack of transistors pulling the output high and an N-stack of transistors pulling it low. The operation of each stack is controlled by a Boolean function of the inputs. This is equivalent to a simple assignment in PRs as the controlling functions for the P-stacks and the N-stacks are equivalent to guards of PRs that set a variable to *true* or *false*, respectively. See Figure 2.1 for clarification.

To allow a direct mapping of operators to their implementations, there is a restriction on the Boolean functions in the guards. For a negative assignment ($x\downarrow$), the guard can only contain non-negated input variables. This allows to connect inputs directly to the gates of n-channel FETs. For a positive assignment ($x\uparrow$), only negated forms of input variables can be used to allow the connection to p-channel FETs. Note that this limitation is reversed for some implementations of state-holding operators, where an additional inverter is placed at the output (e.g. the static implementation as shown below). The requirement of non-interference for complementary PRs prevents short circuit paths between the positive voltage supply and ground to be opened in operators by evaluating both guards to true. However, non-interference does not affect the presence of intermittent short circuits as a result of dynamic circuit behaviour.

In the implementation, also the difference between combinational and state holding operators becomes clear. Combinational operators force their output to the supply voltage or ground at all times. The output always has a 'strong' level. The state-holding operator, however, has states where neither the P-stack nor the N-stack are driving the output, which remains floating. The implementation has to ensure that its level does not change until it is driven again. In a *dynamic* implementation, this is done by timing assumptions ensuring that the output value will be used (or driven again) before it loses its charge and changes to an undefined level. An additional capacitor can be used to prolong this time. In a *static* implementation, a storage element such as that in Figure 1.5 is added to the

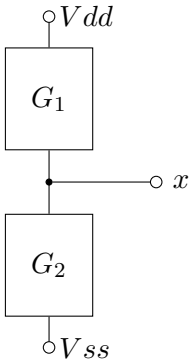


Figure 2.1: The CMOS implementation of a combinational operator with x being the variable in the assignment [26].

circuit to drive the output with the value assigned last.

As mentioned before, PRSs are the least abstract description method in this work, working with single operators, possibly equivalent to a transistor netlist. Handshaking protocols thus have to be implemented explicitly, every transition needs to be modelled.

As a simple example, the following PRS describes a fork block (see Section 1.2.4) with dataless push channels using RTZ four phase handshaking with all wires initially low. In the description, a is the input channel, x and y are the output channels and the *ack* and *req* indexes refer to the acknowledge and request wires the channels are composed of. The optional reset circuitry is omitted.

$$\begin{aligned} a_{req} &\mapsto x_{req}\uparrow, y_{req}\uparrow \\ \neg a_{req} &\mapsto x_{req}\downarrow, y_{req}\downarrow \\ x_{ack} \wedge y_{ack} &\mapsto a_{ack}\uparrow \\ \neg x_{ack} \wedge \neg y_{ack} &\mapsto a_{ack}\downarrow \end{aligned}$$

As can be seen, the first two PRs implement a combinational operator with the identity function. This can be implemented with a simple wire fork with no transistors needed. The other two PRs implement a state-holding operator — a Muller-C element. Non-interference is trivially fulfilled. Stability has to be guaranteed by the environment. In this case, adherence to the four phase protocol is enough to ensure stability of the guards.

As mentioned at the beginning of this section, PRS are used as an intermediate representation in the CAST toolchain [25]. PRS are usually automatically generated using decompositions and transformations from a higher level description (see Section 2.6.3) but they can also be used as the design entry description method. Since PRS can already be straightforwardly mapped to CMOS operators, the further processing of PRS in the CAST toolchain is already part of the physical design. As a first step, the description is transformed into *extended production rule sets* (XPRS), where transistor sizing and gate ordering are added. The final step is the chip layout design using a place and route tool, which is also part of the CAST toolchain.

2.3 Asynchronous Finite State Machines

A *finite state machine* (FSM) is a graphical model of an abstract computing device used to describe the behaviour of programs and digital circuits. It consists of a finite number of states, a function describing the transitions between the states and a function generating the outputs. An equivalent model used in formal computer science or mathematics is called a *finite transducer* or, if the only output is a Boolean variable (often used as indication of the acceptance of an input sequence), a *finite automaton* and both are backed up by rigorous theory. The two most common FSM models were introduced by Mealy [28] and Moore [29] and at present, these models are described in virtually every coursebook for digital design.

Formally, a finite state machine is a sextuplet $FSM = (\Sigma, \Gamma, S, s_0, \delta, \omega)$, where:

- Σ is the input alphabet
- Γ is the output alphabet
- S is the finite non-empty set of states
- s_0 is the initial state
- δ is the state transition function
- ω is the output function

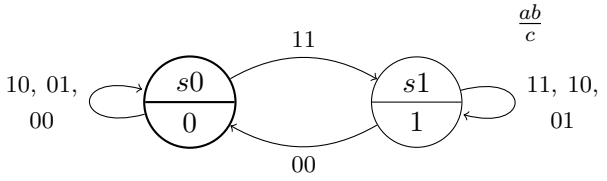
At the beginning of a computation, each FSM is in its initial state s_0. The actual operation of an FSM consists of a possibly infinite sequence of atomic computing steps, each theoretically performed in zero time and incorporating these actions:

- *Read the input.* Formally, an FSM reads symbols, which are elements of Σ, from the input. For hardware description, the symbols represent states of all input wires.
- *Perform a state transition* according to the state transition function $\delta : S \times \Sigma \mapsto S$.
- *Generate output* according to the output function ω. Formally, an output symbol from Γ is written to an output tape. The symbol can represent states of all output wires for hardware description.

There are two common alternatives for the output function ω for FSMs. For a Moore FSM, the output function is $\omega : S \mapsto \Gamma$. That means that the output is only a function of the current state. In a Mealy FSM, the output function is defined as $\omega : S \times \Sigma \mapsto \Gamma$, therefore making the output dependent on the current state as well as the current input[1]. Although the Mealy FSM model seems to be mightier than the Moore FSM model, they are actually equivalent.

FSMs are most commonly represented in two ways, either graphically or with a table. The graphical representation is a labelled directed graph in which the vertexes represent the states, the initial state being drawn thickly, while the edges labelled with input values (symbols) represent the state transition function. The description of the outputs differs

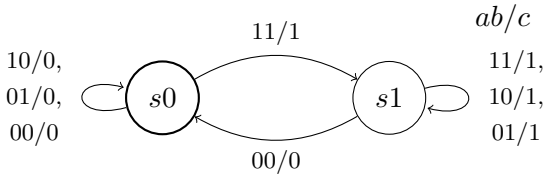
[1]The inputs are only read at the beginning of a computation step and therefore the current input is always the input that was read before the state transition.



(a) Moore type, graphical representation

state/output (c) \ input (ab)	00	01	10	11
next state				
$s0/0$	$s0$	$s0$	$s0$	$s1$
$s1/1$	$s0$	$s1$	$s1$	$s1$

(b) Moore type, state transition table



(c) Mealy type, graphical representation

state \ input (ab)	00	01	10	11
next state/output (c)				
$s0$	$s0/0$	$s0/0$	$s0/0$	$s1/1$
$s1$	$s0/0$	$s1/1$	$s1/1$	$s1/1$

(d) Mealy type, state transition table

Figure 2.2: An FSM describing a Muller C-element.

between the Moore and Mealy FSM. In a Moore FSM, the output values for each state are defined in the labelling of the vertexes. In a Mealy FSM, where more output values are possible for the same state depending on the input, the edges are labelled with the output that should be generated after the corresponding state transition. An example of the graphical representation of an FSM describing a Muller C-element with the inputs a, b and an output c can be seen in Figure 2.2a as a Moore FSM and Figure 2.2c as a Mealy FSM.

The tabular representation, called *state transition table*, is a table in which each row represents one state and each column one input value. For each state, the cells record the next state the FSM should change to after receiving the input corresponding to the column. The outputs to be generated, for a Moore FSM, are specified together with the states, one for each row. A Mealy FSM requires the outputs to be specified in the cells, one for each row/column combination. As an example, the same state machine like above, described with a state transition table, can be seen in Figure 2.2b for the Moore FSM and Figure 2.2d for the Mealy FSM.

Finite state machines are very efficient in describing circuit behaviour because they model functionality without considering the temporal behaviour of the implementation. For synchronous circuits, this is appropriate — the typical implementation of a FSM consists of a register holding the current state carefully clocked such that all combinational paths have finished evaluation and all signals are stable. The clock edge then initiates a new computing cycle by updating the state in the register which causes the next state to be generated.

In asynchronous circuits, however, time cannot be so easily abstracted away. For example, it is not possible to input zeros on a multiple wire bus as one symbol and ones on the same wires as the following symbol without observing the intermittent state of mixed zeros and ones on the wires as they are switching. Without timing assumptions or limiting the state transition function to use only inputs that can be recognised from previous ones, it is not possible to know when the switching is finished and the values on the wires represent a consistent input. Additionally, even when all accepted inputs are such that no

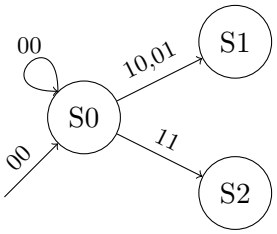


Figure 2.3: Example of a FSM fraction that cannot be implemented with a SIC fundamental mode AFSM

intermittent value represents a valid input which might cause a wrong transition, the state transition function must be designed to produce hazard-free outputs even with multiple inputs changing.

2.3.1 Fundamental Mode Asynchronous Finite State Machines

The approach for the design of asynchronous circuits initiated by Huffman in [30] uses timing assumptions for both the circuit and the environment.

The environment that Huffman circuits operate in must adhere to the restriction that after a change on one input wire, the circuit is given enough time to stabilise before another input is allowed to change. An asynchronous FSM (AFSM) operating in such an environment is said to operate in *single input change* (SIC) *fundamental mode*. The stabilisation of the circuit is effectively the evaluation of the next-state function resulting in a state transition which, depending on the state machine's behaviour, may trigger another state change for the same inputs. Only after the FSM reaches a stable state, that is, a state in which for a given input the state transition function assigns the same state to be the next, the inputs may change again. The minimum time between two input changes is thus limited by the speed of the FSM and as a logical consequence, the gate and wire delay within the circuit have to be bounded to be able to determine this time.

A further requirement is that also the signals internally generated as the evaluation of the next-state function must adhere to the SIC fundamental mode. This is a consequence of the missing temporal masking of hazards as in synchronous FSMs. Practically, this means that additional delays have to be inserted in the feedback path of the state signals.

The SIC fundamental mode operation, however, also imposes limitations on the expressiveness of a state machine. As an example, consider the behaviour modelled by the FSM fragment in Figure 2.3. State S0 is a stable state with the input values being '00' (only incoming transition). If the designer wished to change to one state (S1) if a single input becomes high and to another (S2) if both become high, such an FSM would work without problems in the synchronous implementation. The SIC restriction, however, makes state S2 unreachable, as always one of the inputs have to change first and the circuit has to be given time to perform a state transition, which in this case would change the FSM to the state S1. This also applies to state signals, where a direct transition from a state encoded with '00' to a state encoded with '11' is not possible, if the intermittent signal values '01' or '10' could lead to different stable states.

2.3.2 Burst-Mode Asynchronous Finite State Machines

To weaken the strong assumptions about the environment in SIC fundamental mode AFSMs, *burst-mode* AFSMs were introduced.

Burst-mode AFSMs also work in fundamental mode, but restricted *multiple input change* (MIC) is allowed — predefined sets of transitions, so called *bursts*, are allowed to occur concurrently without any timing assumptions placed upon them. Even though AFSMs allowing restricted MIC could also be described using one of the formalisms described above, the term "burst-mode AFSM" also refers to a new state machine formalism. In this work, every reference to burst-mode AFSMs refers to this formalism.

The formal definition of a burst-mode FSM, taken from [31], reveals the slight differences to standard FSMs as described above, especially in the description of the state transitions. A burst-mode FSM is a labelled directed graph, $BM = (V, E, I, O, v_0, in, out)$, where:

- V is a finite set of vertices (describing the states)
- E is the set of edges (describing the transitions)
- I is the set of inputs
- O is the set of outputs
- v_0 is the initial state
- in is a labelling function describing values of inputs in each state
- out is a labelling function describing values of outputs in each state

The edges $E \subseteq V \times V$ form a relation describing which transitions are possible from each state. I and O are simply sets of input or output signals respectively, e.g. $I = \{a, b\}$ and $O = \{x\}$ for a circuit with inputs a, b and an output x. At the beginning of its operation, a machine is in the initial state, v_0. The functions $in : V \to \{0,1\}^{|I|}$ and $out : V \to \{0,1\}^{|O|}$ assign values of the input and output signals respectively to each state. While out is equivalent to the output function ω in classical Moore-type FSMs (see the introduction of Section 2.3), the function in, rather than specifying which state the machine should change to with given inputs, as the state transition function δ does, assigns the values of all inputs to each state, called its *unique entry point*. This implies, that no matter from where the machine changes to a state v, the input values have to be exactly $in(v)$. This does not restrict the expressiveness of the notation, since specifications that do not meet this requirement can be transformed to valid ones by splitting states.

Although the behaviour of a burst-mode AFSM is described with the set of states, edges between them and functions labelling the states, in the graphical representation the more practical labelling functions for the edges are used. The functions are called $trans_i : E \to \mathcal{P}(I)$ and $trans_o : E \to \mathcal{P}(O)$ and they label each edge with a subset of input and output signals respectively that change their value between their source and destination state. In the label, $trans_i$ is separated from $trans_o$ with a slash and each signal name is augmented with a '$-$' or '$+$' depending on whether the signal changes from *true* to *false* or vice versa.

Apart from the requirement for each state to have a unique entry point, there is another restriction for valid burst-mode AFSM: Given some state, no possible input burst can be a subset of another one originating from the same state (formally, $\forall v, u_1, u_2 \in V, \quad trans_i(v, u_1) \subseteq trans_i(v, u_2) \Rightarrow u_1 = u_2$). This is called the *maximal set* property and eliminates the possibility of ambiguous behaviour of state machines when, after a valid input burst, it could either perform a state transition or wait for other inputs to change to complete another valid input burst.

An example of a tool for circuit synthesis from an AFSM representation is *3D*, which was developed initially at Stanford University and later at the University of California, San Diego, from where it is currently available. It uses *extended burst mode* (XBM) AFSMs, introduced in [32] for circuit descriptions. Those add the possibility of sampling level sensitive signals as conditions for state transitions and *directed don't cares*, which allow to expand the path between a rising and a falling transition of a signal over multiple states by describing its value to be undefined in intermittent states. 3D uses the synthesis method described in [33, 34]. Another example is the toolkit *MINIMALIST* [35], developed at Columbia University, which is composed of different tools performing tasks such as state minimisation, state encoding and logic minimisation.

2.4 Signal Transition Graphs

The first part of this section will describe *Petri nets*, a powerful graphical description method commonly used for the modelling of concurrent systems. *Signal transition graphs* (STGs) are the most common description method for asynchronous circuits using Petri nets and will be described in the second part of this section.

2.4.1 Petri Nets

Petri nets (PN) are another graphical representation of an abstract machine performing a computation, which can be used to describe asynchronous circuits. They were introduced by C. A. Petri in [36]. An excellent source for information on Petri nets is [37].

Petri nets can be seen as a generalisation of finite state machines — each FSM can be directly transformed into an equivalent PN (see page 25). The advantages of PNs, compared to FSMs, are the ability to model concurrent, asynchronous, distributed systems and non-determinism, which allows the application of PN theory to prove properties like the absence of deadlocks.

Formally, a Petri net is a sextuple $PN = (P, T, F, W, K, M_0)$, forming a weighted directed bipartite graph consisting of interconnected *places* and *transitions*.

- P is the finite set of places
- T is the finite set of transitions
- F is the flow relation
- W is the weight function
- K is the capacity function

- M_0 is the initial marking

In the graphical representation, places are drawn as circles, transitions as possibly filled rectangles. The flow relation $F \subseteq (P \times T) \cup (T \times P)$ defines the arcs of the graph. Note that since Petri net graphs are bipartite, only interconnections of places with transitions are possible. The directed arcs define the predecessor/successor relations — all places connected to a transition with incoming arcs are called *predecessor* places and ones connected with an outgoing arc are called *successor* places. The same applies with places and transitions reversed.

The state of a PN is given by its *marking* $M : P \to \mathbb{N}_0$, that assigns a non-negative amount of tokens, depicted as filled circles inside places in the graphical representation, to each place. The initial marking M_0 describes the marking of a PN at the beginning of a computation. The weight function $W : F \to \mathbb{N}$ assigns each arc a weight, equivalent to its multiplicity. The capacity function $K : P \to \mathbb{N}$ assigns each place the maximal amount of tokens it can hold. A Petri net in which the capacity function is not defined, therefore all places have unbounded capacity, is called an *infinite capacity net*. When K bounds the capacity of places, the resulting net is referred to as a *finite capacity net*. Each finite capacity net can be transformed to an equivalent infinite capacity net.

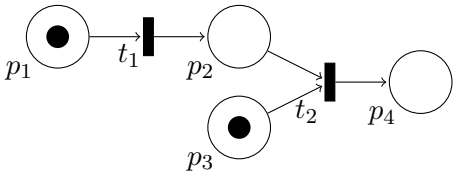
Petri nets in which places are allowed to hold multiple tokens are useful for describing processes in a higher level. For example, one place could describe a FIFO buffer and the transitions connected to it would represent events adding and removing objects from that buffer. When describing circuits, however, PN transitions represent transitions of input or output signals and whether a place has a token or not is usually implemented as the logic value of a wire. To keep a close relationship between a PN and the resulting circuit, *1-bounded* Petri nets are used for the description of circuits. A Petri net is 1-bounded when every place has the capacity of maximally one token (formally, $\forall p \in P, \quad K(p) = 1$). As each place in a 1-bounded PN can only have one or no tokens, using edges with weight more than one would permanently disallow the firing of the transition being either the source or the destination of such an edge. A PN, in which every edge has weight one is called *ordinary*. In the following, 1-bounded ordinary Petri nets are assumed. Note that such PNs can be specified without the weight and capacity function.

The semantics of Petri nets can be described with just a few rules:

- Transitions in which all predecessor places have tokens are *enabled*.
- All enabled transitions must eventually *fire*.
- When a transition fires, tokens are removed from all predecessor places and added to all successor places.

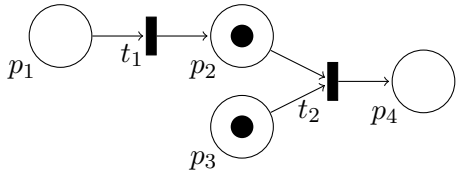
For a better understanding, Figure 2.4 shows a Petri net in three different computation steps. In the initial marking (Figure 2.4a), only transition t_1 has tokens on all predecessor places and therefore is enabled. After its firing, the token from p_1 is removed and added to p_2 (Figure 2.4b), which disables t_1 and enables t_2, now being the only allowed event in the modelled system. After t_2 has fired, tokens from both p_2 and p_3 are removed and a token is placed on p_4 (Figure 2.4c). This is the final state of the PN as no further transitions are enabled. Note that this is the only possible execution of this PN.

Figure 2.5 shows Petri net structures that model important types of behaviour often occurring in asynchronous circuits. In Figure 2.5a, the two transitions are concurrent —

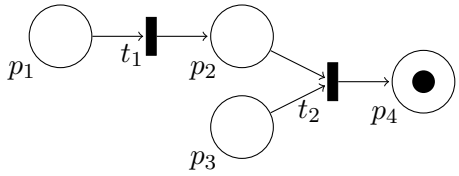


v

(a) Initial marking



(b) Marking after the firing of transition t_1



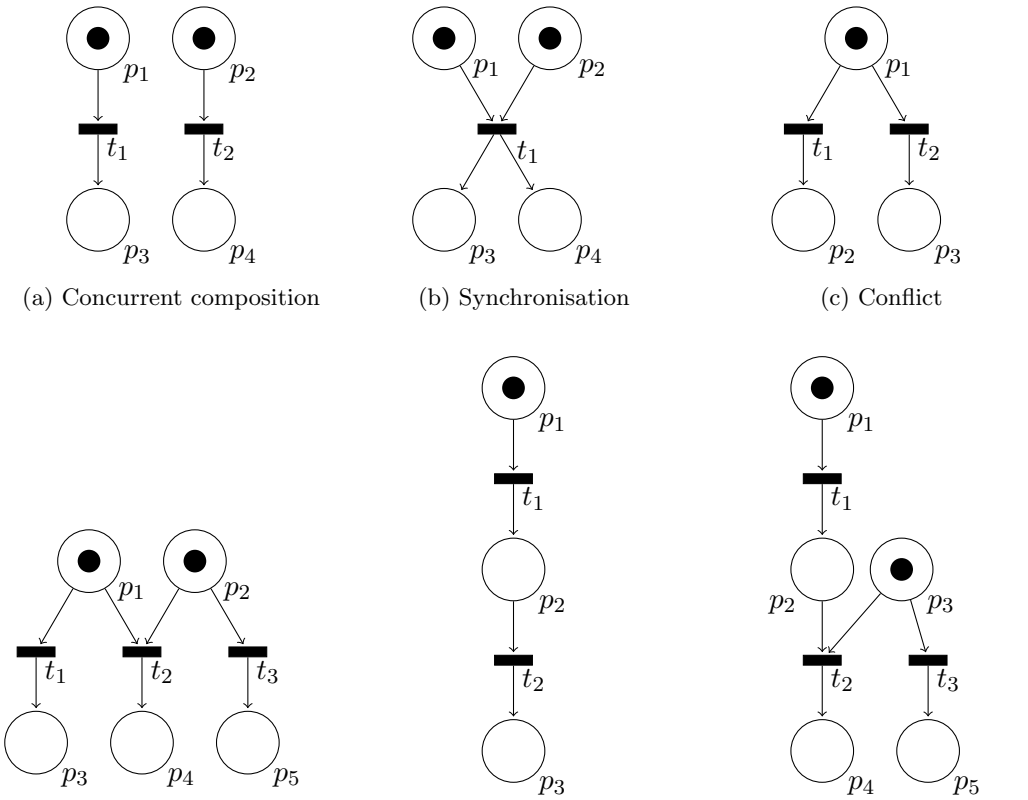
(c) Marking after the firing of both transitions

Figure 2.4: An exemplary Petri net in three computation steps.

each can fire independently of the other and no constraints on their temporal or causal relationship are given. The structure in Figure 2.5b models synchronisation — t_1 can only fire when both p_1 and p_2 have tokens. The behaviour modelled by the structure in Figure 2.5c is called *conflict*. Here, the firing of the two transitions is mutually exclusive — either t_1 or t_2 fires. This is by definition a non-deterministic decision without restricting properties like fairness. In Figure 2.5d, concurrency and conflict are combined in a structure called *symmetric confusion*. Transitions t_1 and t_3 are concurrent, while both being in conflict with t_2 — either t_2 fires and disables the other transitions, or they fire concurrently but one of them earlier than t_2. Sequential composition is depicted in Figure 2.5e. Here, t_2 can only fire after the firing of t_1. The combination of all three, conflict and concurrent and sequential composition is shown in Figure 2.5f. This behaviour is called *asymmetric confusion*. Transitions t_1 and t_2 are sequential, t_1 and t_3 are concurrent and t_2 is in conflict with t_3. Note, however, that the conflict only occurs when t_1 fires first.

For simplifying the analysis of different properties of computation models, there are several subclasses of Petri nets, restricting their structures. Those restrictions allow the use of more efficient algorithms but also reduce the expressiveness of Petri nets.

- *State machines* (SMs) are such ordinary PNs, in which each transition has exactly one predecessor and one successor place i. e. they allow conflict but not synchronisation. Furthermore, the number of tokens in the net does not change throughout the computation.
- *Marked graphs* (MGs) are such ordinary PNs, in which each place has exactly one predecessor and one successor transition i. e. they allow synchronisation, but not



(a) Concurrent composition (b) Synchronisation (c) Conflict

(d) Symmetric confusion (e) Sequential composition (f) Asymmetric confusion

Figure 2.5: Typical Petri net structures

conflict.

- *Free-choice nets* (FCs) are ordinary PNs, which generalise SMs and MGs. Both conflict and synchronisation are allowed, but not at the same time. This means, that each arc going from a place is either the only output arc of that place (no conflict) or the only input arc to the target transition (no synchronisation).
- *Extended free-choice nets* (EFCs) are ordinary PNs, which allow conflict and synchronisation at the same time, however they do not allow confusion, i.e. if two places share a successor transition, then all of those places' successor transitions must be shared by both of them.
- *Asymmetric choice nets* (ACs) are ordinary PNs, which add asymmetric confusion to the modelling power of EFCs, i.e. for each two places sharing a successor transition, each successor transition of one of the places must be shared amongst them.

Note that if there is a conflict in SMs, FCs and EFCs, the transitions in conflict are either all enabled or disabled. ACs allow the disabling of a subset of conflicting transitions — asymmetric confusion. Symmetric confusion, however, is only allowed in general ordinary Petri nets.

There are some properties that describe the dynamic behaviour of Petri nets. Amongst the most important are:

- *Liveness* A Petri net is live if in any marking reachable from the initial marking, there is always at least one enabled transition that can fire. This property ensures the absence of deadlocks in the PN.
- *Boundedness* or *safeness* A Petri net is k-bounded if in any marking reachable from the initial marking, no place in the net is marked with more than k tokens. Furthermore, a 1-bounded Petri net is called *safe*.
- *Persistence* A Petri net is persistent, if every transition, once enabled, will stay enabled until it fires. Note that a persistent PN cannot contain any conflicts (as in Figure 2.5c), as several transitions can be enabled with the marking of one place and all but one are disabled by the removal of the token by the firing transition.

When describing circuits with FSMs, the designer has to specify all possible states of the circuit and a relation which assigns succeeding states to input events. When describing with Petri nets on the other hand, all events the circuit can process or generate are modelled as PN transitions. The places can be thought of as preconditions that must be fulfilled before an transition can fire and postconditions which become fulfilled with the firing of the event. Tokens describe the fulfilment of those conditions.

The state of a Petri net is distributed over all places it contains and the computation is distributed over all transitions that operate independently. This is in correspondence to actual circuit components that, unless explicitly synchronised, operate independently and yet are part of a bigger circuit.

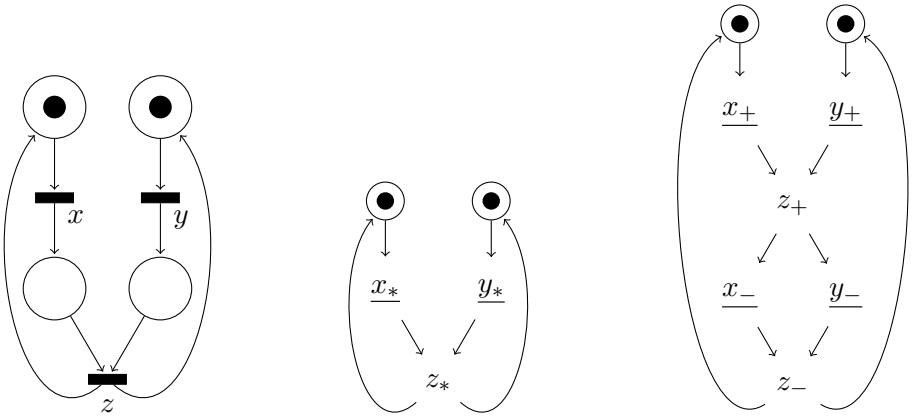
2.4.2 Signal Transition Graphs

In general, a computation described with a Petri net consists of a number of events the occurrences of which are governed by the position of tokens in the net. However, no direct correspondence of input and output signals to and from a circuit is given.

Signal transition graphs (STGs) are a form of interpreted Petri nets which create exactly this correspondence. They were introduced in [38], where the set of transitions of a PN is interpreted as transitions on signals of the circuit described, formally $T = J \times \{+, -\}$ where J is the set of all signals of the circuit. For a signal $x \in J$, x_+ denotes a rising $(0 \to 1)$ transition and x_- denotes a falling $(1 \to 0)$ transition. Using *general transitions*, denoted x_*, to represent a transition where the direction is not important is also common. In order to distinguish between input signals, that is ones controlled by the environment only, and non-input signals, these are outputs and internal state signals, the input signals are always underlined. Note that in the original definition in [38], each signal was represented by exactly two transitions — one rising and one falling. This is no longer required since present tools accept STGs with multiple transitions of the same signal in the graph.

To improve the readability of the graphical representation, signal transition graphs introduce some simplifications:

- Transitions are not drawn (as bars or rectangles) but instead replaced by their label.
- Places with one predecessor, one successor and no token are not drawn. Instead, arcs go directly from their input transition to their output transition.



(a) PN with labelled transitions

(b) STG with '∗' transitions

(c) STG with '+' and '−' transitions only

Figure 2.6: Comparison of a PN and STG for descriptions of a Muller C-gate

An example of an STG in comparison to a PN, both describing a Muller C-element, is shown in Figure 2.6. In Figure 2.6a, the behaviour of the element is described with a Petri net with transitions labelled with signals on which the events occur. The corresponding STG, using general transitions (x_*) is shown in Figure 2.6b. Often, these transitions are not supported by analysis and synthesis tools and only rising and falling transitions can be used, which duplicates their amount in the description. The STG with rising and falling transitions only is shown in Figure 2.6c.

STGs inherit all properties of PNs like liveness or safeness and their structures also have attributes like SM, MG or FC. However, there are additional attributes that describe properties associated with the interpretation of the PN transitions in STGs:

- *Output-persistence* is equivalent to PN persistence, however, only considering non-input transition. An output-persistent STG can thus only have conflicting input transitions.
- *Consistency* is given if, for each signal in an STG, rising and falling transitions are interleaved in every possible firing sequence. This ensures that every firing of an STG transition is really a transition on the corresponding output signal instead of a level assignment which might not change the state of the signal.

Different approaches to the synthesis of STGs have different requirements to the input specification. For example, the synthesis described in [38] required the input STGs to be live, safe, free choice nets, satisfying the one-token SM restriction[2] and be output-persistent. Many of the requirements can be met by behaviour-preserving transformations. In the remainder of this work, STGs will not be specially designed or transformed to meet those requirements.

Examples of tools that can process STG descriptions are *Petrify* [39], developed at the Technical University of Catalonia, and *SIS* [40], developed at the University of California,

[2]Each SM structure of the STG must contain exactly one token.

Berkeley. Both can not only be used to synthesise circuits described with STGs but also allow analysis, transformations and optimisations of STGs. In both tools, the synthesis algorithm includes the generation of a state graph, similar to a state machine having all reachable markings as states, followed by complete state coding, which may require the introduction of additional state-holding signals.

2.5 Timed Event/Level Structures

In the previously discussed description methods, timing was handled with an implicitly holding global timing assumption required for the correctness of the description or globally unrestricted (STGs). With timed event/level structures, one of the description methods allowing the modelling of timing assumptions or constraints within the circuit will be presented. Note that for most description methods, there exist timed variants, including automata (state machines) [41], Petri nets [42], as well as communicating sequential processes [43], which will be described in Section 2.6.

2.5.1 Timed Event/Level Structures

Timed event/level (TEL) structures are based on timed event rule structures introduced in [44] and were first introduced in [45] and presented with slight changes in the formal definition in [46].

A TEL structure is an 8-tuple $T = \langle N, s_0, A, E, R, C, R_0, \# \rangle$ where:

- N is the set of signals
- s_0 is the initial state of the signals
- A is the set of actions
- E is the set of events
- R is the set of rules
- C is the set of constraint rules
- R_0 is the set of initially marked rules
- $\#$ is the conflict relation

The set of signals N is initialised to $s_0 \in \{0,1\}^{|N|}$. The set of *actions* A contains rising and falling transitions of all signals, similarly to the set of transition labels in STGs, however, with the addition of the sequencing event \$ which does not cause any signal transition. In contrast to STGs[3], TEL structures allow to use the same action multiple times in a description. For this purpose, the set of *events* E pairs each action with a natural number to distinguish between instances.

The *rules* and *constraint rules* connect events to form causal and temporal relations between them. Both are of the form $\langle e, f, l, u, b \rangle$, where:

- e is the enabling event

[3]The original formal description by [38]

- f is the enabled event
- l is the lower bound of the timing constraint
- u is the upper bound of the timing constraint
- b is a Boolean function over the signals from N

The events, similar to transitions in Petri nets or STGs, *fire* if the preconditions are met. When an event fires, the action it contains is processed (for input signals) or generated (other signals). Sequencing events \$ allow to describe behaviour that requires changes in the state of a TEL structure without changing the state of any signal.

The rules specify the preconditions for the firing of events in a way somewhat similar to that of places in Petri nets. Each rule connects two events, e with f to put them in causal and temporal relation. When an enabling event (e) of a rule fires, the latter becomes *marked*. A marked rule, the Boolean function (b) of which is true, becomes *enabled*. There are two different semantics concerning enabled rules. In *disabling* semantics, if the Boolean function of an enabled rule becomes false again by changed signal states, it is reverted to being marked. In *non-disabling* semantics, once a rule is enabled, a change in the value of the Boolean function does not have any impact on the evaluation of the rule. Even though a differentiation is not part of the formal definition, both semantics can be combined in a specification. A rule that has been enabled for l time units becomes *satisfied*. When u time units elapse since the enabling of a rule, it becomes *expired*. Once the enabled event of a satisfied or expired rule fires, it is unmarked similarly to the withdrawal of a token from a place in a Petri net.

Constraint rules are virtually equivalent to standard rules. The only difference is, that while standard rules control the firing of events, constraint rules define assertions to be fulfilled when events fire — all constraint rules that enable an event must be satisfied when it fires. This essentially means that constraint rules are used for verification only while standard rules are used for circuit synthesis.

Also note the similarity between TEL structure rules and production rules (see Section 2.2). The Boolean function in TEL structure rules can be compared with the guard of a production rule and the enabled event can be compared with the assignment.

In Figure 2.7, the graphical representation of an exemplary TEL structure describing a Muller C-element is depicted. The vertices of the graph represent the set of events while rules are represented by directed arcs going from enabling to enabled events of the rules. Constraint rules share the same graphical representation with standard rules except for being marked with a 'C'. However, there are no constraint rules in this example. If a rule is initially marked, the corresponding arc is drawn with a dashed line. Furthermore, the arcs are labelled with the timing constraints and the Boolean function that are part of the rule. If disabling semantics should be applied to a rule, the label furthermore contains a 'd'. A simplification of the representation can also be seen in the figure — node labels corresponding to events that contain the only instance of an action omit its associated number. When two or more events contain the same action and therefore more instances of the latter exist, the label contains the action and the number separated by a slash (e. g. $z+/1$).

The behaviour of the circuit described with the TEL structure in Figure 2.7 is as follows: The rule having $z+$ as its enabled event (dashed arc) is initially marked. Therefore,

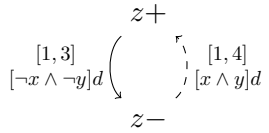


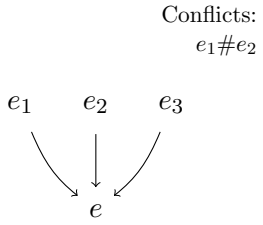
Figure 2.7: TEL structure of a Muller C-gate

the value of the Boolean function $x \wedge y$ determines the enabling of the rule. When both x and y become true, the rule is enabled. Since disabling semantics are to be applied for the rule, if either x or y return to false before one time unit elapses, the rule is disabled (marked only) again. After x and y have been true for one time unit, the rule becomes satisfied and $z+$ is allowed to fire. Another three time units later, the rule becomes expired and since it is the only rule enabling the event $z+$, it must fire by then. By the firing of $z+$ the rule is unmarked and the next rule, having $z+$ as its enabling event is marked. In a similar fashion, after one to three time units of both x and y being false, $z-$ fires.

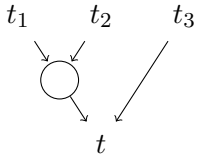
The *conflict relation* # allows to model choice and merge structures. Two events that are in conflict, denoted $e_1 \# e_2$ cannot occur at the same time. Generally, every event is allowed to fire when, in case of no conflicts, all of the rules enabling it are satisfied and it must fire before they all are expired. This behaviour is that of a join structure. If, however, multiple rules share the same enabled event e and have conflicting enabling events, e is allowed to fire after enough of these rules are satisfied to form a *sufficient set*. For a set of rules enabling some event e to be sufficient, the enabling event of every such rule that is not in the set must be in conflict with the enabling event of some rule from the set. Therefore, the conflict relation allows to model merge structures or even combine joins and merges in one structure. For example, event e in Figure 2.8a is allowed to fire after e_1 with e_3 or e_2 with e_3 have fired. This behaviour is equivalent to that of the STG in Figure 2.8b, which uses a common output place for the transitions in conflict.

Choice is modelled with multiple rules sharing one enabling event e having conflicting enabled events. When such an event e fires, all of the rules having it as their enabling event become marked. However, only one event from each *conflict set* can fire. A conflict set is such a set of events in which each event is in conflict with all other events of the set. That means, that by the firing of an event, all rules enabling events that are in conflict with that event are unmarked. Which event from a conflict set actually fires, can be the result of a nondeterministic decision or can be determined by other conditions, such as the Boolean functions within rules. As an example, consider the TEL structure in Figure 2.8c. After event e has fired, all three rules become marked, however, by the firing of e_1, the rule enabling e_2 is unmarked and vice versa. Therefore only e_1 with e_3 or e_2 with e_3 can fire. This behaviour is equivalent to that modelled by the STG in Figure 2.8d where the transitions in conflict share an input place.

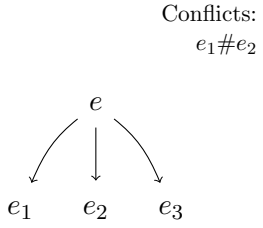
An example of a tool that can process TEL structures is *ATACS*, developed at the University of Utah. The tool can be used for the synthesis, analysis and verification of timed circuits. Even though there is no publication about ATACS as such, there are multiple publications about the algorithms it utilises [47, 48], including a POSET algorithm for an efficient timed state space exploration [49], and its successful application [50].



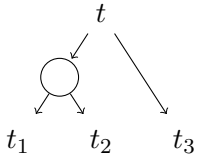
(a) Three rules with the same enabled event and conflicting enabling events



(b) An equivalent STG to the TEL structure in 2.8a



(c) Three rules with the same enabling event and conflicting enabled events



(d) An equivalent STG to the TEL structure in 2.8c

Figure 2.8: Conflicts in TEL structures with equivalent STGs

2.6 CSP-Based Descriptions

In this section, three of the most common, high abstraction level description methods comparable to hardware description languages will be presented. Namely, those are Communicating hardware processes (CHP), Haste and Balsa. All of these description methods[4] are based on Hoare's Communicating sequential processes (CSP) [51] and, as explicitly listed in the respective documentation, Dijkstra's Guarded commands [27], even though CSP is already based on the latter.

CSP will be described in the first part of this section, since the methods inherit many common constructs and characteristics from it. In the remainder, the non-CSP related common characteristics are described, after which each method with its syntax and unique features will be briefly described.

2.6.1 Communicating Sequential Processes

Communicating sequential processes (CSP) is a formal language for describing concurrent systems as set of interacting processes. CSP was first introduced in [51], an enhanced version with formal definitions to form a process algebra that allows formal reasoning about a system's behaviour was later published in [52]. Another good source of information about CSP with more recent information about its use and supporting tools is [53].

The building part of a system in CSP is a *process*. Every process in CSP is an object, that engages in various *events* which describe its interaction with other processes. Formally,

[4]Even though "languages" might be more appropriate in this case, the term description methods will be used consistently in this work, since the formerly described methods, e.g. STGs, are no languages and due to Martin's statement: *"The source notation is a program notation and not a hardware description language."* in [26].

the behaviour of a process is fully described with the set of all possible *traces* it could generate and its *alphabet*. A trace is a list of events occurring during an execution of a process, or even system. It is equivalent to a firing sequence in Petri nets or a schedule in distributed systems terminology. The times, when events occur are irrelevant and they do not have execution times. The alphabet of a process is simply the set of all events a process can participate in. Note that not all events from the alphabet of a process must occur in any of the traces it can generate.

In the following, the most important constructs for describing CSP processes will be described.

There are two predefined **special processes**, $STOP$ and $SKIP$. The process $STOP$ never engages in any events and is therefore used for the description of failure or deadlock, often as a result of an analysis of a system. $SKIP$ describes a process that successfully terminates, therefore only engaging in the special event $\surd$, which by definition can only occur at the end of a trace. Practically $SKIP$ is often used as a placeholder where the syntax of a construct requires a process but no operation should be performed.

The most basic operator in CSP is the **prefix operator**, which is of the form $(x \rightarrow P)$, pronounced "x then P". When this operator is applied to a process, the result is a new process that first engages in the event x and then behaves according to the description of P.

Recursion can be used to describe processes with long or even infinite traces. For example, a clock ticking infinitely long can be described as:

$$CLK = (tick \rightarrow tock \rightarrow CLK)$$

Furthermore, **mutual recursion** is also allowed. For example, the clock process can also be described with:

$$CLK_1 = (tick \rightarrow CLK_2)$$
$$CLK_2 = (tock \rightarrow CLK_1)$$

The **choice** construct allows to describe a process the behaviour of which depends on the first event. The syntax of the choice construct is $(a \rightarrow P \mid b \rightarrow Q)$, pronounced "$a$ then P choice b then Q", with P and Q having the same alphabet containing events a and b. Semantically the result is a process which can initially engage in either a or b and subsequently behaves like P if the first event was a and Q if it was b. Note that this construct only allows deterministic choice — a cannot be equal to b.

The **general choice** construct $(P \,\Box\, Q)$ is the generalisation of the choice construct that allows to model processes with nondeterministic decisions. If there are no events that are initially accepted by both P and Q, $\Box$ behaves exactly like $|$. The $\Box$ operator, however, is also applicable where both processes accept the same event initially. In this ambiguous case, the resulting process will behave like one of these two processes chosen nondeterministically. Please note that the decision is based on the first event only and therefore

$$((a \rightarrow b \rightarrow P) \,\Box\, (a \rightarrow c \rightarrow Q)) \neq (a \rightarrow ((b \rightarrow P) \,\Box\, (c \rightarrow Q)))$$

as in the left hand description only one of b and c is accepted after a, depending on the

nondeterministic decision, while in the right hand description both b and c are accepted and the (deterministic) choice depends only on which of these two events the process engages in. Please also note that although replacing all choice operators $|$ with general choice operators $\Box$ would lead to equivalent descriptions, it is not advisable to do so when describing circuits due to the difference in the implementation.

The **sequential composition** operator $(P\,;Q)$ allows to describe a process which behaves exactly like P until it successfully terminates (by engaging in the event $\surd$), after which it behaves like Q. Essentially, sequential composition is the prefix operator for connecting whole processes. Since the successful termination event $\surd$ can occur only at the end of a trace, it is suppressed when P engages in it to allow continued execution of the resulting process. As an example, consider the sequential composition of a process which engages in a and terminates successfully with the process P. The described behaviour is equivalent to that of the process a then P:

$$((a \rightarrow SKIP)\,;P) = (a \rightarrow P)$$

A very important operator for describing concurrent systems is the **parallel composition** operator $(P \parallel Q)$, pronounced "P in parallel with Q". This creates a process behaving like two concurrently executed processes interacting by synchronously engaging in events their alphabets have in common. Here, the alphabets of processes play an important role.

For illustration, consider the following example:

$$P = (a \rightarrow c \rightarrow d \rightarrow e \rightarrow P)$$
$$Q = (b \rightarrow c \rightarrow e \rightarrow d \rightarrow Q),$$

the alphabet of P and Q being $\{a, c, d, e\}$ and $\{b, c, d, e\}$ respectively. Initially, the process $(P \parallel Q)$ can engage in a, since it is accepted by P and not common to both alphabets or b by the same reasoning. Since c is an event shared by both processes' alphabets, both P and Q must engage in it simultaneously. Therefore, the only possible second event the process can engage in is the second of the pair a and b so that both processes are ready to engage in c. After engaging in c, the processes deadlock since P is waiting for Q to accept d and vice versa with e. Therefore, in this example

$$(P \parallel Q) = (a \rightarrow b \rightarrow c \rightarrow STOP \mid b \rightarrow a \rightarrow c \rightarrow STOP).$$

Each process can be assigned read-only or read-write access to **variables**. The variables cannot be used for communication of concurrent processes in the sense that if a process is assigned writable access to a variable, the same variable cannot be accessible from any other process running in parallel. An **assignment** has the common syntax $x := e$ for the expression e being assigned to the variable x. The expression is always evaluated prior to the assignment. Variables are useful for the description of processes. The most important construct serving this purpose is the **if-else** construct which has the form $P \lessdot b \gtrdot Q$. The result is a process that evaluates the Boolean expression b and then behaves like P if b was true or Q otherwise. Another typical use of variables is in the short notation for **loops** $b * Q$, even though formally they are simply a special case of recursion using the if-else construct.

As described above, concurrent processes use common events for synchronisation,

which is functionally equivalent to dataless handshaking. The exchange of data between processes is modelled similarly with the use of **communication events**. These have the form $c.v$, where c is the name of the communication channel and v is the exchanged value. The set of all values communicable over a channel by a process P is called the alphabet of the channel, denoted $\alpha c(P)$. To distinguish between the generation of a value and its reception and also simplify the use of variables for received values, the sender engages in the output event $c!v$ with v being the value to be transmitted while the receiver engages in the input event $c?v$ where v is a variable local to the process. Note that in traditional CSP, by convention, communication channels only allow direct connections of two processes. Broadcast and multicast channels are not allowed.

As and example for the use of communication events, consider the following process which repeatedly receives a binary value from the channel $left$ and outputs its inverse on the channel $right$:

$$INV = (left?x \rightarrow right!(1 - x) \rightarrow INV).$$

This kind of communication by means of a joint engagement in an event, which effectively means using implicitly synchronised channels with the handshaking protocols and other implementation details abstracted away, is a key property of CSP.

2.6.2 Common characteristics

There are many similarities between CSP and the considered CSP based description methods. There are, however, also some additions and changes that improve the description of hardware with those methods. Those common to all three methods will be described in this subsection.

- An important addition to CSP introduced in [54] is the *probe*. A process, in any state can accept a set of events while not accepting others. It is suspended until another process (or the environment) is ready to engage in one of the accepted events. For the description of hardware, it is important to be able to change the behaviour of a process (circuit) based on whether an event is *pending* or not. Choosing a pending event to be the next accepted event will guarantee immediate engagement without suspension. A probe of an event is a Boolean expression which evaluates to *true* when the event is pending. In hardware, this represents a pending communication on a channel. Another variant is the *dataprobe*, which allows access to values from pending communications.

- In the description methods, there is no such distinction between events and processes as in CSP. Both are referred to as *statements*. This also makes the prefix operator obsolete, since all statements can be composed with the sequential composition operator. Note that when describing the syntax of the methods below, "statement" can also refer to multiple statements composed sequentially and/or concurrently.

- A synchronised communication like the mutual engagement in events in CSP, whether associated with a data exchange or not, is performed over *channels*. To allow the mapping of descriptions to hardware, all channels as well as variables must be declared with a predefined *type*. All methods allow the use of common types a programmer would expect — signed and unsigned integers with definable bit widths,

Booleans, enumerated types as well as composite types like arrays and structures. The assignment of variables uses the common syntax `variable := expression` in all the methods.

- Since all expressions need to be mapped to hardware, only a limited number of supported operators can be used. All methods support all common Boolean operators, basic arithmetic operations like addition, subtraction, multiplication, comparison, and, usually only for expressions that can be evaluated at compile time, division and exponentiation.

- None of the description methods allows to describe processes recursively. This allows to determine the required amount of storage directly from the description, since recursion generally requires dynamic storage (comparable to a stack in software). However, tail recursion, i.e. recursive calls at the end of a process, procedure or function just before termination, does not require dynamic storage (stack space) but, nevertheless, is not allowed. Although several loop constructs as special cases of recursion are available, they do not support descriptions equivalent to mutual recursion.

2.6.3 Communicating Hardware Processes

Communicating hardware processes (CHP) is the first of the three description methods in this work which are based on CSP. CHP is used as the highest abstraction level description method in the *Caltech Asynchronous Synthesis Tools* (CAST) [25]. It is described in [26] as a program notation, not a hardware description language, with a syntax that uses non-standard characters and therefore cannot be directly input to a computer. In the manual for *CHPsim* [55], a simulator for the CHP language, the complete syntax of a machine-readable CHP description is given. This will be used as a reference for CHP throughout this work. Another document [56], available only in its incomplete form, describes the CAST language used as a common input language for tools which are a part of the CAST toolchain. This language allows the designer to describe a circuit at different abstraction levels, CHP being at the top end, PRS (see Section 2.2) being the bottom end.

In the following, the fundamentals of CHP and its typical constructs with their syntax will be briefly described along with differences between CSP and CHP.

- The term "process" is used for an entity describing a circuit block that can be later instantiated and interconnected.

- The statement **`skip`** provides the same functionality as the $SKIP$ process. No statement with the function of $STOP$ is provided.

- Choice and general choice have a similar syntax in CHP as that of CSP:

  ```
  [ expression -> statement [] expression -> statement ]
  [ expression -> statement [:] expression -> statement ]
  ```

  However, these constructs are more similar to Dijkstra's guarded commands than CSP. The Boolean expressions are called *guards* and their evaluation is deciding. If no guards are *true*, the process is suspended until a choice can be made. General

choice allows nondeterministic decisions when multiple expressions may become *true* at the same time, therefore uses an arbiter in the hardware implementation.

- There are three constructs for loops:

  ```
  *[ expression -> statement [] expression -> statement ]
  *[ expression -> statement [:] expression -> statement ]
  *[ statement ]
  ```

  The first have similar behaviour to choice and general choice, however, no loop construct ever suspends. When at least one guard is *true*, it is selected and after the termination of the associated statement, the selection is repeated. When no guards are *true*, the construct terminates. When used with only one choice, the first construct implements a simple while-do loop. The last construct simply repeats a statement indefinitely.

- The sequential composition operator is '`;`' as in CSP, the parallel composition operator, however, uses a simple comma '`,`'. Their functionality remains the same as in CSP. Parallel composition binds tighter, braces can be used to override operator precedence.

- Variables are declared at the beginning of a process' body and are local to that process. As in CSP, they cannot be used for communication between concurrent processes (since being local) and concurrently composed statements. Apart from the standard assignment statement using '`:=`', Boolean variables can be assigned a positive or negative value by appending a '`+`' or '`-`' after the name of such variable.

- For the description of communication in CHP, the statements *`port`*, *`port`*`!`*`expression`* and *`port`*`?`*`variable`* can be used for dataless synchronisation, output and input, respectively. Additionally, *`port`*`!`*`port`*`?` can be used for a *pass*, which is a direct retransmission of a value without intermediate storage or processing, and *`port`*`#?`*`variable`* describes a *peek*, where a value is received but not yet acknowledged so that a subsequent input statement receives the same value. CHP does not support channels internal to a process, but only external ports, each of which must be defined as a process parameter. In the definition, the direction of a port is denoted with an '`!`' or a '`?`' for output or input ports, respectively. Dataless synchronisation ports are defined without a direction and type.

- The probe is of the form *`#port`*. The dataprobe is limited to a Boolean expression

  ```
  #{ port1, port2 : expression }
  ```

  which evaluates to *true* if the probes of all ports in the list are *true* and the Boolean *`expression`*, which can refer to ports as read-only variables, also evaluates to *true*.

A usual CHP description of a circuit consists of process definitions and at least one *meta* process, which instantiates CHP processes and describes their interconnection. Common tasks can be defined in *procedures*, which can be used as statements in processes, and *functions*, which return a value and therefore can be used as expressions. Processes can also contain *meta parameters* which can be used for the description of parametrised circuit

blocks. For example, a process can describe a FIFO buffer with its width and depth configurable with two parameters during instantiation.

As already mentioned, CHP is the high-level description method used in tools included in the CAST toolset, developed at the California institute of technology. It includes tools for the synthesis and simulation of CHP descriptions. The synthesis process is described in [26] and includes transformations such as process decomposition, handshaking expansion and the transformation into PRS, which are described in Section 2.2.

2.6.4 Haste

The next description method based on CSP is *Haste*. It is the description method used in the asynchronous circuit design flow supported by the design environment called *TiDE*. TiDE has been commercially distributed by *Handshake Solutions*, which later on became a part of Philips. Haste was originally known under the name *Tangram*. Also, its notation was similar to that of CSP, and therefore also CHP, before it changed to a more verbose syntax, improving the readability of the code. Detailed information about the syntax and semantics can be found in the Haste language manual [57].

The most important constructs, syntax examples and some important properties of the description methods are shown below.

- The term "process" is not used in Haste. A description only contains procedures and functions.
- Both special processes, $SKIP$ and $STOP$ are available as special statements with the same name.
- There are several constructs for choice. The **if** and **case** constructs known from common general purpose programming languages are to be used with stable expressions and they terminate if no expressions are *true*. The select construct, which is intended to be used with unstable expressions such as channel probes, performs arbitrated selection and suspends until some expression becomes *true*. The syntax of the **if** construct is:

  ```
  if expression then statement
  or expression then statement
                else statement
  fi
  ```

  The syntax of selection is very similar, however, without the optional else statement and using the **sel** and **les** keywords.
- The general form of a loop construct is the **do** construct with the syntax:

  ```
  do expression then statement
  or expression then statement
                else statement
  od
  ```

  As in CHP, it has similar functionality to the **if** construct with repetition until no expression is *true*. The optional **else** clause is equivalent to **or true then** and therefore causes the loop statement to never terminate. Other loop constructs are:

```
repeat statement until expression
forever do statement od
for expression do statement od
```

The first two are special forms of the **do** construct, however, allow a more efficient hardware implementation. The **for** construct can be used to repeat a statement a bounded number of times, resulting in the instantiation of a counter.

- Sequential composition is denoted with a '`;`' and parallel composition with a '`||`'. The parallel composition operator binds tighter and brackets can be used to override operator precedence.

- Variables in Haste can have scopes spanning several procedures and can be accessed even from concurrent statements. Modifiers can be used to determine whether the mutual exclusion introduced for concurrent assignments requires an arbiter (**arb**`!`) or not (**narb**`!`) and for the purpose of error suppression for read/write accesses from concurrent statements (**narb**`:`) when conflicts are precluded by design. Furthermore, the **ff** modifier can be used when a variable should be implemented with a flip-flop instead of a latch, thus allowing auto-assignment[5] without the need for an additional variable for temporary storage.

- Communication statements in Haste have the syntax *channel*`!`*expression* and *channel*`?`*variable* for an output and input, respectively. The dataless synchronisation uses the same statements with the symbol '`~`' in place of the expression or variable denoting that no data is exchanged and the same symbol is used as the type for dataless channels. Haste allows channels to be used for communication with the environment, when being procedure parameters, but also for connecting internal statements with a handshaked communication. Moreover, each channel can be read and written to from multiple statements, even concurrent ones. The modifiers **broad** and **narrow** can be used to determine, whether all concurrent receivers should be involved in a single communication (broadcast channel) or always just a single one (narrowcast channel). The modifiers **arb**`!` and **narb**`!` can be used to specify whether arbitration is needed to resolve conflicts if two or more concurrent output statements access the same channel and in case of a narrowcast channel the analogous modifiers **arb**`?` and **narb**`?` are used for concurrent input conflicts. When channels are used as parameters, the additional modifiers **act** and **pas** can be used to determine whether the ports should be active or passive.

- The probes in Haste have the syntax **probe**`(`*channel*`)`, **inprobe**`(`*channel*`)` and **outprobe**`(`*channel*`)`. The general probe evaluates to *true* if any action is pending on a channel, whereas the inprobe and outprobe only denote pending input and output actions, respectively. The **dataprobe**`(`*channel*`)` returns the current value on a channel's wires and should therefore only be used after an outprobe to guarantee stable values. For the completion of a handshake, the dataless input (*channel*`?~`) can also be used with channels of other types when the value should be discarded.

[5] An auto-assignment is such that the variable being assigned is also referenced in the expression of the assignment statement, e.g. *x* `:=` *x*`+1`.

- In addition to synchronised communication over channels, Haste allows the use of **wire**-s to represent non-handshaked signals. The value of each internal or output **wire** is determined by one expression which is continuously evaluated and therefore may result in unstable values.

- There are a number of functions which work with unstable expressions, such as probes an wires. Where constructs require stable expressions, the **sample**(*expression*) function, which introduces a synchroniser with a metastability filter, can be used to obtain a stable sample for evaluation. The **wait**(*expression*) statement suspends until the expression is *true*. Additionally, the statements **edge**(*expression*), **posedge**(*expression*) and **negedge**(*expression*) can be used similarly to **wait**, however, they terminate only on encountering any, a positive or a negative transition, respectively.

- In Haste, there is a very important distinction between a definition and a declaration of procedures and functions. A call of a defined procedure or function results in the instantiation of a dedicated hardware block performing its function at the place of call. A declaration, on the other hand, results in the instantiation of hardware by itself. A call of a declared procedure or function activates the existing block, in case of multiple calls introduces multiplexing of call parameters and results as well as mutual exclusion of calls. In the description, a definition is denoted with an equal sign between an identifier and its type while a declaration uses a colon instead.

The aforementioned design environment TiDE [58] includes tools for the analysis, simulation and compilation of Haste descriptions. During compilation [59], a description is first translated into a *handshake circuit*, which is composed of interconnected simple blocks such as a sequencer or an adder. This representation can then be used for the mapping onto a user-defined technology library.

2.6.5 Balsa

The last of the three considered description methods based on CSP is *Balsa*. Balsa is the name of both the description method, and the tool for circuit synthesis using the method. Both have been developed at the University of Manchester. Balsa was motivated by Tangram, the predecessor of Haste/TiDE, which manifests mainly in a related synthesis method. The full description of the language can be found in [60].

The most common constructs with syntax examples and properties of Balsa are listed below.

- Like in Haste, the term "process" is not used in Balsa. Instead, a description is composed of procedures and functions.

- The special processes $SKIP$ and $STOP$ are implemented as the **continue** and **halt** statements, respectively.

- Multiple constructs are provided for the description of choice. Firstly, there are the common constructs **if** and **case** with the same functionality as in Haste. The syntax of the **if** construct in Balsa is:

```
if expression then statement
|  expression then statement
              else statement
end
```

Then there are two selection constructs for choosing amongst channels. One is **select** and the other is **arbitrate**. They are almost equal with respect to syntax and semantics, the only difference being that the former denotes non-arbitrated selection, where no two inputs can become *true* simultaneously, while the latter introduces arbitration to the selection to resolve possible conflicts. The syntax of the select construct is:

```
select channel then statement
|      channel then statement
end
```

It should be noted that selection implicitly evaluates probes of the listed channels and that within the statements after the selection, the channels behave like read-only variables. Therefore, they can be directly used in expressions, and assignments should be used instead of input statements to read a channel value. The handshake of a selected channel is completed when the corresponding statement terminates.

- Balsa has a very universal loop construct with the syntax:

```
loop
  statement
while expression then statement
|     expression then statement
                also statement
end
```

The statement between **loop** and **while** is optional and it is executed at each loop iteration before evaluating the expressions. The expressions are evaluated in sequence, therefore, when multiple evaluate to *true*, the first is always chosen. Note that the expressions must be stable during evaluation. The optional **also** clause can be used for statements that should be executed at the end of each loop iteration excluding the last, where no expression was *true*, after which the loop construct terminates. The simpler constructs

```
loop statement end
loop statement while expression end
```

can also be used. Even though there is a **for** construct, this is used to generate multiples of similar hardware blocks which can be compared to full loop unrolling, and should not be confused with loops, where the same hardware is used for each iteration.

- Sequential and parallel composition use the same operators as in Haste, '`;`' and '`||`', respectively. Again, the parallel composition operator binds tighter and square brackets can be used to override operator precedence.

- Variables in Balsa can be declared within procedures and functions and since these can have overlapping scopes, variables can also be used to exchange data between

them. However, implicit mutual exclusion, optionally with arbitration, of concurrent accesses to variables is not supported by Balsa. Moreover, read-write accesses of variables from concurrent statements are not allowed. Auto assignment of variables is allowed in Balsa and the required temporary storage will be introduced by the compiler.

- The communication in Balsa uses the syntax

  ```
  channel <- expression
  channel -> variable
  channel -> channel
  channel -> then statement end
  ```

  for output, input and pass, respectively. The last line shows a construct that waits for a pending output on the specified channel after which the statement is executed. As in the selection construct, the channel can be used as a read-only variable within that statement. The handshake is completed when the statement terminates. Even though this construct has the same functionality as a **select** with a single channel, there is one notable difference. All ports in Balsa are active by default. The only exception is that when channels are used in a selection, they become passive. This does not apply to the **channel -> then** construct, where the channel, even though it is being probed, remains active. Write access to channels from concurrent statements is not supported in Balsa.

  Dataless synchronisation channels are declared by using the keyword **sync**, in case of procedure parameters, **sync** is used instead of the direction, **input** or **output**. The same keyword is used in the actual communication statement, **sync channel**, which performs a handshake on a channel, possibly suspending until the handshake completes.

- There are no explicit probes in Balsa. The functionality of probes and dataprobes needs to be described with the handshake enclosure constructs, **select**, **arbitrate** or **channel -> then**.

The design compiler from the Balsa toolset [61], being also based on Tangram, translates high-level descriptions into handshake circuits like the TiDE compiler. The most notable difference between TiDE and Balsa is that Balsa allows the handshake circuits to be parametrised, thus allowing more efficient implementations.

Chapter 3

Comparison by Example

In this chapter, the considered description methods will be illustratively compared by means of the design of an exemplary circuit with all of these methods. Furthermore, the examples will serve as additional sources for better understanding of the description methods.

In the first part of this chapter, the chosen circuit will be informally described and the motivating factors for its selection will be enlisted. In the subsequent parts, the iterative description of the circuit, starting from its building blocks, with all of the considered description method will be performed. Possible design choices and their effect on the description and the strengths and shortcomings of the description methods will be discussed.

3.1 The Circuit

The circuit chosen to be used for the first comparative example is a *ring topology network interface controller*. It is a circuit of moderate complexity that provides an attached host with means of communication using a simple interface and protocol.

Data transported over the network are organised in packets, the head word being used for addressing using a hop counter which is decremented before leaving each node. If this counter is zero in a received packet, the whole packet, except for the counter itself, is delivered to the host attached to the current node.

To avoid deadlocks in the network, which may result from multiple nodes transmitting simultaneously and therefore blocking the links, the circuit controls access to the network by passing a *privilege token* over the ring. This arbitration protocol is equivalent to the one described by Martin in [26] in the example "Distributed Mutual Exclusion on a Ring of Processes". The example in this work adds communication over the ring to this description in replacement of the CS process. Furthermore, Low and Yakovlev study and describe different token ring arbiters with Petri nets and STGs in [62]. The equivalent to Martin's mutual exclusion example is the "Lazy Ring Protocol". Again, only the problem of mutually exclusive access to a shared resource is addressed in their work.

For the communication with the attached host, the circuit provides a data channel associated with a dataless synchronisation channel in each direction. The synchronisation channel is used to indicate the end of a transmission. In order to send a message through the network, the host first communicates the destination address in the form of a number representing the distance of the destination host from the source, the value zero causing

the return of the message to the sender without crossing any network links. The handshake governing the passing of the address triggers the network arbitration protocol in the circuit and is not completed until the token has been acquired. The following data from the host will be delivered to the addressed communication partner by being passed as many times as the address word specified. After communicating the last word of the message, the host indicates the end of the packet by a synchronisation on the dataless channel used for this purpose. Every instance of the circuit, i.e. a node of the network, has a one place buffer. Note that sending a packet with an address greater or equal to the number of nodes in the network would cause a deadlock, if it was long enough to fill all the nodes' buffers.

3.1.1 Building Blocks

The circuit can be divided into two main parts: network arbitration and data handling. The function of both parts will be informally described below and constructs typical for asynchronous circuits they might contain will be adverted to. The structure of the circuit with all external and internal channels is depicted in Figure 3.1.

The first part described, the *arbitration* block, is responsible for managing access to the network to guarantee that only one host at a time is transmitting. It is a modification of the aforementioned descriptions from [26] and [62]. Two external synchronisation channels, t_R and t_L, are used for requesting and passing the privilege token from and to neighbours. An internal channel, t_U is used for requests to transmit data and signalising the end of transmission to allow the passing of the token to potentially waiting neighbours. Furthermore, either an additional channel to the data handling part or access to a shared variable must be present to prevent passing the token while a data packet is relayed from one neighbour to another. In the Figure 3.1, this is the channel *t_hold*. It ensures that the token cannot overtake a packet crossing the network in the same direction which could lead to a deadlock.

The main building block of this arbitration block is a basic arbiter, which chooses between the independent requests from either the left-hand neighbour or the attached host, i.e. t_L or t_U. If the current node is in hold of the token, the chosen request is acknowledged, else the request is forwarded to the right-hand node, i.e. the t_R port.

The second part is responsible for handling data. The circuit has four data channels in total, each being associated with a dataless channel used to signalise the end of a packet. The nomenclature for those channels is as follows: Each data channel has the prefix d while the associated synchronisation channel is prefixed with e. Then follows an underscore and one of L, R, U denoting where the channel should be connected to. L and R denote the left- and right-hand neighbours, respectively, U denotes channels connected to the host ("up"). Finally, i and o, standing for input and output, denote the direction of the data transfer. All external ports of the circuit are summarised in Figure 3.1.

When a host initiates a transmission, the token is first requested using the internal t_U channel. Only after the request has been acknowledged, the data is routed through the network. After the transmission, an additional synchronisation on the t_U channel unlocks the token allowing it to be passed to neighbours again.

Since the arbitration protocol ensures that if a host is allowed to transmit, no data can arrive on the *d_Ri* port until the transmission is finished, the data coming from the host and from the neighbour can be merged without arbitration. The first word of a packet contains the address and is therefore compared for zero after merging. If the value was

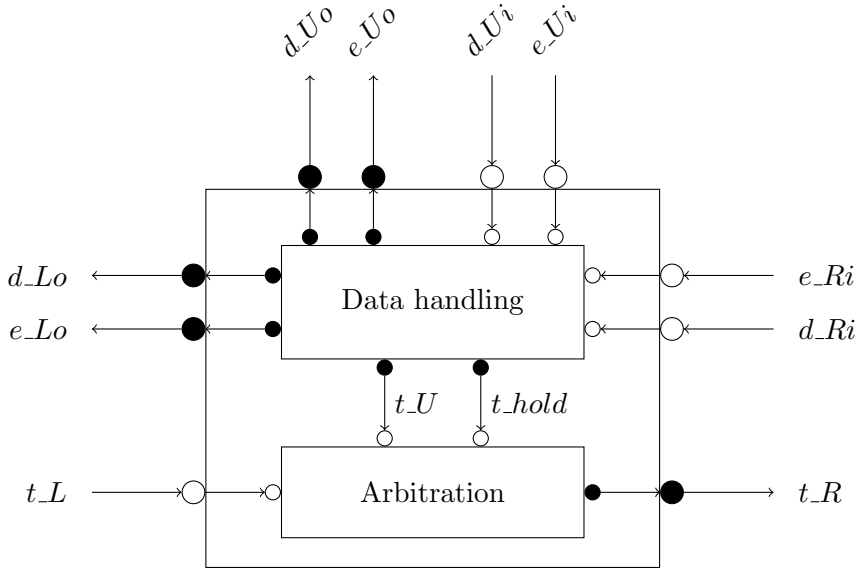


Figure 3.1: The structure of the circuit with external and internal channels, filled and unfilled circles denoting active and passive ports, respectively.

non-zero, the packet is forwarded to the left-hand neighbour with a decremented address, otherwise the address is discarded and the following words of the packet will be delivered to the attached host. However, before being demultiplexed, the data are buffered in a one place buffer.

3.1.2 Motivation for the Choice

This circuit was chosen, because it contains many constructs that are typical in asynchronous design. The most notable ones being:

Arbitration The selection of requests from the channels t_L and t_U must be arbitrated, since the sources of these requests are independent. This will show the description of possibly the most important construct in asynchronous circuits — arbitrated selection.

Merging The selection of requests that are a priori known not to occur simultaneously, as are data from d_Ri and d_Ui[1] when being merged in this circuit, can be implemented more efficiently by omitting the arbiter. The difference between the descriptions of arbitrated and non–arbitrated selection can therefore be shown.

State holding The description of state holding variables will be demonstrated. Examples of such variables are one that determines the presence of the token and another that saves the destination for the current packet.

Buffering The buffer in this circuit will allow to show how to describe data storage and the necessary input and output channels with sequenced access to the buffer.

[1] After the privilege token has been acquired.

Arithmetics In this circuit, the address value must be compared to zero and decremented if forwarded. With an eight bit wide channel as an example, there will be vast differences in the comfort of describing those arithmetic operation depending on the abstraction level of the description method.

Handshake enclosure In this circuit, handshake enclosure is used for obtaining the privilege token within the handshake of the address value from the host, or when performing address manipulation before buffering the data.

Abstraction Two abstractions that simplify the design to the biggest extent are a) handling multiple bit wide variables and channels as units, not as separate signals and b) hiding handshaking signals and protocols. Although it is clear that the high level, CSP-based languages will include these simplifications, the implications of the abstractions will be observed.

3.2 The Descriptions

This section shows the description of the exemplary circuit. Please note that for the low abstraction level description methods that require the behaviour of each signal to be separately described, namely PRS, AFSMs, STGs and TEL structures, only the arbitration block and a half-adder with dual rail return to zero (NCL) coded inputs and outputs will be described. Such half-adder could be used for the implementation of the decrement operation of the address word, and the complexity of the descriptions of even this 2-bit data path block should demonstrate that describing the full 8-bit data path with those methods would be unfeasible.

3.2.1 Production Rule Sets

The description of the token ring arbitration block in PRS will start with the basic arbiter that is needed to ensure mutual exclusion of the requests:

$$
\begin{aligned}
t\_L_R \wedge \bar{U} &\mapsto \bar{L}\downarrow \\
t\_U_R \wedge \bar{L} &\mapsto \bar{U}\downarrow \\
\neg t\_L_R \vee \neg \bar{U} &\mapsto \bar{L}\uparrow \\
\neg t\_U_R \vee \neg \bar{L} &\mapsto \bar{U}\uparrow
\end{aligned}
$$

In the description, the signals t_L_R and t_U_R are the high active unrestricted external requests from the left hand neighbour and the attached host, respectively. $\bar{L}$ and $\bar{U}$ are the output signals representing mutually exclusive versions of the requests. The bars above the signal names denote low active signals and are not part of the PRS syntax. They have been added to the description for better readability only.

The description of the arbiter is an exact copy from [26], with renamed signals. The newly introduced signals, $\bar{L}$ and $\bar{U}$, represent the requests after arbitration. Note that since for both signals, the complementary production rules also have complementary Boolean expressions in their guards, this PRS describes two combinational operators, more precisely the two NAND gates that the bare arbiter consists of (see Figure 1.4). However, the analog

filter that ensures correct operation in the case of simultaneous requests is missing and there is no possibility to describe its presence with production rules. Also note that, if the request lines go high simultaneously, the stability of the PR's can be violated.

The rest of the PRs deal with the generation of the output signals for requesting the token and acknowledging requests but also introduce a state-variable "T" which indicates the presence of the token:

$$
\begin{aligned}
(t\_R_A \vee T) \wedge \neg\bar{U} \wedge \neg t\_L_A &\mapsto t\_U_A \uparrow \\
\bar{U} &\mapsto t\_U_A \downarrow \\
(t\_R_A \vee T) \wedge \neg\bar{L} \wedge \neg t\_U_A \wedge \neg t\_hold &\mapsto t\_L_A \uparrow \\
\bar{L} &\mapsto t\_L_A \downarrow \\
(\neg\bar{U} \vee \neg\bar{L}) \wedge \neg T \wedge \neg t\_R_A \wedge \neg t\_L_A \wedge \neg t\_U_A &\mapsto t\_R_R \uparrow \\
t\_L_A \vee t\_U_A &\mapsto t\_R_R \downarrow \\
t\_U_A &\mapsto T \uparrow \\
t\_L_A &\mapsto T \downarrow
\end{aligned}
$$

In this description, all four PR pairs describe state holding operators. Also, unlike in the PRS describing the arbiter, the mapping to transistors is not straight forward, as there are a number of variables that would require an additional inverter before their use as inputs to either the positive or negative transistor stack.

The function of the circuit can be understood from reading the description with moderate effort. The first two operators generate the acknowledge signals t_U_A and t_L_A, which are going to the attached host and the left hand neighbour, respectively. Except for the t_hold signal needed to suspend the acknowledgement of the left hand neighbour's request, they are equivalent. Both acknowledge a request if it is present ($\neg\bar{U}$), the other request is not acknowledged ($\neg t\_L_A$) and either the token is present at the current node or the right hand neighbour is passing it by an active acknowledge signal ($t\_R_A \vee T$). The acknowledgement remains high until the respective request is withdrawn. The next pair of PRs requests the token from the right hand neighbour (t_R_R) if some of the requests is active ($\neg\bar{U} \vee \neg\bar{L}$) while the node is not in possession of the token ($\neg T$) and none of the acknowledge lines is high ($\neg t\_L_A \wedge \neg t\_U_A$), including the one from the right hand neighbour ($\neg t\_R_A$) which must be low before the next request for correct four phase protocol operation. The last two PRs describe the state variable T which represents the presence of the token in the node. It goes high when the request to the attached host is acknowledged (t_U_A) and similarly it goes low when the request from the left hand neighbour is acknowledged (t_L_A).

Although describing operators as well as understanding existing descriptions using PRS is very intuitive and straight forward, the disadvantage of this description method is that it does not visualise the interconnections hiding behind variable names nor any causality between different assignments. This makes understanding the dynamic behaviour of a circuit more difficult. As an example, note that in the description above, there is a race condition which may cause an erroneous request for a token. The problematic expression is "$\neg T \wedge \neg t\_L_A$" which could produce a glitch if the node was in possession of the token, the left hand neighbour requested it which would cause t_L_A to go high immediately and subsequently T being pulled low. The erroneous request would be generated, if the fork of

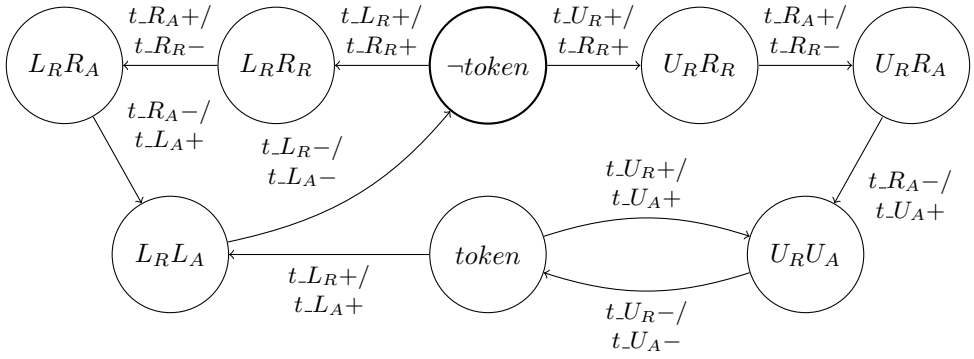


Figure 3.2: The burst-mode AFSM description of the arbitration block

the t_L_A signal had an unfavourable difference in delay causing it to arrive at the input of the t_R_R operator after T has already gone low. This could be avoided by introducing another variable masking this possible source of error.

The description of the NCL half adder with a PRS follows:

$$\begin{aligned} A_p \wedge B_p &\mapsto S_n \uparrow, C_p \uparrow \\ (A_p \wedge B_n) \vee (A_n \wedge B_p) &\mapsto S_p \uparrow, C_n \uparrow \\ A_n \wedge B_n &\mapsto S_n \uparrow, C_n \uparrow \\ \neg A_p \wedge \neg A_n \wedge \neg B_p \wedge \neg B_n &\mapsto S_p \downarrow, S_n \downarrow, C_p \downarrow, C_n \downarrow \end{aligned}$$

In this example the advantage of a description method which uses signal levels is apparent. The three first PRs set the appropriate output signals high when both inputs change from *NULL* to either a positive or negative value. The last PR is responsible for resetting the output signals to *NULL* when both inputs change to *NULL* again, as required by the chosen four phase protocol.

Again, it can be observed that all four operators are state holding. This is necessary in order to adhere to the prescribed protocol. The C_p and C_n operators could also have been implemented with a single combinational *AND* and *OR* gate respectively. However, this implementation would violate the four phase protocol in that it would output the C_n value before both inputs would change to a valid value and C_p would go to *NULL* before both inputs were *NULL*. Both violations are acceptable provided that timing assumptions prohibit race conditions in circuits using such blocks.

3.2.2 Asynchronous Finite State Machines

The description of the main part of the circuit's arbitration block with a burst-mode AFSM is depicted in Figure 3.2. The state names are chosen as lists of positive valued signals in each state, input and output signals mixed and both without the "$t\_$" prefix. As exceptions, the names of the two idle states where no signal is high show whether the circuit is in possession of the token in that respective state. This naming scheme was chosen to show that the unique entry point requirement is fulfilled, that inputs and outputs have the same values in a state, no matter from where the machine changes to it.

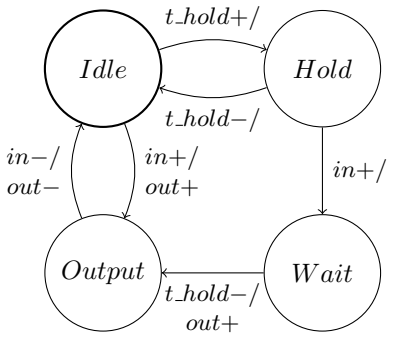


Figure 3.3: The BM AFSM description of the token holder

The function of the state machine can effortlessly be comprehended from the figure: Initially the state machine is in the state $\neg token$ and waits for requests. Depending on which request will arrive, it changes to one of the following states while generating the t_R_R request in order to obtain the token. Once the token has been obtained (t_R_A+) the handshake with the right hand neighbour is completed and the machine turns to a state where it is in possession of the token and can therefore acknowledge the request. The function of the rest of the arcs will not be further explained.

The problem with this description, however, is, that an AFSM requires its environment to adhere to some fundamental mode restriction. This also means, that when one request arrives in an idle state, no further inputs are allowed to change until the machine stabilises. This means, that the required arbitration of the two request sources is implicitly assumed to be done before the signals are fed to the machine. The description in Figure 3.2 is only correct if t_L_R and t_U_R are (non-inverted) outputs of an arbiter preprocessing these signals.

The biggest disadvantage of using AFSMs for describing asynchronous circuits is the impossibility to model concurrency. Unlike production rules, which are all evaluated concurrently and therefore describing sequential behaviour requires additional effort, state machines are inherently sequential but there is no way of modelling concurrent events unless their occurrences do not violate the fundamental mode. As an example, consider that the *t_hold* signal is allowed to rise and fall concurrently to the circuit's operation with the only restriction being that it must not rise at the time when t_R_A goes high until an acknowledge is generated. Adding this behaviour to the AFSM in Figure 3.2 would require doubling every state with *t_hold* high in one and low in the other. Instead, a second AFSM that operates concurrently with the one described above and implements the function of the *t_hold* signal is introduced. Please refer to Figure 3.3 for its description. The signals *in* and *out* denote the t_L_A signal coming from the AFSM in Figure 3.2 and going directly to the external interface, respectively.

In Figure 3.4, the NCL half adder is described. Here, the difference between describing simple logic functions as part of an RTZ data flow with level sensitive descriptions, as are production rules in the previous subsection, and transition (event) sensitive descriptions can be observed. Although in the *NULL* to data phase, the descriptions are very similar since every positive level is caused by a rising edge, in the data to *NULL* phase, not all negative levels are caused by falling edges. Therefore in BM AFSMs and other edge

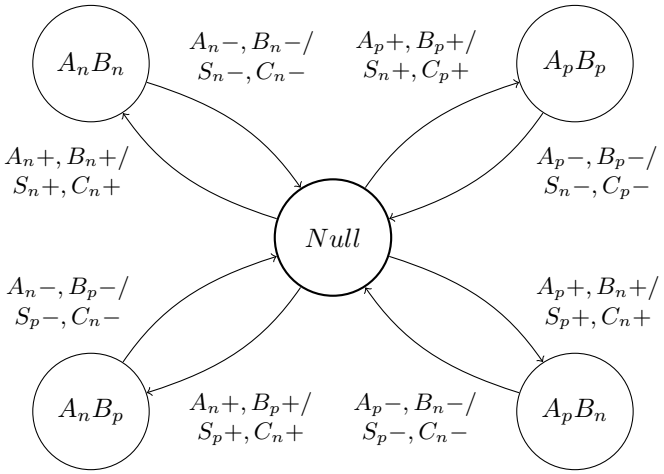


Figure 3.4: The BM AFSM description of an NCL half adder

sensitive descriptions, the different output combinations which cause different falling edge combinations have to be distinguished.

Also note that the higher abstraction level that AFSMs offer by hiding sequencing and state variables disallows to model the area optimised implementation with simple combinational gates for the C_n and C_p outputs as described in Section 3.2.1. While changing the description to output C_n after any of the inputs reads a valid negative value is possible, the fundamental mode operation would require the other input not to change until this state transition was finished which, of course, is unacceptable.

3.2.3 Signal Transition Graphs

As mentioned in Section 3.1, an STG description of the arbitration block, albeit without the *t_hold* signal, can be found in [62]. This description was derived from a higher abstraction level Petri net model in two transformations. In the first step, Petri net transitions were turned into signal transitions leading to an STG with general transitions which implements the behaviour of the Petri net with a circuit that uses NRZ signalling. In the second step, the model was expanded to consist of rising and falling transitions only, therefore describing a circuit with RTZ signalling. In this work, a simpler description was chosen for better readability of the STG. The difference in functionality can be observed when the left hand neighbour requests the token and the circuit needs to request it from the right hand neighbour. After the right hand neighbour acknowledges the request, the circuit changes to the state of being in possession of the token, which in turn allows the token to be passed to the left hand neighbour therefore losing it again. In an implementation, this would lead to a pulse on a variable that would indicate possession of the token which would make the circuit slower and less energy effective.

In Figure 3.5, which depicts the STG of the arbitration block, the description of the arbitration of the two request sources can be observed. The topmost place represents the core functionality of this circuit, the mutual exclusion. Since it is a common predecessor of both of the request transitions, it creates a conflict therefore allowing only one of them to fire. The token is returned to this place when the shared resource, in this case

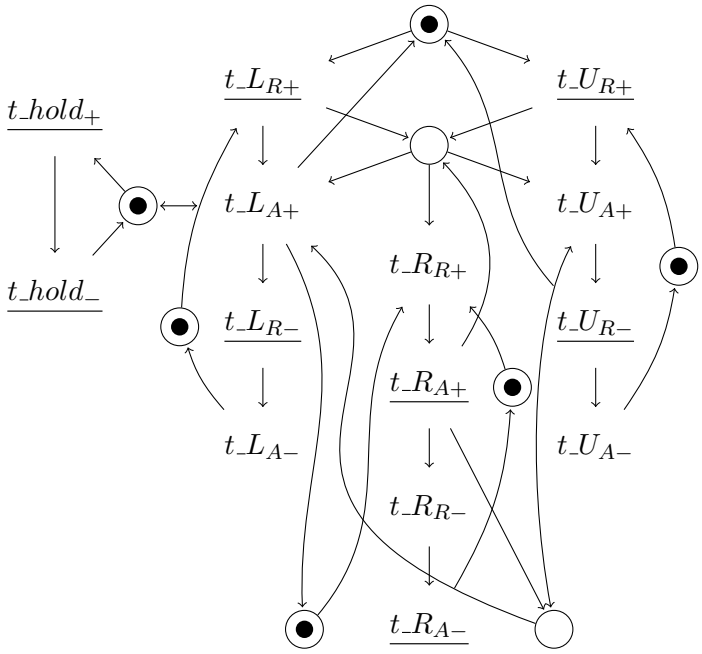


Figure 3.5: The STG description of the arbitration block

the network, is free again. Although this allows simple modelling of mutually exclusive behaviour, explicit specification of *OR*-behaviour without the need of arbitration is not possible. As an example, consider the place below the mutual exclusion place, which is used to start a right hand request cycle in case the token is not present in the circuit. It is a common predecessor of the t_L_{A+}, t_U_{A+} and t_R_{R+} transitions therefore introducing mutually exclusive behaviour amongst them. There is however no marking in which any two of those would be enabled at the same time which effectively means that no arbiter is needed to resolve any conflicts. On the other hand, if t_U_{R+} and t_L_{R+} were mutually exclusive, but known not to occur simultaneously, there is no way to model this without fully describing the environment's behaviour within the STG.

The advantages of the possibility to model concurrency with STGs can also be observed in the description in Figure 3.5. As a first example, consider the transitions of the *t_hold* signal which can fire concurrently and only affect the circuit's behaviour when t_L_{A+} is generated. As another example, note that after t_L_{A+} fires, the communication with the left hand neighbour is completed and the token from the mutual exclusion place is returned, therefore allowing a request on t_U_R to be processed. The sequential execution of t_L_{R-} and t_L_{A-} before t_L_{R+} can fire again, required by the RTZ signalling scheme, is performed concurrently to the operation of the rest of the circuit.

The next example which is the description of the NCL half adder can be found in Figure 3.6. Although RTZ signalling was specified, this STG uses general transitions for simplification of the graph and therefore effectively implements a circuit that uses NRZ signalling. Note however, that even with this description, the resulting circuit would be compatible with RTZ environments (assuming correct initial signal values), albeit considerably slower and of bigger complexity than circuits designed specially for NCL.

As a brief description of this STG, applying valid values to both inputs will cause two

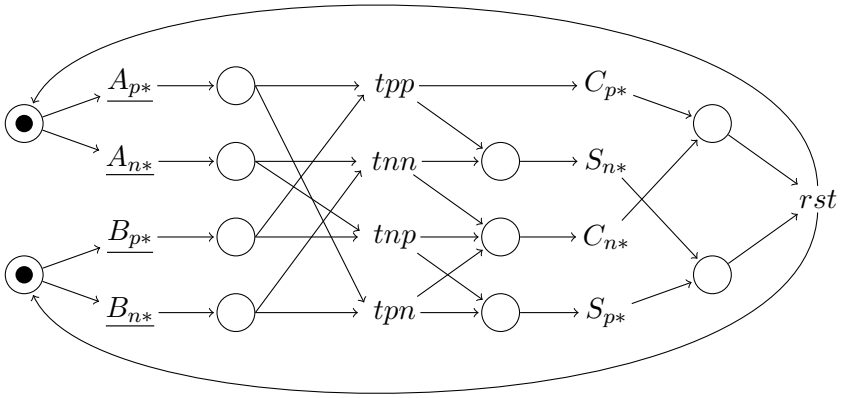


Figure 3.6: The STG description of a half adder

positive transitions and place two tokens in the following column of places. All four transitions prefixed with "t" represent a different combination of input values. After the inputs are valid, one of them will fire and enable the corresponding output signal transitions. Once both output transitions have fired, the rst transition can fire leaving the STG in its initial state.

Here, the problems resulting from using STGs for describing data flow elements can be observed. Firstly, as already discussed in Section 3.2.2, using a purely transition sensitive description method for RTZ data flow functions has disadvantages compared level sensitive methods. In the description of the half adder, this is the reason why the simpler STG with general transitions was chosen. In order to correctly describe the return to *NULL* phase, a considerable amount of additional places, edges and transitions would be needed since only negative transitions of signals that had positive transitions can be enabled. Otherwise, the STG would not be consistent by allowing two or more subsequent firings of a signal's negative or positive transitions. This applies to both, input and output signals. The second problem is that all logic operators need to be modelled as Petri net structures. An *AND*–operator can to be modelled with a synchronisation structure (Figure 2.5b) while an *XOR*–operator can be modelled with conflict (Figure 2.5c). These constructs consume tokens with each use of a term. A simple expression like $(A_p \wedge B_p) \vee (A_n \wedge B_n)$ which needs to be evaluated for the generation of the S_n output therefore requires two synchronisation structures (the transitions tpp and tnn) chained to a conflict (the input place to S_{n*}).

3.2.4 Timed Event/Level Structures

As mentioned in Section 2.5, TEL structures are a hybrid description method combining the advantages of level sensitive (PRS) and transition sensitive (STG) methods. This also becomes apparent in the following examples. Please note that in these examples, a default timing specification $[0, \infty]$ is used in all rules (with some exceptions, see below). This corresponds to the description of delay insensitive circuits where no optimisations based on timing assumptions can be performed.

The description of the arbitration block is depicted in Figure 3.7. It can be observed,

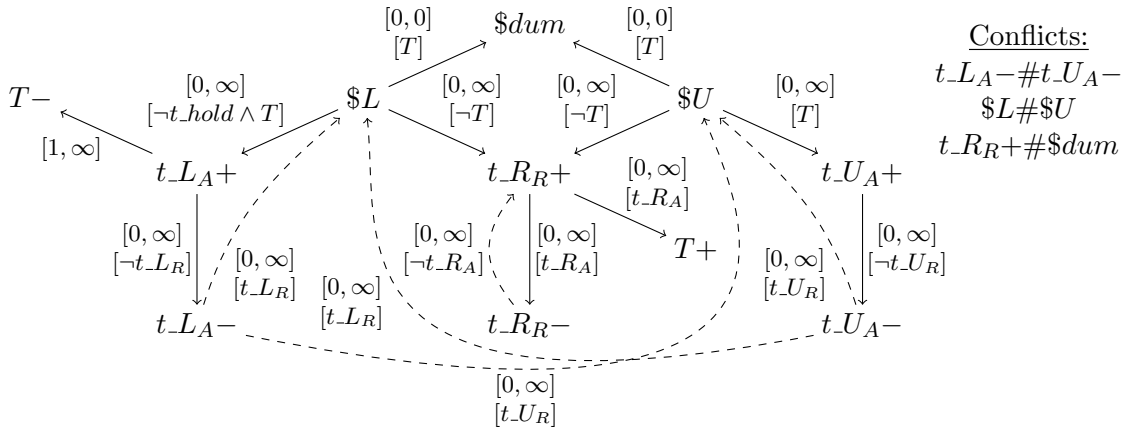
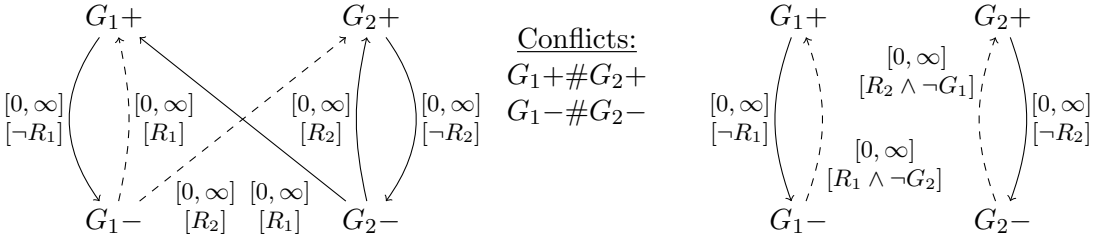


Figure 3.7: The TEL structure of the arbitration block

that compared to the STG description of the same circuit, this graph is considerably simpler, even though the timing specifications make it a little cluttered at first sight. Similarly, the implicit sequencing represented by rules allows the Boolean functions to be considerably simpler than the guards in the PRS description of this block.

Note how the requesting of the token from the right hand neighbour, if necessary, is implemented. In the initial state, as shown in the figure, only the two sequencing events $\$L$ and $\$U$ are enabled. Since they are in conflict, only one can fire and this choice is the result of arbitration. After one of the events has fired, all four of the initially marked rules become disabled, since t_L_A- and t_U_A- are also in conflict. The rule with the synchronisation event that won arbitration as its enabling event and the corresponding acknowledge signal as its enabled event is marked and becomes enabled when the circuit is in possession of the token (signal T). If the token is not present at the circuit, the rule with t_R_R+ as its enabled event that also was marked by the firing of $\$U$ will become enabled (and satisfied, since the lower timing constraint is 0), therefore allowing a right hand request cycle to start. The interesting thing to observe is the event $\$dum$, which serves only the purpose of disabling the rule having t_R_R+ as its enabled event in order to prevent it from firing at a later point when T becomes false. Note that the rules with $\$dum$ as their enabled event have the timing constraint $[0, 0]$ and that the rule with t_L_A+ as its enabling event and $T-$ as its enabled event has the lower timing constraint 1 which allows to conclude that the latter of those rules will not change T until $\$dum$ fires. These constraints ensure the correctness of the model. They must however hold in the resulting circuit (i.e. the disabling of t_R_R+ must be faster than the causal chain t_L_A+, $T-$) for it to be correct. If it was not possible to define time constraints, additional sequencing would have to be introduced to ensure that the $\$dum$ event always fires before $T-$ can fire. The chosen specification is therefore also a good example for optimisations using time constraints.

The description of arbitration is similar to that in STGs, except that the conflict is simply listed in the conflict relation. In contrary to STGs however, if the environment is known not to produce two request at the same time, a mutual exclusion can be modelled



(a) With arbitration (b) Without arbitration

Figure 3.8: TEL structures of mutual exclusion elements

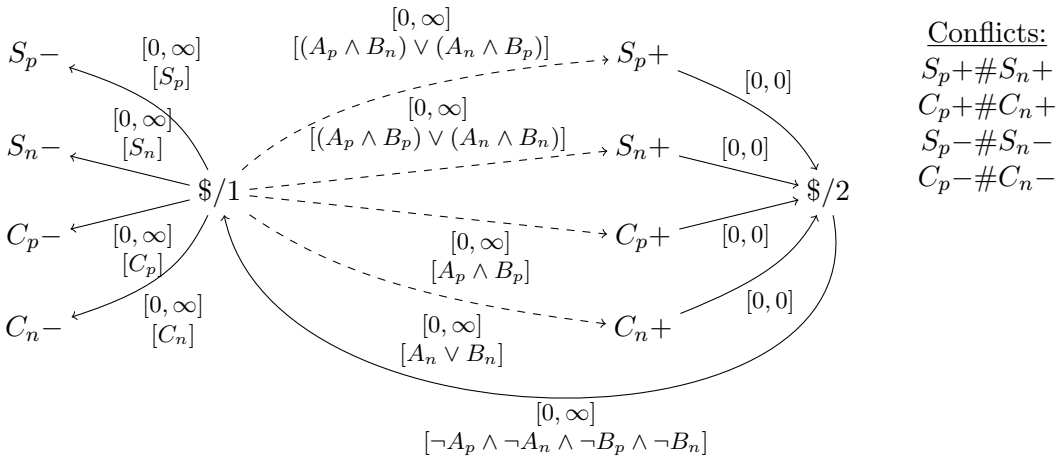


Figure 3.9: The TEL structure of a half adder

without the need of an arbiter, since Boolean functions in rules can also be used for mutual exclusion. For a direct comparison, consider the two descriptions of a mutual exclusion element (with inputs $R_{1,2}$ and corresponding outputs $G_{1,2}$) using RTZ signalling in Figure 3.8.

The description of the half adder depicted in Figure 3.9 also shows the advantages of the hybrid transition/level sensitive description. Even though the description could have been chosen to be equivalent to the PRS in Section 3.2.1, in this example additional sequencing is introduced. All of the initially marked (dashed) rules represent equivalent behaviour to that of PRs assigning a positive value to the output signals. The bottommost rule describes the condition for the back to *NULL* assignments of the output signals. In contrast to the PRS description however, the Boolean functions are not evaluated continuously. The rule with the back to *NULL* function is marked only after both outputs have been assigned a valid value. Even though this behaviour is of little practical importance, the example shows how the structure of the graph influences the behaviour of the described circuit even when all Boolean functions are equivalent to those of the PRS.

Another thing to observe in the TEL structure of the half adder are the Boolean conditions of the leftmost rules that enable the negative transitions of the output signals. These are needed to ensure that events with negative transitions of signals that are already negative cannot fire. This is equivalent to the consistency property of STGs (see Section 2.4).

Since it is known that one of each pair of signal wires will be positive, it is enough to put the pair in conflict to ensure that none of those rules remain marked after the reset to *NULL*. If this was not the case, the marking would have to be removed by the firing of an additional rule, just like it was the case in the description of the arbitration block above, where the sequencing event $\$dum$ was used for this purpose.

3.2.5 Communicating Hardware Processes

CHP is the first of the CSP based description methods used in this chapter and also the first to implement the whole circuit including data flow.

Firstly, the type to be used for data within the circuit is defined to be an 8 bit wide unsigned integer.

```
1  type data_t = {0 .. 255};
```

The next code listing shows the description of the arbitration block. It is a process with one parameter, which defines whether an instantiation of the process should be initially in possession of the token.

```
2  process arbitration(t_init : bool)
3                      (holdtoken, t_U, t_L, t_R)
4  CHP {
5    var token : bool = t_init;
6    *[[ #t_U -> [  token  -> skip
7                [] ~token -> t_R
8                ] ;
9                {t_U; t_U}, token+
10   [:] #t_L -> [  token  -> skip
11               [] ~token -> t_R
12               ] ;
13               {holdtoken ; t_L} , token-
14   ]]
15 }
```

This process is, with the exception of the integration of the token holding capability using the `holdtoken` channel, an exact copy of the *"Distributed Mutual Exclusion on a Ring of Processes"* example from [26] with the syntax from [55].

In the listing, note how each port in the form of a channel hides the subdivision into a request and acknowledge wire and the handshake protocol, data channels even hide the usually dual rail data coding, as will be shown in the processes describing the data path which can be found below. With this abstraction in mind, note the two sequentially executed synchronisations on the `t_U` channel in line 9 which represent requesting access to the shared resource (the network) and freeing it again. To achieve the expected behaviour where freeing of the resource is signalised by the withdrawal of the request, NRZ signalling has to be chosen for the `t_U` port. Specifying handshaking protocols to be used in implementations of described circuits is however not possible in the CHP language and must therefore be offered by the compiler. Furthermore, CHP also does not allow to describe ports to be active or passive.

For the description of the selection of the request sources' port probes, the language construct for arbitrated selection has been used. In case the selection did not require arbitration, the similar construct with `[]` instead of `[:]` could be used. Since CHP does not allow access to variables from concurrent threads, using a Boolean variable to indicate

when the token should not be passed is not possible. Therefore, the *holdtoken* channel is used to control the passing of the token. A synchronisation on this channel must occur before the token is passed to the left hand neighbour. The following listing shows the process *holding*, which describes the other connection point of the *holdtoken* channel.

```
16 process holding()(holdtoken, t_hold)
17 CHP {
18   *[[ holdtoken -> holdtoken
19   []  t_hold    -> t_hold ; t_hold
20   ]
21 }
```

For a successful synchronisation on the *holdtoken* channel, it must be selected in line 18 of the *holding* process. If however the *t_hold* channel is selected, two subsequent synchronisations on the latter have to occur until the control flow returns to the selection, effectively inhibiting the passing of the token between the first and the second synchronisation.

Apart from the arbitration block described above, the rest of the circuit is composed of another four processes, which form the data path. From a high level, functional view, the data channels coming from the two sources are first merged into an internal channel from which, after being buffered in a one–place buffer, they are demultiplexed to one of the outputs based on the address counter in the first byte of each packet. The first two processes, namely *handleU* and *handleR* perform the necessary communication with the arbitration block before the first byte and at the end of each packet. The description of the *handleU* process is shown in the code listing below.

```
22 process handleU()(d_Ui?, d_merge1! : data_t;
23                   t_U, e_Ui, e_merge1)
24 CHP {
25   var end : bool;
26   *[[#d_Ui -> {t_U ; d_merge1!d_Ui?} , end-;
27               *[~end -> [  #d_Ui -> d_merge1!d_Ui?
28                         [] #e_Ui -> e_Ui, e_merge1 , end+
29                         ]
30                ] ; t_U
31   ]]
32 }
```

The core functionality of the process can be found in line 26. After the probe becomes true, which ultimately means there is a request from the attached host pending, network arbitration is performed by synchronising on the *t_U* port before the data are passed to the *d_merge1* port. The rest of the process is composed of a loop that directly passes all further data from *d_Ui* to *d_merge1* until the host synchronises on the *e_Ui* port, which is passed to the *e_merge1* port and ends the loop, and another synchronisation with the *t_U* port after the whole packet has been passed in order to free the token for other service requests.

Apart from having *d_Ri* and *e_Ri* as inputs, *d_merge2* and *e_merge2* as outputs and synchronising on the *t_hold* channel instead of *t_U* at the beginning and the end of an incoming packet, the process *handleR* is equivalent to the *handleU* process.

```
33 process handleR()(d_Ri?, d_merge2! : data_t;
34                   t_hold, e_Ri, e_merge2)
35 CHP {
36   *[[#d_Ri -> {t_hold ; d_merge2!d_Ri?} , end-;
```

```
37                *[~end -> [  #d_Ri -> d_merge2!d_Ri?
38                          [] #e_Ri -> e_Ri, e_merge2 , end+
39                          ]
40                ] ; t_hold
41    ]]
42  }
```

Since all communication over channels in CHP must be point–to–point and only sequentially executed code can make use of implicit merging, a separate process is used to merge the two of each, *d_merge* and *e_merge* channels into one. It is called *merge* and listed below. Note that this merge is performed without arbitration, since the communication with the arbitration block in the *handleU* and *handleR* processes ensures that only one channel at a time will supply data to be merged.

```
43  process merge()(d_merge1?, d_merge2?, d_merged! : data_t;
44                  e_merge1, e_merge2, e_merged)
45  CHP {
46    *[[ #d_merge1 -> d_merged!d_merge1?
47    []  #d_merge2 -> d_merged!d_merge2?
48    ]] ,
49    *[[ #e_merge1 -> e_merged, e_merge1
50    []  #e_merge2 -> e_merged, e_merge2
51    ]]
52  }
```

The last part of the data path is the *demux* process listed below, which determines the destination of the packet from the address byte, decrements it if necessary and then routes the packet to the correct destination.

```
53  process demux()(d_merged?, d_Uo!, d_Lo! : data_t;
54                  e_merged, e_Uo, e_Lo)
55  CHP {
56    var sink : data_t;
57    var buffer : data_t;
58    var up : bool;
59    var end : bool = false;
60    *[[  #{d_merged : d_merged =  0} -> up+ , d_merged?sink
61      [] #{d_merged : d_merged != 0} -> d_merged?buffer ;
62                                        up- , d_Lo!(buffer - 1)
63      ] ;
64      *[ ~end ->
65        [  #d_merged -> d_merged?buffer ;
66                        [  up  -> d_Uo!buffer
67                        [] ~up -> d_Lo!buffer
68                        ] , end-
69        [] #e_merged -> [  up  -> e_Uo
70                        [] ~up -> e_Lo
71                        ] , e_merged , end+ ;
72        ]
73      ]
74    ]]
75  }
```

The variable *up* is used to determine the destination of a packet and it contains the result of the comparison of the address byte to zero. The *buffer* variable describes the storage used to buffer the incoming stream of data. The variable *sink* in line 56 does not serve any special purpose and is only used as a data sink meant to be optimised away by a

tool processing the description. It had to be introduced in order to complete the handshake on the *d_merged* channel after it has been probed without using the data, since CHP does not allow to perform handshaking on input ports without storing the communicated value.

Please note that the decrement of the address byte is performed at the output operation in line 62, therefore after its buffering. The reason for this is that CHP does not allow manipulation of data prior to its assigning to a variable. The value probe, also used in the selection in lines 60 and 61, allows to compute a Boolean expression using the probed data. However, it does not provide any possibility to store or send results of arithmetic operations.

To complete the description, a *meta* process has to be introduced that does not describe any of the circuit's behaviour but instead instantiates all of the above listed processes, sets their parameters where available, and specifies their interconnection.

```
76  process nic()(d_Ri?, d_Ui?, d_Uo!, d_Lo! : data_t;
77                e_Ri, e_Ui, e_Uo, e_Lo, t_R, t_L)
78  META {
79    instance arb: arbitration;
80    instance hdg: holding;
81    instance hlU: handleU;
82    instance hlR: handleR;
83    instance dmx: demux;
84    instance mer: merge;
85    arb(false);
86    connect arb.t_U, hlU.t_U;
87    connect arb.holdtoken, hdg.holdtoken;
88    connect hdg.t_hold, hlR.t_hold;
89    connect hlU.d_merge1, mer.d_merge1;
90    connect hlR.d_merge2, mer.d_merge2;
91    connect mer.d_merged, dmx.d_merged;
92    connect hlU.e_merge1, mer.e_merge1;
93    connect hlR.e_merge2, mer.e_merge2;
94    connect mer.e_merged, dmx.e_merged;
95    connect hlR.d_Ri, d_Ri;
96    connect hlU.d_Ui, d_Ui;
97    connect hlR.e_Ri, e_Ri;
98    connect hlU.e_Ui, e_Ui;
99    connect dmx.d_Uo, d_Uo;
100   connect dmx.d_Lo, d_Lo;
101   connect dmx.e_Uo, e_Uo;
102   connect dmx.e_Lo, e_Lo;
103   connect arb.t_L, t_L;
104   connect arb.t_R, t_R;
105 }
```

3.2.6 Haste

The first listing of the description using Haste shows the definition of the type used for the data path, which is an unsigned byte, and the beginning of the declaration of the top level process *nic*, which embodies all other processes that describe the circuit.

```
1  data_t=type [0..255]
2  &
3  nic: main proc(d_Ri, d_Ui ? chan data_t pas
4               & d_Lo, d_Uo ! chan data_t act
```

```
5               & t_L, e_Ri, e_Ui ? chan ~ pas
6               & t_R, e_Lo, e_Uo ! chan ~ act).
7  begin
8    t_hold: var bool narb! := false
9  & up: var bool narb! narb:
10 & d_merged: chan data_t narb!
11 & e_merged, t_U: chan ~ narb!
```

In the specification of the external ports in the header of the *nic* process, it can be observed how the input and output ports are described as passive and active respectively. The two variables as well as the internal channels that are declared within the scope of the *nic* process are local to that process and can also be accessed from nested scopes. The processes declared within the *nic* process, that will be described in the following, can thus access these resources. Note the various **narb** specifiers in the declarations which tell the compiler that no read or write conflicts can occur which would require arbitration to be resolved.

The next listing shows the description of the arbitration block, which is a process declaration in the scope of the *nic* process.

```
12 & arbitration: proc().
13   begin
14     token: var bool narb! := false
15   & gettoken: narb proc().
16       if -token then t_R!~ fi
17   |
18     forever do
19       sel outprobe(t_U) then gettoken(); (t_U?~ ; t_U?~) || token := true
20       or  outprobe(t_L) then gettoken(); (wait(-t_hold) ; t_L?~) || token := false
21       les
22     od
23   end
```

It can be seen that the *arbitration* process is described, apart from a few exceptions, very similarly as in CHP. One of the exceptions is the introduction of the *gettoken* process, which makes use of the elaborate method of describing hardware sharing in Haste. It performs the task of requesting the token by synchronising on the *t_R* port if the circuit is not in the possession thereof, a task that has to be performed irrespective of which request is being served. Also note that the token holding functionality is provided by the shared Boolean variable *t_hold* which, when set to true, inhibits the passing of the token. The last thing to note is that in the header of the *arbitration* process, there are no ports. This is because all communication directly uses channels that are local to the *nic* process.

The arbitrated selection of the request sources is described with the **sel** statement. There is however no similar statement to describe selection that does not require arbitration. Although the **if** statement could be used for this purpose after waiting for one of the probes to become true, the Haste manual discourages the use of probes expressions in other than select statements.

The *handle* process, which is listed below, also uses hardware sharing. In the CHP description, the process was split into two, mostly equivalent, yet independent processes. To share the common parts to save hardware, additional multiplexing would have to be introduced. In Haste, the multiplexing needed for accessing shared resources is introduced implicitly.

```
24 & handle: proc().
25   begin
26     sharedproc: narb proc(d_ch ? chan data_t
27                           & e_ch ? chan ~).
28     begin
29       endloop: var bool := false
30     |
31       if dataprobe(d_ch) = 0 then up := true
32         else up := false ; d_merged!(dataprobe(d_ch) -1) fit data_t
33       fi ;
34       d_ch?~ ;
35       repeat
36         sel outprobe(d_ch) then d_merged!dataprobe(d_ch) ; d_ch?~
37         or  outprobe(e_ch) then endloop := true || e_merged!~ || e_ch?~
38         les
39       until endloop
40     end
41   |
42     forever do
43       wait(outprobe(d_Ui)) ;
44       t_U!~ ;
45       sharedproc(d_Ui, e_Ui) ;
46       t_U!~
47     od ||
48     forever do
49       wait(outprobe(d_Ri)) ;
50       t_hold := true ;
51       sharedproc(d_Ri, e_Ri) ;
52       t_hold := false
53     od
54   end
```

The *sharedproc* process, declared in the beginning of the *handle* process describes most of the task associated with data. Firstly the address byte is compared to zero in line 31, after which the variable *up* is set accordingly. The address byte is decremented and passed if necessary and the handshake on the *d_ch* is completed in line 34. After that, all incoming data are passed to the *d_merged* until a synchronisation on the *e_ch* port occurs which, after being forwarded, causes the process to terminate.

In the body of the *handle* process, there are two concurrent loops in which the necessary communication with the arbitration block is performed before the *sharedproc* process is accessed to carry out the common task. Here, note how within the *sharedproc* process only the *d_ch* and *e_ch* channels defined in its header are read. These are assigned one of the set of the external channels *d_Ui* and *e_Ui* or *d_Ri* and *e_Ri* when *sharedproc* is accessed in lines 45 and 51 respectively.

The *demux* process listed below is responsible for the distribution of the data packets including the *end* handshakes to the output selected by *up*. Also the one place buffer is part of this process. Compared to the *demux* process in CHP, this one is considerably simpler due to the fact that the *d_merged* and *e_merged* channels are only written after the *up* variable has been set to the correct value as opposed to the CHP description, where also the address byte was passed unprocessed.

```
55 & demux: proc().
56   begin
57     buffer: var data_t
```

```
58    |
59      forever do
60        d_merged?buffer ;
61        if up then d_Uo!buffer
62        else d_Lo!buffer
63        fi
64      od ||
65      forever do
66        wait(outprobe(e_merged) ;
67        if up then e_Uo!~
68        else e_Lo!~
69        fi ;
70        e_merged?~
71      od
72    end
```

The description of the *demux* process is relatively straight forward. In the first of the two concurrent loops, the incoming data is read into the buffer and subsequently output to the chosen destination. In the second loop, the *e_merged* channel is first probed and the handshake is completed only after the synchronisation on the chosen external port to prevent a race condition that could result to the variable *up* being changed before or during its reading for the comparison.

To complete the description of the circuit, the the listing below shows the body of the top level *nic* process where the processes described above are all run in parallel. As mentioned earlier, no connections have to be specified since the processes communicate over channels and variables shared within the *nic* process, and not over ports local to each process that would need external connection.

```
73  | arbitration() ||
74    handle() ||
75    demux()
76  end
```

3.2.7 Balsa

As with the previous design, the description of the circuit with Balsa starts with the definition of the desired type for the data path and the declaration of the top level procedure, *nic* with its external ports. Furthermore, the local channels, which will be used for communication between the procedures the circuit is made of, are declared.

```
1  type data_t is 8 bits
2
3  procedure nic(input d_Ri, d_Ui : data_t;
4                output d_Lo, d_Uo : data_t;
5                sync t_R, t_L, e_Ri, e_Lo, e_Ui, e_Uo) is
6  local
7    channel d_merge1, d_merge2, d_merged : data_t
8    sync e_merge1, e_merge2, e_merged, t_U, t_hold
```

Balsa, same as CHP, does not allow to define the ports to be active or passive in the procedure header. However, all ports are by default active and only if the **select** or **arbitrate** statement is used with a channel, it becomes passive, giving the designer an indirect way to determine this property of ports within the code.

The next listing shows the description of the arbitration block. In line 13, the shared procedure *gettoken* is declared which has the same functionality as in the Haste description in the previous chapter. In line 18, the explicit initialisation of the variable *token* is performed, since Balsa does not allow to set initial values for declared variables. Apart from using the shared *gettoken* procedure, the *arbitration* procedure is equal to the concurrent composition of the CHP processes *arbitration* and *holding*, which had to be separated in CHP because no local channels were allowed. The **arbitrate** statement describes the arbitration that is required to resolve the concurrent requests. Where no arbitration is needed, the **select** statement, which has equivalent syntax, can be used.

Note that in contrast to CHP and Haste, the handshake of a selected, probed channel is implicitly completed at the end of the associated command, within which the channel behaves as a variable. This requires the introduction of variables to indicate which channel was selected if the behaviour after completing the (first) handshake of the latter depends on the selection. This is also the case in the arbitration block, where two subsequent synchronisations on the *t_U* channel have to be performed after it has been selected. In this case, the variable *token*, which is dependent on the selection is used in line 25 to perform the second synchronisation on the *t_U* channel. Furthermore, it can also be observed in the loop starting at line 27 which has the functionality of the *holding* process, where the variable *holdvar* had to be introduced to be able to perform two synchronisations after the selection.

```
9   procedure arbitration is
10  local
11    variable token, holdvar : bit
12    sync holdtoken
13    shared gettoken is
14    begin
15      if not token then sync t_R end
16    end
17  begin
18    token := 0 ;
19    loop
20      arbitrate
21        t_U then gettoken() ; token := 1
22      | t_L then gettoken() ;
23                 sync holdtoken || token := 0
24      end ;
25      if token then sync t_U end
26    end ||
27    loop
28      select
29        holdtoken then continue
30      | t_hold then holdvar := 1
31      end ;
32      if holdvar then
33        sync t_hold || holdvar := 0
34      end
35    end
36  end
```

The next listing shows the description of the *handleU* procedure which is equivalent to the CHP process *handleU* (see Section 3.2.5).

```
37  procedure handleU is
```

```
38      variable endloop : bit
39    begin
40      loop
41        select
42          d_Ui then sync t_U ;
43                    d_merge1 <- d_Ui
44        end ||
45        endloop :=  0 ;
46        loop while not endloop then
47          select
48            d_Ui then d_merge1 <- d_Ui
49          | e_Ui then sync e_merge1 || endloop := 1
50          end
51        end ;
52        sync t_U
53      end
54    end
```

The procedure *handleR* (not listed) is, as it was in CHP, very similar to the *handleU* procedure with *d_Ui*, *e_Ui*, *d_merge1* and *t_U* replaced by *d_Ri*, *e_Ri*, *d_merge2* and *t_hold*, respectively.

The procedure *merge* which merges the data and the synchronisation channels for the packet ending is listed below.

```
55    procedure merge is
56    begin
57      loop
58        select
59          d_merge1 then d_merged <- d_merge1
60        | d_merge2 then d_merged <- d_merge2
61        end
62      end ||
63      loop
64        select
65          e_merge1 then sync e_merged
66        | e_merge2 then sync e_merged
67        end
68      end
69    end
```

The demux procedure, which is also equivalent to the CHP process *demux* with two exceptions is listed below. The first exception is the introduction of the *send* variable which is necessary to overcome the limitation of the selection in Balsa. This also eliminates the need of a variable which terminates the inner loop, since the variable *send* can be used for this purpose (line 98). The second and more distinctive difference is that since selected channels behave like variables within the selection block, the decrement of the address byte (line 78) can be performed with the probed channel prior to buffering.

```
70    procedure demux is
71      variable buffer : data_t
72      variable send : bit
73      variable up : bit
74    begin
75      loop
76        select d_merged then
77          if d_merged = 0 then up := 1 || send := 0
```

```
78          else up := 0 || buffer := (d_merged - 1 as data_t) || send := 1
79          end
80        end ;
81        if send then d_Lo <- buffer end ;
82        loop
83          select
84            d_merged then
85              send := 1 ||
86              buffer := d_merged
87          | e_merged then
88              send := 0 ||
89              if up then sync e_Uo
90                    else sync e_Lo
91              end
92          end ;
93          if send then
94            if up then d_Uo <- buffer
95                  else d_Lo <- buffer
96            end
97          end
98        while send end
99      end
100   end
```

The last code listing completes the declaration of the top level *nic* procedure by running all, internally declared processes in parallel. Note that, same as in Haste, there are no connections necessary.

```
102   arbitration() ||
103   handleU() ||
104   handleR() ||
105   merge() ||
106   demux()
107 end
```

Chapter 4

Comparison

This chapter summarises the information about properties of the considered description methods from the previous chapters and compares the latter directly. The comparison is based on various criteria showing the capabilities of the description methods. Each criterion or group thereof addressing different aspects of asynchronous circuit design is presented in a separate section.

4.1 Arbitration

When describing the arbitration block of the exemplary circuit in Chapter 3, the description of arbitrated and non-arbitrated selection was discussed for each description method.

Table 4.1 shows a summary of the use of arbitration in the description methods. In the table, *no* denotes that the method does not allow to describe circuits requiring the analog arbiter, *implicit* means that by definition, every conflict is arbitrated while methods with the entry *selectable* offer possibilities to describe both, arbitrated and non-arbitrated selection. For details, please consult Chapter 3.

Desc. method	Arbitration
PRS	No
AFSM	No
STG	Implicit
TEL structures	Selectable
CHP	Selectable
Haste	Selectable[1]
Balsa	Selectable

Table 4.1: Summary of description capabilities for arbitration

The inability to describe arbitration, as with PRSs and AFSMs, causes external arbiters to be required for every non-deterministic choice (see also Section 3.2.2). Each group of input signals that are in conflict thus have to be routed through an arbiter after circuit

[1] Even though there is only an arbitrated selection statement, there are other possibilities to describe non-arbitrated selection.

synthesis. Where internal signals are in conflict with others, they must be output from the described circuit and again input after the external arbitration. In STGs, conflicts that require arbitration can also be modelled as an interface to an external arbiter. The synthesis tool could then be set to exclude arbitration from STG conflicts. Arbiters described with another description method or imported from a design library could then be connected to the dedicated interfaces.

4.2 Concurrency and Sequence

As no implicit synchronisation takes place in asynchronous circuits, every event is concurrent to all others that it is not explicitly synchronised with. Therefore a description method should provide constructs for both concurrent and sequential composition of events.

Table 4.2 shows in a summary whether concurrent and/or sequential composition are supported in the considered description methods. In the table, *yes* denotes that a method provides constructs to simply describe concurrently or sequentially composed events, while the other three options are used with methods that do not offer such constructs. *Implicit* indicates which kind of composition these methods inherently use, *no* means that it is not possible to compose events in that particular way and *manual* means that even though no construct is provided for such composition, it is possible to describe it manually by introducing additional variables and states.

Desc. method	Concurrency	Sequence
PRS	Implicit	Manual
AFSM	No	Implicit
STG	Yes	Yes
TEL structures	Yes	Yes
CHP	Yes	Yes
Haste	Yes	Yes
Balsa	Yes	Yes

Table 4.2: Summary of description capabilities for concurrency and sequence

The lack of sequential composition as in PRSs does not limit the expressiveness of the description method. The lack of concurrent composition as in AFSMs, however, does impose significant limitations. Every part of a circuit with concurrent operation must either be serialised, or described with a dedicated AFSM with each synchronisation using handshakes on dedicated channels.

4.3 Timing

Some description methods require certain timing assumptions about the environment of the circuit being described to hold for correct operation. These can decrease the possibilities for optimisation or even disallow the description of arbitration.

Table 4.3 shows a summary of timing assumptions made by the considered description methods. Note that the assumption that signals must adhere to the specified protocol (which excludes faulty behaviour such as glitches) is not included in the table.

Desc. method	Timing assumption
PRS	Stability of guards
AFSM	(Restricted) Fundamental mode
STG	Persistence
TEL structures	Specifiable
CHP	None
Haste	None
Balsa	None

Table 4.3: Summary of timing assumptions

As known from synchronous circuits, a strong timing assumption greatly simplifies the design process, verification and even the circuit implementation. However, if the assumption is too strong, it may cause considerable problems such as the area, power and delay overheads as in synchronous systems. On the other hand, since it is unrealistic for wires and/or gates to have arbitrary delay, from zero to near infinity, designing circuits with the minimal timing assumption of isochronic forks in the implementation can also cause area, power and delay penalties. Only TEL structures allow to trade timing related optimisations for decreased reliability at the designer's will.

4.4 Asynchronous communication

There are three important parameters for describing an asynchronous communication channel — the handshaking protocol (two phase or four phase), which connection point is active and which passive and the signalling scheme for the data, naturally, except for pure synchronisation channels. As each of the different combinations of these parameters has advantages and disadvantages, it might beneficial to be able to use more than one setting for all the channels within a circuit. Table 4.4 shows which options are available in the considered description methods.

Note that the physical implementation of communication channels in CSP based methods is not specifiable. In practise, however, this choice would be possible within the design environment.

Desc. method	Handshaking	Active/Passive	Signalling
PRS	All	Yes	Delay insensitive[2]
AFSM	All	Yes	Delay insensitive[2]
STG	All	Yes	Delay insensitive[2]
TEL structures	All	Yes	All
CHP	Not specifiable	No	Not specifiable
Haste	Not specifiable	Yes	Not specifiable
Balsa	Not specifiable	Indirect[3]	Not specifiable

Table 4.4: Summary of description capabilities for arbitration

The inability to describe (complete) bundled data circuits with the first three description methods does not have a negative effect on their practicality. These low abstraction level methods would anyway not be used by a designer to describe the data path of a circuit. Instead, the control circuit would and can be described with these methods and appropriate delays determined from the data path's implementation would be added in a post-synthesis step. The inability of choosing a handshaking protocol and a signalling scheme for data in the CSP-based descriptions does also not limit the designer in practise, since most synthesis tools would offer these choices in the design environment. The inability of choosing external ports to be active or passive in CHP, however, does impose a limit since supplementary circuits would be required for the connection of two active or two passive ports.

4.5 Level/Event Sensitivity

While some description methods are level sensitive and others event sensitive, their expressiveness is not limited by this choice. Describing the probing of a signal level in a purely event sensitive method can be done by using the positive and negative transitions of the signal and changing the state of the circuit to represent which transition occurred last. Similarly, detecting events in a purely level sensitive description method can be done by saving the last observed level of the signal and comparing the currently sensed level with it.

However, certain circuits can more conveniently be described with a level sensitive description method while the opposite is true for others. Table 4.5 shows what sensitivity can be described with the considered methods.

For the first four description methods, using separate signals for a circuit description, the distinction of level and event sensitivity is relatively clear. The CSP-based methods inherently use events for all communication and are therefore considered event sensitive. It could be argued that the probing construct present in all three methods allows level sensitive descriptions, however, since probes only indicate pending events and only the events can be used for communication, they are not considered as true level sensitive constructs in this work.

Desc. method	Sensitivity
PRS	Level
AFSM	Event
STG	Event
TEL structures	Event/Level
CHP	Event[4]
Haste	Event/Level[5]
Balsa	Event

Table 4.5: Summary of sensitivity to events or levels

[2]The use of single rail (bundled data) signalling requires timing constraints which are not specifiable.
[3]See Section 2.6.5, page 42

As stated above, the difference between a level and an event sensitive description method lies within the comfort of the description of circuits using four phase (level sensitive) and two phase (event sensitive) handshake protocols. For CHP and Balsa, however, pure event sensitivity means that only channels with implicit handshaking can be used for communication. This disallows the description of single wire signals because the minimal communication medium offered is a synchronisation channel with a request and an acknowledge wire.

4.6 Modularity and Parametrisation

The use of modularity is a key technique for an effective description of large-scale circuits. Often, modules described with different levels of abstraction are composed to form bigger functional units which may again be composed to form a circuit. In this comparison, however, a description method is considered to support modularity, if it offers capabilities to compose multiple sub-circuits described with the same method into a bigger circuit.

The possibility of using parameters in descriptions allows more general modules to be described, which then can be altered to a limited extend during instantiation. This extends the applicability of predefined modules and thus also increases the importance of design libraries. A typical example of a parametrised module is a FIFO buffer of variable length and width.

Table 4.6 shows a summary of inherent modularity and parametrisation support in the description methods.

Desc. method	Modularity	Parametrisation
PRS	No	No
AFSM	No	No
STG	No	No
TEL structures	No	No
CHP	Yes	Yes
Haste	Yes	Preprocessor
Balsa	Yes	Yes

Table 4.6: Summary of description capabilities for arbitration

As stated above, the lack of modularity in a description method does not prevent the designer from reusing circuits after synthesis. It is the comfort of reusing even small building blocks within the description that is missing. Parametrisation merely extends the applicability of modules and thus further increases the comfort.

[4]Based on [55], there is a *wired* type which might allow direct (level sensitive) access to wires in CHP, however, it is not clear from the manual how it is used and what functionality it allows.

[5]Level sensitive due to the non-handshaked signal support in Haste, see Section 2.6.4, page 40.

4.7 Level of Abstraction

The level of abstraction of a description method can substantially simplify the description of a circuit. On the other hand, hiding details of the implementation of various constructs inevitably causes a group of circuits to become indescribable.

Table 4.7 shows the most important abstractions for each description method. A brief discussion follows. Note that the abstraction of time is not considered in this section, since timing was compared in Section 4.3.

Desc. method	Abstraction
PRS	State holding
AFSM	Sequencing
STG	Sequencing, Conflicts
TEL structures	Sequencing, Conflicts
CHP	Channels, Types, Arithmetics
Haste	Channels, Types, Arithmetics, Multiplexing
Balsa	Channels, Types, Arithmetics

Table 4.7: Summary of the levels of abstracion

PRSs describe circuits at a level close to netlists in the form of operators. The most notable abstraction in PRSs is the description of state holding operators by allowing states with no value assigned to the output signal. This causes the implementation style of state holding operators, i.e. static or dynamic, to be unavailable to the designer.

In AFSMs, the most important abstraction is the sequencing of state transitions, which includes and at the same time hides the next-state function and state coding.

STGs, being more general than AFSMs, abstract not only sequencing of events and the state holding necessary for that, but also other constructs resulting from concurrent descriptions. Most notably, this includes conflicts, where the choice of implementation with or without an arbiter is hidden from the designer.

TEL structures are very similar to STGs regarding level of abstraction. The main difference is that due to additional Boolean guards in each rule, the abstraction of conflicts does not disallow to describe a conflict without the need of arbitration in the implementation.

The most important abstractions in CSP based description methods include implicitly synchronised communication channels, multiple bit wide data types and arithmetics. Haste additionally abstracts shared access to channels and variables and the multiplexing and mutual exclusion required for this purpose. Communication channels hide details about the used handshake protocol, data coding schemes and initial states of the channel wires. The atomic handling of multiple bit-wide data most notably hides synchronisation related circuits like completion detection. The description of arithmetic operations with simple operators within expressions hides complete implementations of circuits like adders and multipliers.

4.8 Summary

The summary of the comparison with all the criteria is shown in Table 4.8. From the observations made during the description of the exemplary circuit in Chapter 3 and the comparisons in this chapter, it can be seen, that each description method has its advantages for the description of some types of circuits, while it is unsuitable for or even precludes the description of other types. Generally, those description methods working with individual signals are only useful for very simple circuits or ones that do not handle data, such as the control flow of larger circuits. With their high abstraction level CSP based description methods on the other hand are most useful for the description of data processing circuits, since complex processing blocks can be described with concise procedures.

PRSs are not very practical as a design entry description method. Their primary application is the representation of intermediate synthesis results from a higher level description method. However, given that PRSs allow the description of operators at a very low level, they can be used to describe templates of operator implementations to be used by the tool performing the synthesis from a higher level description.

AFSMs can be used for simple circuit descriptions, but the incapability to express concurrency makes them less suitable for the description of the control flow of larger circuits. Typically, they would also be generated automatically from a higher level description.

STGs have the very useful property of being very closely related to Petri nets. This gives a designer the possibility to simply refine a high level Petri net model, which has been used for various analyses and optimisations, into a circuit with the same properties. This is most useful for the description of the control flow of complex data-processing circuits. The only disadvantage for this application is the lack of differentiation between arbitrated and non-arbitrated selection.

TEL structures have the same application domain as the above mentioned methods, however, due to the possibility to describe temporal behaviour, they can also be used for timing related optimisations. Moreover, since both, conflicts as in STGs and guards as in PRSs are available in TEL structures, it is the only of the four low abstraction level methods that allows the description of both, arbitrated and non-arbitrated selection, which is important for the description of control flows of complex circuits.

The three CSP based description methods are suitable for the design entry of large scale circuits. Synthesis tools can then be used to directly generate netlists or to generate a description of the same circuit with a less abstract method, allowing the designer to refine the low-level behaviour. Since these three methods are substantially different from the other considered methods, it is more interesting to observe the differences between the former, rather than between the two groups of methods.

CHP is characterised by a very concise syntax, using special characters rather than words. Balsa is functionally almost equal to CHP and the most notable distinction is a very verbose syntax. Compared to CHP and Balsa, Haste has more advanced probing constructs and special constructs for handling non-handshake signals, allowing it to describe circuits that would not be describable with the former two methods. Additionally, it allows to access shared resources from concurrent statements which considerably simplifies a description. One negative property of Haste is the limited support for parametrisation due to the simple preprocessor used.

Desc. method	Arbitration	Timing assumption	Signalling	Parametrisation
	Concurrency	Handshaking	Sensitivity	Abstraction
	Sequence	Active/Passive	Modularity	
PRS	No	Stability of guards	Delay insensitive	No
	Implicit	All	Level	State holding
	Manual	Yes	No	
AFSM	No	(Restricted) Fundamental mode	Delay insensitive	No
	No	All	Event	Sequencing
	Implicit	Yes	No	
STG	Implicit	Persistence	Delay insensitive	No
	Yes	All	Event	Sequencing, Conflicts
	Yes	Yes	No	
TEL structures	Selectable	Specifiable	All	No
	Yes	All	Event/Level	Sequencing, Conflicts
	Yes	Yes	No	
CHP	Selectable	None	Not specifiable	Yes
	Yes	Not specifiable	Event	Channels, Types, Arithmetics
	Yes	No	Yes	
Haste	Selectable	None	Not specifiable	Preprocessor
	Yes	Not specifiable	Event/Level	Channels, Types, Arithmetics, Multiplexing
	Yes	Yes	Yes	
Balsa	Selectable	None	Not specifiable	Yes
	Yes	Not specifiable	Event	Channels, Types, Arithmetics
	Yes	Indirect	Yes	

Table 4.8: Summary of the comparison

Chapter 5

Exemplary Design

In this chapter, all presented methods will be used to describe an applicable example related to the design of a processor. In the first part, the considered processor will be introduced. The second part shows the exemplary descriptions.

The motivation for this chapter is, that in the previous examples, the description methods have been analysed and compared based on a predefined circuit. Even though the circuit was carefully chosen to show the properties of a wide spectrum of description methods, they were not used in their intended application. This chapter will show some examples that present each description method from a different point of view — what was it designed to be used for?

5.1 The Processor

In order to visualise how the chosen examples can be relevant in a useful design process, in contrast to examples of academic significance only, all of the chosen circuits can be employed in the design of an asynchronous processor. Therefore, the (synchronous) *SPEAR2, Scalable Processor for Embedded Applications in Real-time Environments* soft core processor, developed at the Embedded Computing Systems Group of the Vienna University of Technology, has been used as a source of exemplary sub-circuits.

The SPEAR2 processor [63] is a four stage pipelined RISC architecture processor that was designed with the goal of high predictability of application execution times, in order to be employable in real-time environments. It uses Harvard memory architecture, thus instruction and data memories are separated. Moreover, both, a ROM and a RAM are mapped to the instruction memory space, the former to contain boot code and the latter to save application code accessed by the boot loader through one of the peripherals. The bit-width of data words is configurable to either 16 or 32 bits. Instructions constantly use 16 bit wide words. There are no caches or speculative load or execution systems, as these would worsen the worst case performance and rise the complexity of execution timing analysis, both of which are important in real-time systems. There is a forwarding unit to prevent long pipeline stalls because of data hazards, when the same register is written and read in subsequent instructions. The data memory is accessible byte-wise to simplify memory operations. The processor recognises 122 instructions some of which can contain conditional execution flags. The register file contains 16 registers, 14 of which are general purpose.

The processor is built modularly — the core contains only essential circuits like the program counter, the instruction decoder, the ALU, the forwarding unit and the memory access unit. The rest of the processor, even status flags and frame pointer registers, are external modules of the processor. This gives the processor a great implementation variability with a simple addition of custom modules. In a later addition, also an AMBA bus interface was added to the processor as a collection of extension modules [64].

5.2 The Examples

This part presents the various examples, one for each description method. The chosen circuits are parts of the SPEAR2 processor, however, they are altered to fit in the asynchronous design style where necessary. These modifications will be adverted to. Nevertheless, the examples only show parts of the description of an asynchronous equivalent to the processor with the same structure and communication channels as in the original synchronous description. Circuits converted in such a way can achieve a similar performance and size to the original synchronous circuits [65]. However, a redesign of the processor to utilise the advantages of asynchronous circuits to a bigger extend could lead to a notable improvement of performance, power and area.

5.2.1 Production Rule Sets

The typical application of PRSs is the representation of intermediate results of circuit syntheses as in [25] or the description of implementation templates to be used by the synthesis tools. The closeness of PRS to netlists allows them to be used for low level circuit optimisations. As an example, [10] introduces the *precharge half-buffer* (PCHB), a template for the implementation of functions to be used within pipelines using four phase handshaking protocols. PCHBs address the problem of long transistor chains of P-FETs required for the implementation of the neutrality test. The neutrality test indicates whether all input variables are in the neutral *NULL* state, which is required for the back-to-*NULL* phase in NCL.

The chosen example is motivated by the PCHB and has a similar implementation style. The circuit is a binary 2-to-1 multiplexer using NCL coded inputs and control signal, as it would be used in an asynchronous implementation of the barrel shifter within the ALU of the processor. The typical implementation with two state holding operators is as follows:

$$\begin{aligned} a_0 \wedge c_0 \vee b_0 \wedge c_1 &\mapsto x_0 \uparrow \\ a_1 \wedge c_0 \vee b_1 \wedge c_1 &\mapsto x_1 \uparrow \\ \neg a_0 \wedge \neg a_1 \wedge \neg b_0 \wedge \neg b_1 \wedge \neg c_0 \wedge \neg c_1 &\mapsto x_0 \downarrow, x_1 \downarrow \end{aligned}$$

with a and b being data inputs, c being the control signal and x the output. Each signal has two wires indexed 0 and 1 due to the NCL coding.

When these multiplexers are interconnected to form a barrel shifter, i.e. 5 stages for the 32 bit data path, in the back-to-*NULL* phase, the neutral data values must propagate through all the multiplexers. This creates an inefficient long logic path with long chains of P-FETs driving the intermediate signals. As a proposed optimisation, consider the following implementation of the same multiplexer, inspired by the PCHB.

$$
\begin{aligned}
en \wedge (a_0 \wedge c_0 \vee b_0 \wedge c_1) &\mapsto x_0 \uparrow \\
en \wedge (a_1 \wedge c_0 \vee b_1 \wedge c_1) &\mapsto x_1 \uparrow \\
\neg en &\mapsto x_0 \downarrow, x_1 \downarrow \\
\neg x_0 \wedge \neg x_1 &\mapsto ne \uparrow \\
x_0 \vee x_1 &\mapsto ne \downarrow
\end{aligned}
$$

There are two additional signals introduced. The *en* signal enables the evaluation of the inputs to generate the outputs when being high, and resets the outputs to the neutral *NULL* state when being low. The *ne* signal indicates the *NULL* state of the outputs when high.

Using this implementation, resetting the multiplexers to *NULL* can be performed in parallel by pulling down the *en* signal when the data input to the barrel shifter change to *NULL*. The *ne* signals from the various stages can be combined in a single gate or a tree structure to the signal indicating that the full circuit has reset to the *NULL*, as required for delay insensitive implementations.

5.2.2 Asynchronous Finite State Machines

The use of AFSMs can be compared to the use of standard FSMs in synchronous design. Typical applications are simple control circuits. This, however, only applies to circuits described directly with the graphical or tabular state machine description. When used to represent the result of syntheses from higher level descriptions, even complex circuits can be described as state machines, usually with a substantial amount of states.

In asynchronous circuits, every operator adhering to a handshake protocol and thus usually state holding, can be described with an AFSM. In this example, the control circuit for a 4-to-1 multiplexer using four phase handshaking is described. Such multiplexers are used to select operands for the ALU of the processor. Please find the description in Figure 5.1.

The multiplexer has four passive input channels, named a, b, c and d in the description. Since this is a control circuit only, each of the input channels is represented by two wires only. The first wire using the subscript V indicates valid data on the channel, thus it is the output of a completion detection circuit and equivalent to a request on the respective channel. The second wire is the acknowledgement, using the subscript A, and is an output of the control circuit. To allow the selection of one of the four input channels, the circuit has a two bit wide control channel with the two bits called $ctl0$ and $ctl1$, both having additional indexes 0 and 1 because of the dual rail NCL coding. The wire ctl_A is the acknowledgement wire for the control channel. Furthermore, x_A is the acknowledgement wire of the output channel.

The multiplexer selects a value to be passed by assigning one of en_a, en_b, en_c and en_d a high value. The implementation of the data path can thus consist of AND gates to mask the input channels and 4-input OR gates for merging the data.

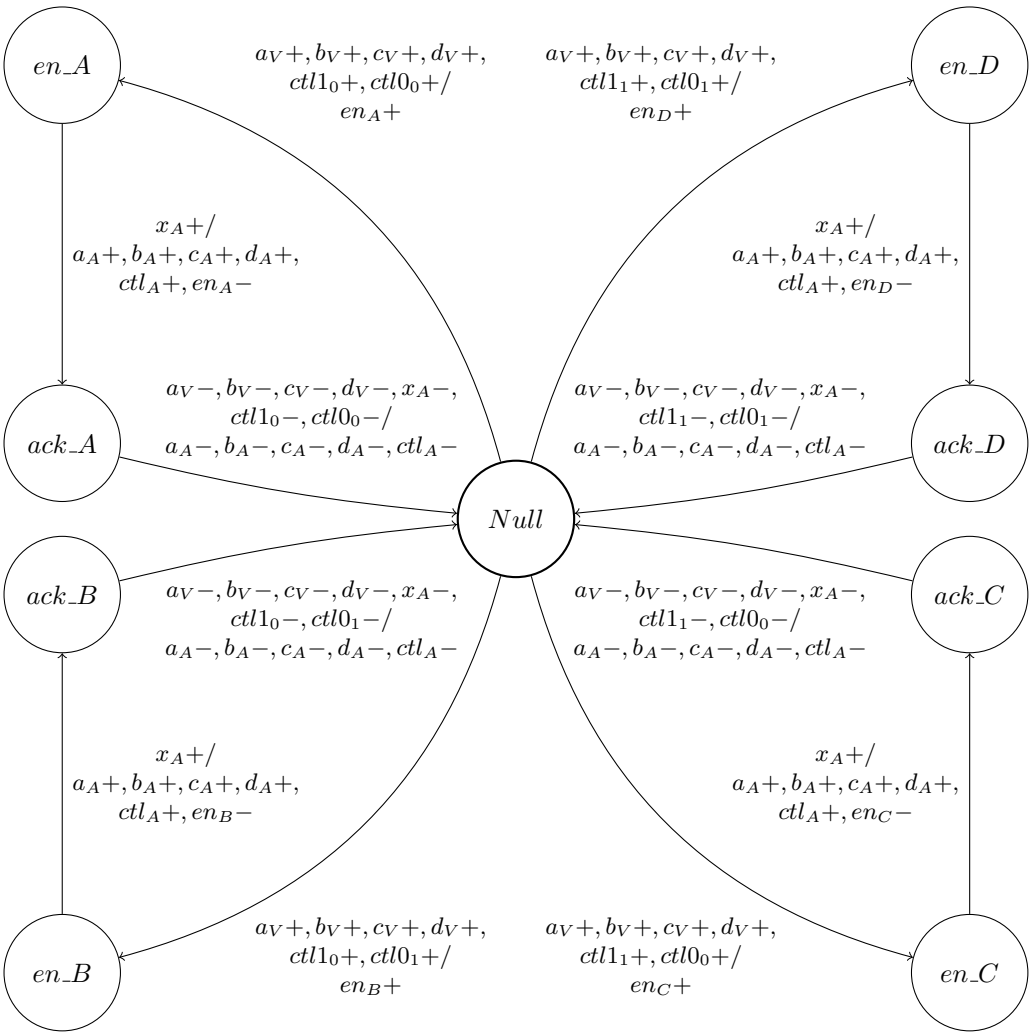


Figure 5.1: The AFSM description of the 4-to-1 multiplexer control circuit

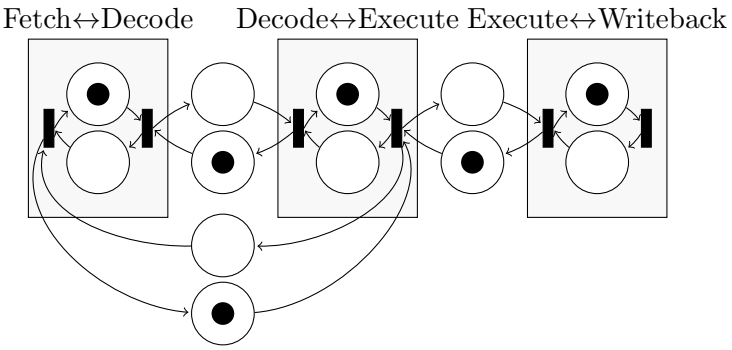


Figure 5.2: The high level PN model of the simplified pipeline

5.2.3 Signal Transition Graphs

STGs have the important property of being essentially labelled Petri nets, therefore allowing all the established PN theory to be applied to them. Moreover, it is common to use high level PN models of complex pipelined circuits for performance analysis and verification of properties like freedom from deadlocks. STGs can be used to refine such a model to generate control signals for the circuit described. Their use therefore ranges from small circuits to control circuits for complex circuits.

For the purpose of this example, the pipeline of the processor will be considered as simplified to a symbolic four stage pipeline with only one feedback path. The chosen feedback path is the signal going from the decode pipeline register to the fetch stage, where it controls whether the program counter should be incremented or the computed jump destination should be used as the next value.

Figure 5.2 shows the high level PN model of the pipeline, where each register is modelled as two latches, enclosed in a shaded rectangle in the figure. The result is a *doubly-latched* pipeline as described in [66]. This model can already be analysed to validate the data flow in the processor and verify essential properties like liveness.

Figure 5.3 shows a very basic refinement of the PN model to an STG that describes the generation of control signals for the latches, using the approach from [65]. The two latches replacing each register are called the *master* and *slave* latch in the order of the data flow. In the figure, the M signals, with the number of the register as suffix, are the control signals for the master latches. S signals control the slave latches accordingly. It is assumed, that a high control signal level makes a latch transparent.

Note that the control circuit described in Figure 5.3 could only be used for bounded delay implementations, after the appropriate additional delays would have been added to the control lines following the synthesis of the circuit. A more advanced control circuit for the doubly-latched pipeline is presented in [66].

5.2.4 Timed Event/Level Structures

Although TEL structures can be used equivalently to PRSs and STGs for the same descriptions as above, the main advantage of TEL structures is their ability to describe timing. Therefore, the typical use case for TEL structures is the description of circuits with timing based optimisations and the verification of temporal circuit behaviour.

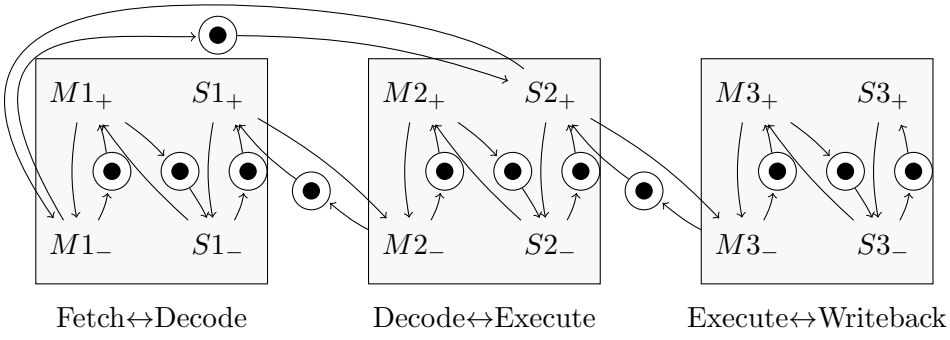


Figure 5.3: The STG refinement of the PN model

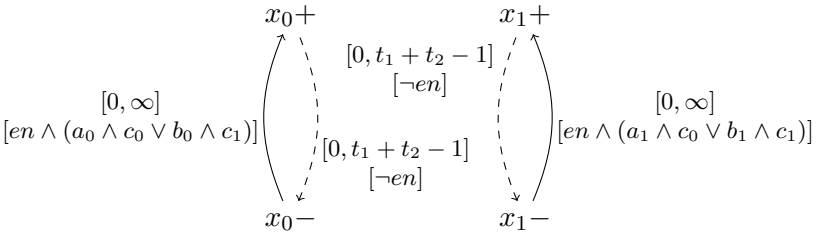


Figure 5.4: The TEL structure of the 2-to-1 binary multiplexer

For this example, consider the 2-to-1 binary multiplexer from Section 5.2.1. The proposed implementation used an additional signal *en* to increase concurrency in the back-to-*NULL* phase, and each multiplexer had a signal *ne* indicating validity/neutrality of the output value. Using TEL structures, the circuit can be described without the *ne* signal as it is used to prevent a non-critical race condition that can be precluded using a simple timing assumption. Figure 5.4 shows a description of the multiplexer equivalent to that in Section 5.2.1, with the *ne* signal omitted. Note that the time required for the outputs to be pulled back to *NULL* after *en* has changed to *false* is limited by the upper bound $t_1 + t_2 - 1$. For the time being, please assume this to be any positive integer.

In order to verify the correctness of the circuit, the environment and its assumed temporal behaviour has to be modelled, too. Figure 5.5 shows the considered environment. For better readability, it was chosen to consider only a single multiplexer enclosed by pipeline registration points. In the figure, the simple TEL structure on the left hand side shows the acknowledgement generation on the receiver's side, depending on the validity of data from the last stage of multiplexers (if there were more in the example). The larger TEL structure shows the generation of the input signals to the multiplexer, including the *en* signal. There are two timing assumptions laid upon the environment in the description. The first, t_1 in the rule having \$1 and \$2 as enabling and enabled events, respectively, requires the first transition on any of the inputs to occur at least t_1 time units after the acknowledgement from the receiver drops to *false*. The second assumption, t_2 in the smaller TEL structure, requires the acknowledgement to be generated at least t_2 time units after the data being input to the receiver have been sensed as valid. Note that apart from these assumptions, the circuit — most notably, the complete forward path where

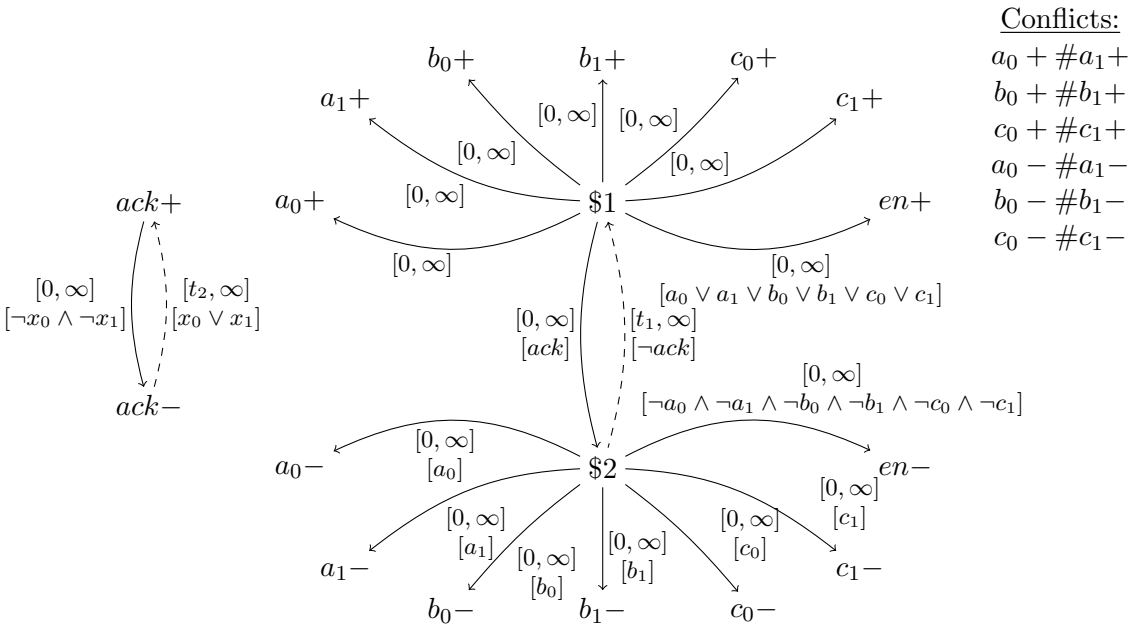


Figure 5.5: The TEL structure of the environment for the 2-to-1 binary multiplexer

the desired functionality, in this case the barrel shifter, is implemented — is described as delay insensitive.

Let t_1 and t_2 be any positive integers, either guessed or derived from post-synthesis circuit timing analysis, with t_2 including the delay of the acknowledgement wire between the registration points. Then, $t_1 + t_2$ is the minimal time between the return to *NULL* of the data output of the last stage of multiplexers and the input of new data to the first stage. Since $t_1 + t_2 - 1$ (see Figure 5.4) is the maximum time any multiplexer can delay the return to *NULL* of its outputs (including the delay of the *en* wire), it can be seen that when new data are being input, all stages have already cleared their outputs. A tool can be used to verify such constraints for circuits of much higher complexity.

5.2.5 Communicating Hardware Processes

For the CSP-based description methods, various parts of the processor's data path were chosen. CHP was used to describe a part of the execute stage, with the ALU limited to a subset of instructions.

The first part of the code defines the necessary data types. It is assumed that the configuration parameter *WORD_W* has already been defined. Please note that symbol types, such as *ALUCTRL* and *CARRYCTRL* in this example, are not bound to predefined hardware implementations. In most cases, the implementation can be affected by settings of the design compiler, however, these types should not be used to exchange data between hardware blocks described with different description methods, unless the compatibility has been examined with great care.

```
1 type WORD = {0..2^WORD_W-1};
2 type ALUCTRL = {ALU_NOP, ALU_LDLIU, ALU_LDHI,
```

```
3                 ALU_AND, ALU_OR, ALU_EOR, ALU_ADD,
4                 ALU_SUB, ALU_CMPEQ, ALU_CMPUGT, ALU_CMPULT,
5                 ALU_CMPGT, ALU_CMPLT,
6                 ALU_NOT, ALU_NEG,
7                 ALU_SL, ALU_SR, ALU_SRA, ALU_RRC, ALU_BYPR1,
8                 ALU_BYPR2, ALU_BYPEXC};
9  type CARRYCTRL = {CARRY_IN, CARRY_NOT, CARRY_ZERO, CARRY_ONE};
10 type ALUFLAG = {0..2^5-1};
11 const COND = 4;
12 const ZERO = 3;
13 const NEG  = 2;
14 const CARRY = 1;
15 const OVER = 0;
```

The next part of the code defines the procedure *alu*, which performs all of the data related operations.

```
16 procedure alu(data1, data2, excvec : WORD;
17              aluctrl : ALUCTRL;
18              carryin : {0..1};
19              res staflag : ALUFLAG;
20              res result : WORD)
21 CHP {
22   var addresult : {0..2^(WORD_W+1)-1};
23   staflag := 0 ;
24   [  aluctrl = ALU_LDLIU -> result[8..WORD_W-1] := data1[8..WORD_W-1] ,
25                             result[0..7] := data2[0..7]
26   [] aluctrl = ALU_ADD -> addresult := data1 + data2 + carryin ;
27                           result := addresult[0..WORD_W-1] ,
28                           staflag[CARRY] := addresult[WORD_W]
29   [] aluctrl = ALU_AND -> result := data1 & data2 ;
30                           staflag[COND] := result != 0
31   [] aluctrl = ALU_CMPGT ->
32        [  data1[WORD_W-1] != data2[WORD_W-1] ->
33             addresult := data1 + data2 + carryin ;
34             [  addresult = 0 -> staflag[COND] := addresult[WORD_W-1]
35             [] addresult != 0 -> staflag[COND] := ~addresult[WORD_W-1]
36             ]
37        [] data1[WORD_W-1] = data2[WORD_W-1] ->
38             staflag[COND] := ~data1[WORD_W-1]
39        ]
40   [] aluctrl = ALU_RRC -> result[0..WORD_W-2] := data1[1..WORD_W-1] ,
41                           result[WORD_W-1] := carryin[0] ,
42                           staflag[CARRY] := data1[0]
43   [] aluctrl = ALU_BYPEXC -> result := excvec
44   ] ;
45   staflag[ZERO] := result = 0 ,
46   staflag[OVER] := data1[WORD_W-1] = data2[WORD_W-1]
47                    & result[WORD_W-1] xor data1[WORD_W-1] ,
48   staflag[NEG] := result[WORD_W-1]
```

The final listing shows the process *execute*, which acts as a "wrapper" for the *alu* procedure. In its first part, it concurrently reads all the inputs. Then, various multiplexing tasks are performed in the processor of which the carry selection and the optional bitwise negation of the second operand are shown in this example. Finally the *alu* procedure is called and the result is communicated to the next stage.

```
49
50 process execute()(exedata1?, exedata2? : WORD;
51                   result! : WORD;
52                   aluctrl? : ALUCTRL;
53                   carry, carryflag : {0..1};
54                   negdata : bool)
55 CHP {
56   var v_exedata1, v_exedata2, v_aluexedata2 : WORD;
57   var v_aluctrl : ALUCTRL;
58   var v_alucarry, v_carry, v_carryflag : {0..1};
59   var v_negdata: bool;
60   var r_staflag : ALUFLAG;
61   var r_result : WORD;
62   *[
63     exedata1?v_exedata1 ,
64     exedata2?v_exedata2 ,
65     aluctrl?v_aluctrl ,
66     carry?v_carry ,
67     carryflag?v_carryflag ,
68     negdata?v_negdata ;
69     [  v_carry = CARRY_IN -> v_alucarry := v_carryflag
70     [] v_carry = CARRY_NOT -> v_alucarry := ~v_carryflag
71     [] v_carry = CARRY_ZERO -> v_alucarry := 0
72     [] v_carry = CARRY_ONE -> v_alucarry := 1
73     ] ,
74     [  v_negdata -> v_aluexedata2 := v_exedata2
75     [] ~v_negdata -> v_aluexedata2 := ~v_exedata2
76     ] ;
77     alu(v_exedata1, v_aluexedata2, v_alucarry,
78         v_aluctrl, v_alucarry, r_staflag, r_result) ;
79     result!r_result
80   ]
81 }
```

5.2.6 Haste

Haste was used to describe the complete fetch stage, including instruction memories.

The first two lines define the data types *WORD* and *INSTR* data types. As in CHP, it is assumed that configuration parameters, namely *WORD_W*, *INSTR_RAM_CFG_C*, *SIZE_BOOT_ROM* and *MAP_BOOTROM_CFG_C* were defined previously or imported from a configuration file.

The most notable language property shown on this example is the handling of memory. The two variables *iram* and *brom* are arrays of instructions representing the memories. The keywords **ram** and **rom**, respectively, cause memory access interfaces to be generated instead of memories during synthesis. The direct access to variable arrays instead of the generation of memory interface signals considerably simplifies the description.

```
1   WORD=type [0..2^WORD_W-1]
2 & INSTR=type [0..2^16-1]
3 &
4 fetch: proc(f_jmpexe ? chan bool pas
5           & f_jmpdest ? chan WORD pas
6           & f_bromdata, f_iramdata ! chan INSTR act
7           & f_pcnt ! chan WORD act).
8 begin
9   iram: ram array [0..2^INSTR_RAM_CFG_C-1] of INSTR arb:
```

```
& brom: rom array [0..2^SIZE_BOOT_ROM-1] of INSTR
& pcnt: var WORD := 2^MAP_BOOTROM_CFG_C-1
& pcnt_new: var WORD
& jmpexe: var bool
& jmpdest: var WORD
|
  forever do
    f_jmpexe?jmpexe ||
    f_jmpdest?jmpdest ;
    if jmpexe then pcnt_new := jmpdest
              else pcnt_new := pcnt + 1
    fi ;
    f_bromdata!brom[pcnt_new] ||
    f_iramdata!iram[pcnt_new] ||
    f_pcnt!pcnt_new ;
    pcnt := pcnt_new
  od
end
```

5.2.7 Balsa

To also demonstrate the use of Balsa in its typical application, it was used to describe part of the decode stage. As well as the ALU in the execute stage, the described instruction decoder only implements a subset of the processor's instructions.

The first part of the description defines all the required data types including enumerations that are also used in the description of the ALU in Section 5.2.5. Again, note that these types are symbolic and the hardware implementation of channels and variables with this type might not be compatible with other description methods.

```
type WORD is WORD_W bits
type INSTR is 16 bits
type REGADDR is 4 bits
constant INSTR_ADDR_NULL = (0 as array WORD_W - MAP_BOOTROM_CFG_C of bit)
type ALUCTRL is enumeration
  ALU_NOP, ALU_LDLIU, ALU_LDHI, ALU_AND, ALU_OR, ALU_EOR,
  ALU_ADD, ALU_SUB, ALU_CMPEQ, A  LU_CMPUGT, ALU_CMPULT,
  ALU_CMPGT, ALU_CMPLT, ALU_NOT, ALU_NEG, ALU_SL, ALU_SR,
  ALU_SRA, ALU_RRC, ALU_BYPR1, ALU_BYPR2, ALU_BYPEXC
end
type STACTRL is enumeration
  SET_FLAG, SET_COND, SAVE_SR, REST_SR
end
type CARRYCTRL is enumeration
  CARRY_IN, CARRY_NOT, CARRY_ZERO, CARRY_ONE
end
```

The next part of the code shows the procedure `decode`, which prior to decoding an instruction chooses whether it should be taken from the boot ROM or the instruction RAM.

```
procedure decode(input f_bromdata, f_iramdata : INSTR;
                 input f_pcnt : WORD;
                 output imm : WORD;
                 output aluctrl : ALUCTRL;
                 output regfwr, staen, useimm, negdata2 : bit;
```

```
22                  output stactrl : STACTRL;
23                  output carry : CARRYCTRL;
24                  output regfaddr1, regfaddr2 : REGADDR) is
25    variable pcnt : WORD
26    variable decinstr : INSTR
27  begin
28    loop
29      f_pcnt -> pcnt ;
30      if #pcnt[MAP_BOOTROM_CFG_C..WORD_W-1] /= INSTR_ADDR_NULL then
31        f_bromdata -> decinstr ||
32        select f_iramdata then continue end
33      else
34        f_iramdata -> decinstr ||
35        select f_bromdata then continue end
36      end ;
```

The rest of the code contains the **case** statement that decodes the instruction and assigns various channels variables that control the function of the processor.

```
37      select decinstr then
38        case (#instr[12..15] as 4 bits) of
39          -- LDLIU
40          0b0010 then regfwr <- 1 ||
41                              regfaddr1 <- (#instr[0..3] as 4 bits) ||
42                    regfaddr2 <- (#instr[4..7] as 4 bits) ||
43                      imm <- (#instr[4..11] as 16 bits) ||
44                      aluctrl <- ALU_LDLIU ||
45                    carry <- CARRY_IN ||
46                      useimm <- 1 ||
47                    staen <- 0 ||
48                      negdata2 <- 0
49          -- CMP_GT
50        | 0b1011 then
51            if (#instr[8..11] as 4 bits) = 0b0010 then
52             aluctrl <- ALU_CMPGT ||
53             regfwr <- 1 ||
54               regfaddr1 <- (#instr[0..3] as 4 bits) ||
55               regfaddr2 <- (#instr[4..7] as 4 bits) ||
56               useimm <- 0 ||n
57               staen <- 1 ||
58               stactrl <- SET_COND ||
59               carry <- CARRY_ONE ||
60             negdata2 <- 1
61            end
62        | 0b1100 .. 0b1110 then
63            case (#instr[8..11] as 4 bits) of
64              -- ADD
65              0b0001 then regfwr <- 1 ||
66                            regfaddr1 <- (#instr[0..3] as 4 bits) ||
67                            regfaddr2 <- (#instr[4..7] as 4 bits) ||
68                            aluctrl <- ALU_ADD ||
69                          useimm <- 0 ||
70                          staen <- 1 ||
71                          stactrl <- SET_FLAG ||
72                          carry <- CARRY_ZERO ||
73                            negdata2 <- 0
74              -- AND
75            | 0b0101 then regfwr <- 1 ||
```

```
76                          regfaddr1 <- (#instr[0..3] as 4 bits) ||
77                          regfaddr2 <- (#instr[4..7] as 4 bits) ||
78                          aluctrl <- ALU_AND ||
79                        useimm <- 0 ||
80                          staen <- 1 ||
81                        stactrl <- SET_FLAG ||
82                        carry <- CARRY_ONE ||
83                          negdata2 <- 0
84            end
85              -- RRC
86            | 0b0101 then regfwr <- 1 ||
87                          regfaddr1 <- (#instr[0..3] as 4 bits) ||
88                          regfaddr2 <- (#instr[4..7] as 4 bits) ||
89                          aluctrl <- ALU_RRC ||
90                        useimm <- 0 ||
91                          staen <- 1 ||
92                        stactrl <- SET_FLAG ||
93                        carry <- CARRY_ONE ||
94                          negdata2 <- 0
95            end
96          | 0b1111 then
97              -- LDVEC
98              if (#instr[8..11] as 4 bits) = 0b101X then
99                regfwr <- 1 ||
100               regfaddr1 <- (#instr[0..3] as 4 bits) ||
101               regfaddr2 <- (#instr[4..7] as 4 bits) ||
102               staen <- 0 ||
103               aluctrl <- ALU_BYPEXC ||
104             carry <- CARRY_ONE ||
105               useimm <- 0 ||
106               negdata2 <- 0
107             end
108         end
109       end
110     end
111 end
```

Conclusion

The aim of this work to create a comparison of seven of the most established description methods based on selected criteria that could serve prospective designers of asynchronous circuits when choosing a description method for a project in addition to being valuable for a didactic presentation of asynchronous circuits.

After the criteria for comparison were extracted from the principles of asynchronous circuit design, the description methods were compared with respect to those criteria. A further comparison was made by systematically describing a carefully chosen exemplary circuit with all the methods and analysing the results. An additional example showed the power of each method in its intended application by describing an appropriate part of the SPEAR2 processor.

The comparison showed that the differences between description methods can be surprisingly extensive, even when comparing methods with approximately the same level of abstraction. This is especially true for the three CSP-based description methods where the expected differences were rather insignificant. From the methods that described the behaviour based on levels or transitions on single wires, TEL structures were found to be very powerful as they unite the advantages from both, PRSs and STGs, with the additional possibility to describe timing assumptions. From three CSP-based description methods that utilise implicitly synchronised channels for communication and synchronisation, Haste was found to be the most powerful and mature, providing implicitly multiplexed access to shared resources, a simple description of memory interfaces and constructs for handling level sensitive single rail wires. Balsa as well as CHP had remarkable limitations in the description of handshake enclosure and CHP was additionally the description method with the worst documentation, partially disallowing to reason about some of its properties.

The work also showed that the description of asynchronous circuits does not have to be a fearsome task and in fact, from the author's experience during the creation of this work, once understanding the properties of asynchronous circuits and the problems arising from the design of highly concurrent systems, the learning curve for the description methods is surprisingly fast. The only difficulties arise when encountering something the description method currently used does not support.

Even after decades of research devoted to asynchronous circuits and numerous successful industrial and academic asynchronous circuits clearly demonstrating their superiority when carefully designed, engineers and decision-makers from the industry remain sceptical and continue to hold on to the approved synchronous design style. It is the belief of the author of this work that with further progresses in technology the difference in the performance and power consumption of synchronous and asynchronous functionally equivalent

circuits will rise to the advantage of the latter. This will eventually force the industry to invest into the technology resulting in progressively more powerful tools. Also the author believes that only a few description methods will be used in the future, e.g. from the three CSP-based description with the same field of application, only one will be used by a significant majority of engineers resulting in better community support, tool development and more concentrated investments.

Bibliography

[1] P. Forshaw and R. Hahn, "Synchronous design: the right technique for digital ASICs," in *Proceedings of the Third Annual IEEE ASIC Seminar and Exhibit, 1990*, pp. P6/1.1–P6/1.5, Sept. 1990.

[2] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Coping with the variability of combinational logic delays," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD 2004*, pp. 505–508, Oct. 2004.

[3] P. Gronowski, W. Bowhill, R. Preston, M. Gowan, and R. Allmon, "High-performance microprocessor design," *IEEE Journal of Solid-State Circuits*, vol. 33, pp. 676–686, May 1998.

[4] S. Wilton, N. Kafafi, J. Wu, K. Bozman, V. Aken'Ova, and R. Saleh, "Design considerations for soft embedded programmable logic cores," *IEEE Journal of Solid-State Circuits*, vol. 40, pp. 485–497, Feb. 2005.

[5] A. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proceedings of the sixth MIT conference on Advanced research in VLSI*, (Cambridge, MA, USA), pp. 263–278, MIT Press, 1990.

[6] R. Manohar and A. Martin, "Quasi-delay-insensitive circuits are Turing-complete. invited article," in *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996.

[7] C. Seitz, "System timing," in *Introduction to VLSI Systems* (C. Mead and L. Conway, eds.), ch. 7, Reading, MA: Addison-Wesley, 1980.

[8] W. Bainbridge, W. Toms, D. Edwards, and S. Furber, "Delay-insensitive, point-to-point interconnect using m-of-n codes," in *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems, 2003*, pp. 132–140, May 2003.

[9] T. Chaney and C. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Transactions on Computers*, vol. C-22, pp. 421–422, Apr. 1973.

[10] A. Martin and M. Nystrom, "Asynchronous techniques for system-on-chip design," *Proceedings of the IEEE*, vol. 94, p. 1089, June 2006.

[11] A. Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," Tech. Rep. Caltech-CS-TR-89-1, California Institute of Technology, 1989.

[12] I. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, pp. 720–738, June 1989.

[13] J. Sparsø, "Asynchronous circuit design - a tutorial," in *Chapters 1-8 in "Principles of asynchronous circuit design - A systems Perspective"*, pp. 1–152, Boston / Dordrecht / London: Kluwer Academic Publishers, dec 2001.

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[15] A. Martin, S. Burns, T. Lee, D. Borkovic, and P. Hazewindus, "The design of an asynchronous microprocessor," Tech. Rep. Caltech-CS-TR-89-2, California Institute of Technology, Department of Computer Science, 1989.

[16] A. Martin, S. Burns, T. Lee, D. Borkovic, and P. Hazewindus, "The first asynchronous microprocessor: The test results," Tech. Rep. Caltech-CS-TR-89-6, California Institute of Technology, Department of Computer Science, June 1989.

[17] S. Furber, P. Day, J. Garside, N. Paver, and J. Woods, "AMULET1: A micropipelined ARM," in *Proceedings of CompCon'94*, pp. 476–485, IEEE Computer Society Press, Mar. 1994.

[18] S. Furber, J. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. Paver, "AMULET2e: an asynchronous embedded controller," *Proceedings of the IEEE*, vol. 87, pp. 243–256, Feb. 1999.

[19] J. Garside, S. Furber, and S.-H. Chung, "AMULET3 revealed," in *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1999*, pp. 51–59, 1999.

[20] A. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. Lee, "The design of an asynchronous MIPS R3000 microprocessor," in *Proceedings of the 17th Conference on Advanced Research in VLSI*, pp. 164–181, IEEE Computer Society Press, 1997.

[21] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, "An asynchronous low-power 80C51 microcontroller," in *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1998*, pp. 96—-107, 1998.

[22] A. Martin, M. Nystrom, K. Papadantonakis, P. Penzes, P. Prakash, C. Wong, J. Chang, K. Ko, B. Lee, E. Ou, J. Pugh, E.-V. Talvala, J. Tong, and A. Tura, "The Lutonium: a sub-nanojoule asynchronous 8051 microcontroller," in *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems, 2003*, pp. 14–23, May 2003.

[23] J. Teifel and R. Manohar, "Highly pipelined asynchronous FPGAs," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, Feb. 2004.

[24] D. Fang, J. Teifel, and R. Manohar, "A high-performance asynchronous FPGA: test results," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2005*, pp. 271—-272, Apr. 2005.

[25] A. Martin and M. Nystrom, "CAST: Caltech asynchronous synthesis tools," in *Proc. of Fourth Asynchronous Circuit Design Working Group Workshop*, June 2004.

[26] A. Martin, "Synthesis of asynchronous VLSI circuits," Tech. Rep. Caltech-CS-TR-93-28, California Institute of Technology, Department of Computer Science, Aug. 1991.

[27] E. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, pp. 453–457, Aug. 1975.

[28] G. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[29] E. Moore, "Gedanken experiments on sequential machines," in *Automata Studies*, pp. 129–153, New Jersey: Princeton University Press, 1956.

[30] D. Huffman, "The synthesis of sequential switching circuits," *Journal of the Franklin Institute*, vol. 257, no. 3-4, pp. 161–190 and 275–303, 1954.

[31] S. Nowick, "Automatic synthesis of burst-mode asynchronous controllers," Tech. Rep. CSL-TR-95-686, Computer Systems Laboratory, Stanford University, Dec. 1995.

[32] K. Yun, D. Dill, and S. Nowick, "Practical generalizations of asynchronous state machines," in *Proceedings of the fourth European Conference on Design Automation, 1993, with the European Event in ASIC Design*, pp. 525–530, feb 1993.

[33] K. Yun and D. Dill, "Automatic synthesis of extended burst-mode circuits. I. (Specification and hazard-free implementations)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 101–117, feb 1999.

[34] K. Yun and D. Dill, "Automatic synthesis of extended burst-mode circuits. II. (Automatic synthesis)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 118–132, feb 1999.

[35] R. Fuhrer, S. Nowick, M. Theobald, N. Jha, B. Lin, and L. Plana, "MINIMALIST: An environment for the synthesis, verification and testability of burst-mode asynchronous machines," Tech. Rep. CUCS-020-99, Computer Science Department, Columbia University, July 1999.

[36] C. Petri, *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[37] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, Apr. 1989.

[38] T.-A. Chu, *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1987.

[39] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. E80-D, pp. 315–325, Mar. 1997.

[40] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "SIS: a system for sequential circuit synthesis," Tech. Rep. UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[41] R. Alur, *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford Univ., Stanford, CA, 1991.

[42] P. Merlin and D. Farber, "Recoverability of communication protocols — implications of a theoretical study," *IEEE Transactions on Communications*, vol. 24, pp. 1036–1043, Sept. 1976.

[43] G. M. Reed and A. W. Roscoe, "A timed model for communicating sequential processes," in *International Colloquium on Automata, Languages and Programming on Automata, Languages and Programming*, (New York, NY, USA), pp. 314–323, Springer-Verlag New York, Inc., 1986.

[44] C. Myers, *Computer-Aided Synthesis And Verification Of Gate-Level Timed Circuits*. PhD thesis, Stanford Univ., Stanford, CA, 1995.

[45] W. Belluomini and C. Myers, "Timed event/level structures," in *Collection of papers from TAU'97*, pp. 39–44, 1997.

[46] W. Belluomini, C. Myers, and H. Hofstee, "Timed circuit verification using TEL structures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 129–146, Jan. 2001.

[47] R. Thacker, W. Belluomini, and C. Myers, "Timed circuit synthesis using implicit methods," in *Proceedings of the Twelfth International Conference On VLSI Design, 1999*, pp. 181–188, jan 1999.

[48] C. Myers, P. Beerel, and T.-Y. Meng, "Technology mapping of timed circuits," in *Proceedings of the Second Working Conference on Asynchronous Design Methodologies, 1995*, pp. 138–147, may 1995.

[49] W. Belluomini and C. Myers, "Timed state space exploration using POSETs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 501–520, may 2000.

[50] C. Myers, W. Belluomini, K. Killpack, E. Mercer, E. Peskin, and H. Zheng, "Timed circuits: a new paradigm for high-speed design," in *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pp. 335–340, 2001.

[51] C. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.

[52] C. Hoare, *Communicating Sequential Processes.* Prentice-Hall International, UK, Ltd., 1985.

[53] A. Roscoe, *Theory and Practice of Concurrency.* Prentice Hall PTR, Nov. 1997.

[54] A. Martin, "The probe: an addition to communication primitives," Tech. Rep. CaltechCSTR:1984.5124-tr-84, California Institute of Technology, Feb. 1984.

[55] M. van der Goot and C. Moore, *CHPSIM user manual*, 2010.

[56] R. Manohar, *CAST language description.* California Institute of Technology, 1997.

[57] Handshake Solutions, *Haste — Programming Language Manual*, 2009.

[58] Handshake Solutions, *TiDE Manual*, 2009.

[59] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," in *Proceedings of the European Conference on Design Automation. EDAC*, pp. 384–389, feb 1991.

[60] D. Edwards, A. Bardsley, L. Janin, L. Plana, and W. Toms, *Balsa: a Tutorial Guide.* Advanced Processor Technologies Group, University of Manchester, 2006.

[61] A. Bardsley, *Implementing Balsa Handshake Circuits.* PhD thesis, University of Manchester, 2000.

[62] K. Low and A. Yakovlev, "Token ring arbiters: an exercise in asynchronous logic design with Petri nets," tech. rep., University of Newcastle upon Tyne, Nov. 1995.

[63] M. Fletzer, "SPEAR2 - an improved version of SPEAR," Master's thesis, Vienna University of Technology, Vienna, feb 2008.

[64] J. Mosser, "AMBA4SPEAR2: An AMBA extension module for the SPEAR2 processor core," Master's thesis, Vienna University of Technology, Vienna, mar 2008.

[65] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, pp. 1904–1921, oct. 2006.

[66] R. Kol and R. Ginosar, "A doubly-latched asynchronous pipeline," in *Proceedings of the 1997 IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD 1997*, pp. 706–711, oct 1997.