School of Electrical, Electronic & Computer Engineering

UNIVERSITY OF
NEWCASTLE UPON TYNE

Automated synthesis of asynchronous circuits using direct mapping for control and data paths

Danil Sokolov

Technical Report Series

NCL-EECE-MSD-TR-2006-111

January 2006

Contact:

danil.sokolov@ncl.ac.uk

EPSRC supports this work via GR/R16754 (BESST) and GR/S81421 (SCREEN).

NCL-EECE-MSD-TR-2006-111

School of Electrical, Electronic & Computer Engineering,

Merz Court,

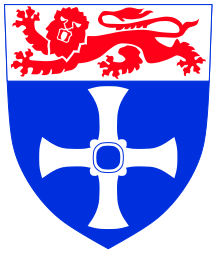
University of Newcastle upon Tyne,

Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

University of Newcastle upon Tyne

School of Electrical, Electronic and Computer Engineering



Automated synthesis of asynchronous circuits using direct mapping for control and data paths

by Danil Sokolov

PhD Thesis

January 2006

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would like to express my gratitude to Alex Yakovlev, my supervisor, for introducing me to the world of asynchronous circuits and providing invaluable guidance during my research. My colleague Alex Bystrov deserves a lot of credit for shaping my ideas in numerous discussions. I am grateful to my teachers, especially to Oleg Maevsky and Gennady Desyatkov, for giving me the background knowledge in computing science. I learnt a great deal about research from these people.

I would like to thank Agnes Madalinski, Frank Burns and Delong Shang for the discussions of the design techniques reviewed in Chapter 3; Alex Bystrov, Maciej Koutny and Victor Khomenko for formalisation of the ideas and algorithms presented in Chapter 4; Alex Kondratyev, Albert Koelmans and Simon Moore for their helpful comments on the data path synthesis method described in Chapter 5; Julian Murphy for design of the AES chip which will be used to evaluate the ideas of Chapter 6. Special thanks to Ellis Solaiman and Agnes Madalinski for reading my thesis and sharing their experience of scientific writing.

Thanks to collaboration with Atmel Inc it was possible to fabricate and evaluate several circuits designed by the method proposed in Chapters 5 and 6. I am grateful to Atmel engineers Russ Hobson and Stephen Pickles for their practical feedback, which helped to improve the software tools developed in scope of this research.

I am thankful to my wife Nina for her love and patience, particularly during my writing up term. I am grateful to my family for the concern and support during this research. My thanks to all my friends who made my life even more fun by involving me into various sport and social activities.

This research was supported by the ORS Awards Scheme grant, the EPSRC grants GR/R16754 (BESST) and GR/S81421 (SCREEN).

Abstract

A method for automated synthesis of asynchronous circuits using direct mapping for control path and data path is presented. The idea of direct mapping is that a graph specification of a system is translated into a circuit netlist by mapping the graph nodes into circuit elements and the graph arcs into circuit interconnects. The key feature of this approach is its low algorithmic complexity and direct correspondence between the elements of the initial specification and the components of the resultant circuit. Unlike other direct mapping techniques, in our method the control path and data path are synthesised separately seeking for greater performance of the circuit.

The control path synthesis starts from an initial specification in form of a Signal Transition Graph (STG). The STG is split into a device and an environment, which synchronise via a communication net that models wires. The device is represented as a tracker and a bouncer. The tracker follows the state of the environment and provides reference points to the device outputs. The bouncer interfaces to the environment and generates output events in response to the input events according to the state of the tracker. This two-level architecture provides an efficient interface to the environment and is convenient for subsequent mapping into a circuit netlist. A set of optimisation heuristics are developed to reduce the latency and size of the control circuit.

The method for data path synthesis is based on a conventional RTL design flow. The data path components are first implemented by a standard RTL synthesis tool, e.g Synopsys. The obtained circuits are then converted into a hazard-free logic by using a dual-rail encoding with a return-to-spacer signalling. A new protocol with two spacers alternating in time is proposed which makes all gates switch per computation cycle. The potential applications of this protocol are security circuits, online testing, dynamic logic.

As a result of this work, several software tools are developed, namely OptiMist for synthesis of low-latency control path, and VeriMap for synthesis of hazard-free data path. The tools are successfully integrated in the BESST design flow to provide a front-end to high-level HDLs and an interface to conventional EDA tools for simulation, timing analysis and place-and-route.

Chapter 1

Introduction

Design of asynchronous circuits has been an active research area since the early days of digital circuit design. Until this decade asynchronous circuits have only been applied commercially as small sub-circuits, often as peripherals to controllers. Emphasis is now shifting to asynchronous Systems-on-Chip (SoCs), which are progressing from an academic research topic to a viable solution to a number of digital VLSI design challenges.

This chapter briefly outlines the motivation of this work and overviews the approaches to design of asynchronous circuits. The main contribution and the organisation of this thesis are also described in this chapter.

1.1 Motivation

The continuous growth of circuit integration level creates a design gap between semiconductor manufacturing capability and the ability of Electronic Design Automation (EDA) tools [1]. One of the ways to deal with the increasing complexity of logic circuits is to improve the efficiency of the design process. In particular, design automation and component reuse help to solve the problem. SoCs have proved to be a particularly effective way to facilitate design automation and component reuse. An important role in the synthesis of SoCs is given to the aspects of modelling concurrency and timing. These aspects have traditionally been dividing systems into clocked and self-timed. The division has recently become fuzzier because systems are built in a mixed timing style: partly clocked and partly self-timed. The argument about the way how the system should

be constructed, synchronously or asynchronously, is moving to another round of evolution. It is accepted that the timing issue should only be addressed in the context of the particular design criteria, such as speed, power, security, modularity, etc.

Most of the circuits produced by industry are *synchronous*. The operation of their components is sequenced by one or more globally distributed periodic timing signals called *clock*. The design flow for synchronous circuits is widely supported by EDA tools, e.g. Cadence, Mentor Graphics, Synopsys, etc. However, trying to combine predesigned components into a globally clocked SoC, a designer faces a *timing closure* problem. Each IP core is designed for a certain clock period, assuming that the clock signal is delivered at the same time to all parts of the system. Finding a common clocking mode for the whole system is an obstacle to component reuse, which is difficult to overcome. A *clock skew* caused by variations of interconnect delay is another problem of synchronous circuits.

A promising method of composing systems from predesigned components is a *Globally Asynchronous Locally Synchronous* (GALS) architecture [22]. Each synchronous block in a GALS circuit is surrounded by an asynchronous wrapper which provides the communication between the blocks in asynchronous manner. This allows individual components to work on different clock speed, thus eliminating the need for a global clock with all of its associated problems, and increasing the modularity of a system.

An alternative for a SoC design is a *self-timed* architecture, where the individual components and all communication between components are designed asynchronously. This architecture offers a set of advantages which GALS approach does not have. In addition to better modularity and avoidance of clock distribution problem, asynchronous circuits can exhibit higher robustness to voltage, temperature and fabrication parameters, greater performance, power saving, lower electromagnetic noise, etc. [11, 35, 108, 81].

The major obstacle on the mainstream use of asynchronous design techniques is the lack of a coherent design flow, compatible with conventional EDA tools and libraries. The compatibility issue is essential because a large part of the design flow is mapping of the circuit netlist into silicon. For this task the traditional place and route tools can be reused. It is also possible to inherit the timing analysis and simulation tools. However, synthesis and verification tools intended

for synchronous systems omit important features of asynchronous components and have to be replaced.

The other impediment is that industry adheres to existing specification languages. The majority of industry designers think in terms of high-level Hardware Description Languages (HDLs), such as System-C, Verilog and VHDL. These languages were created with synchronous circuits in mind and are not easily applicable for asynchronous designs. On the other hand new languages created specially for asynchronous circuits design, e.g. Balsa [6], require significant changes in the industry design flow and training of the engineers, which is extremely costly.

Hardware security is also becoming an important design issue. For example, such applications as smart cards require measures to resist side channel attacks [71]. Whilst alternatives exist at the software level to balance power [94], the need for hardware solutions is also mandatory.

There are two types of side channel attacks: *timing* and *power*. The clock signal is typically used as a reference in timing attacks. System desynchronisation as in [76, 125] can help hide the clock signal. Masking the operation of a complete circuit is a complex task which could demand very expensive changes to the entire design flow.

A cheaper method is rebuilding individual blocks within the same synchronous infrastructure so, that their power signatures become independent of the mode of operation and of the data processed. This method is used in NCL-X approach [62], where synchronous pipelines are transformed into asynchronous circuits using dual-rail coding. The dual-rail coding helps balancing power consumption for bit values 0 and 1. However, the physical implementation of the rails at the gate level is not symmetric and the use of a standard return-to-spacer switching protocol on such dual-rail gates may leak secret data.

Special types of CMOS logic elements which exhibit data independent power consumption have been proposed in [112]. However, this low-level approach requires changing gate libraries and hence is costly for a standard cell or FPGA user. It is also difficult to build a dual-rail gate which consumes the same power regardless of processed data. Even if such a secure gate is built for one set of fabrication parameters (output load, supply voltage, environment temperature) it still can expose unbalanced power consumption in other conditions.

There is clear evidence that the incorporation of the asynchronous approach into the automated

design flow can improve designs. Even though the asynchronous techniques involve significant changes to the conventional design flow, the companies realise that this is the promising route to cover the design productivity gap. Such industry giants as IBM, Infineon, Intel, Philips, Sun etc. invest in synthesis and verification tools for asynchronous circuit design. They also replace parts of their new systems by asynchronous components, gradually replenishing design libraries with asynchronous IP cores.

1.2 Synthesis of asynchronous circuits by direct mapping

One of the ways to design circuits is the '*design-and-validate*' approach where a circuit is assembled from gates and small components in an ad-hoc manner. The correctness of the resultant circuit mostly depends on the experience of a designer and cannot be guaranteed. The circuit validation is performed at the level of hardware implementation which is unacceptable for large designs.

An alternative to the 'design-and-validate' approach is *synthesis* of a circuit from its mathematical specification. In the synthesis approach all verification and functionality checking are performed at the level of the mathematical model. This model is subsequently synthesised by the methods which guarantee that the hardware implementation preserves the functionality captured by the specification. In the last two decades the design flow for clocked circuits has been significantly improved by the ubiquitous use of the synthesis methods. For example, for RTL design flow the industry has stable CAD tools such as Cadence, Synopsys, etc. The asynchronous circuit synthesis, however, is still immature and requires a lot of investment to be used outside a research lab.

Two main approaches to the synthesis of asynchronous circuits are logic synthesis and direct mapping. *Logic synthesis* works with the low-level system specifications which capture the behaviour of the system at the level of signal transitions. In this approach boolean equations for the output signals of the circuit are derived using the *next state functions* [26]. In order to find the next state functions all possible orders of the events must be explored. Such an exploration may result in a state space which exponentially large w.r.t. the initial specification. The circuit optimisation often involves analysis and recalculation of the whole state space.

The logic synthesis approach is now well developed and supported by public tools (Pet-

rify [28], Minimalist [44], 3D [25]). However, this approach suffers from excessive computation complexity and memory requirements, thus it cannot be applied to large specifications. There is no transparent correspondence between the elements of the original specification, the intermediate representation of the state space and the components of the resultant circuit, which complicates the checking of circuit functionality.

The main idea of the *direct mapping* approach is that a graph specification of a system is translated into a circuit netlist in such a way that the graph nodes correspond to the circuit elements and graph arcs correspond to the interconnects. Direct mapping can typically be divided into three independent operations: *translation*, *optimisation* and *mapping*. Firstly, a system specification is translated into an intermediate graph representation convenient for subsequent mapping. Then, peephole optimisation is usually applied to the intermediate representation of a system. Finally, the optimised graph is mapped into a circuit netlist implementation. In a practical design flow, however, some operations can be merged together or not present at all, e.g. optimisation is often performed together with mapping and there are cases when the circuit implementation is obtained directly from the initial specification without converting it into an intermediate form.

The key feature of the direct mapping approach is its low algorithmic complexity. The use of heuristic-based local optimisation (as opposed to state-space global optimisation in a logic synthesis approach) also facilitates the computational simplicity of the method. The transparent correspondence between the elements of the initial specification and the components of the resultant circuit is advantageous for checking the functional correctness of the implementation. Notwithstanding all advantages, this approach is insufficiently studied and existing techniques for direct mapping often produce large circuits with inefficient interface to the environment.

Direct mapping can be applied at various abstraction levels resulting in different properties of the obtained designs. For example, in a syntax-driven translation method, implemented in Balsa [6] and Tangram [12], direct mapping starts at the very high level. The language statements are translated into an intermediate circuit representation, called handshake components. The interconnects between the components are derived from the syntax of the specification captured by its parsing tree. Local peephole optimisation of handshake circuits can be applied to improve size and speed of control logic. The handshake circuits are subsequently mapped into a netlist of

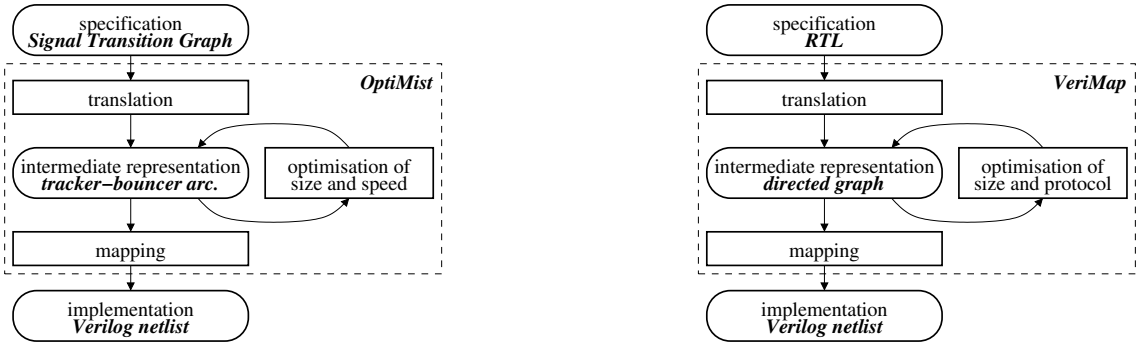
technology-dependent hardware blocks. This method is attractive from the productivity point of view, as it avoids computationally hard global optimisation of the logic. However, the translation of the parsing tree into a circuit structure may produce very slow control circuits.

On the contrary, in a Gate Transfer Level (GTL) method [98] the direct mapping is applied at a very low level of individual gates. In this method the fine-grain pipelining is employed, where gates of a standard RTL netlist are replaced by pipeline stages. Each stage contains the gate itself, a register to store the output, and a dedicated controller, which supports communication of the stage with its neighbours. The main disadvantage of the method is the excessive size of the produced circuits.

Both methods, syntax-driven translation and fine-grain pipelining, synthesise asynchronous circuits without splitting the control path from the data path. The former method is applied too early, when the control is not separated from the data path yet. The latter method is used too late, when the control and data paths are already merged in a circuit netlist. In this work the direct mapping is applied at the intermediate level of abstraction, where a circuit specification is split into internal representations for control and data paths. The separation of these paths allows to improve the desired features of each path independently of the other. For example, the control path can be optimised for low latency and size, while the data path is improved for higher security.

The design flow proposed in this thesis is based on the VeriSyn front-end [16], which is a part of the BESST design flow [101]. VeriSyn converts the initial system specification in a high-level HDL (Verilog, VHDL, System-C) into an intermediate Petri net format convenient for verification and synthesis. The Petri net specification obtained by VeriSyn is subsequently split into a Labelled PN (LPN) modelling the control path and a Coloured PN (CPN) modelling the data path of the system. These nets are optimised and synthesised separately, the produced netlists are subsequently merged into the system implementation netlist.

The direct mapping method for the control path, whose diagram is shown in Figure 1.1(a), uses a Signal Transition Graph (STG) as an initial specification. The STG is obtained by refining the LPN specification to the low level of signal events. The STG is, firstly, split into a device and an environment, which synchronise via a communication net that models wires. The device is then represented as a tracker and a bouncer. The tracker follows the state of the environment



(a) Direct mapping of control path

(b) Direct mapping of data path

Figure 1.1: Asynchronous circuit synthesis by direct mapping

and provides reference points to the device outputs. The bouncer interfaces to the environment and generates output events in response to the input events according to the state of the tracker. This two-level architecture provides an efficient interface to the environment and is convenient for subsequent mapping into a circuit netlist. A set of optimisation heuristics is developed to reduce the latency and size of the control circuit.

The direct mapping of the data path has two levels of granularity. At the higher level the entire data path is obtained by mapping its CPN fragments into hardware components which implement the corresponding mathematical functions. A library of hardware components is produced at the lower level of data path synthesis. Such a library can either be developed by modifying the standard RTL solutions to the asynchronous style manually, or using a logic synthesis methods [114]. The former approach is restricted by the manual intervention. The latter is still in its infancy and produces solutions that are inefficient in terms of speed and area. In this work we concentrate on an automated synthesis of the data path components by direct mapping.

The method for synthesis of the data path components, whose diagram is depicted in Figure 1.1(b), is based on a conventional RTL design flow, similar to the NCL-X approach [62]. Each data path component is first implemented in a standard RTL design flow, e.g Synopsys. The obtained RTL circuit is then converted into an intermediate representation convenient for logic optimisation. Negative logic and completion detection logic optimisations can be applied at this stage. The completion detection logic is added in order to indicate when the computation is fin-

ished. The optimised specification is mapped into a hazard-free logic using a dual-rail encoding with a return-to-spacer signalling.

A new alternating-spacer protocol with two spacers interchanging in time is proposed. The use of this protocol in a dual-rail circuit makes all gates switch per computation cycle. The potential applications of the alternating-spacer protocol are security circuits, online testing, refreshing of dynamic logic, etc. In this thesis the security application of alternating-spacer protocol is studied in depth. The switching of all gates in each computation cycle results in energy balancing. While power signature may still be data-dependent (due to process variations between transistors and interconnects), the integrated power per cycle of computation (energy) is a constant value invariant to processed data. The energy balance can be used to resist a power analysis attack based on power accumulation in some time interval (as opposed to instant power sampling). Power integration methods are of great interest when the attacker does not have direct access to the circuit and can only analyse side effects of its operation, e.g. electromagnetic emission. The effect of the alternating-spacer protocol on energy balancing is studied at the level of the individual gates circuit level, using AES benchmark.

1.3 Main contribution

The main contribution of this thesis is the following:

Direct mapping approach

The direct mapping approach is developed consistently throughout the synthesis of control and data paths. The key features of direct mapping are low algorithmic complexity and transparent correspondence between specification and implementation. These advantages are utilised in the proposed design flow.

Direct mapping in control path

A method for the direct mapping of control circuits from STGs is presented. It is based on a new two-level architecture, where a circuit is represented as a tracker and a bouncer. The tracker follows the state of the environment and is used as a reference point by the device outputs. The

bouncer interfaces the environment and generates output events in response to the input events according to the state of the tracker. This two-level architecture provides an efficient interface to the environment and is convenient for optimisation and subsequent mapping into a circuit netlist. A set of peephole optimisation heuristics is developed aiming at low latency and size reduction.

Direct mapping in data path

A method for automated synthesis of data path components based on RTL design flow is developed. In this method a conventional RTL netlist is converted into a hazard-free logic by applying a dual-rail encoding with a return-to-spacer signalling protocol. The acknowledgement of computation termination is formed by a completion detection logic. Methods improving the size and speed of a dual-rail circuit (negative logic and completion detection optimisations) are presented. A set of converters is designed for integrating a dual-rail circuit into a single-rail environment.

A new alternating-spacer protocol with two spacers, all-ones and all-zeros, interchanging in time is proposed. The use of this protocol in a dual-rail circuit makes all gates switch per computation cycle. The potential applications of alternating-spacer protocol are security circuits, online testing, refreshing of dynamic logic, etc.

Synthesis of security circuits

The influence of the alternating-spacer protocol on the circuit security is studied. Two spacers alternating in time help to balance consumed energy per operation cycle, thus making power analysis more difficult. New security metrics, energy imbalance and exposure time, are defined. These metrics are used to measure the security of individual gates and large cryptographic designs, e.g. AES benchmark. Secure flip-flops and latches supporting the alternating-spacer protocol are designed in scope of this thesis.

Design flow supported by software tools

The methods developed in this thesis have been implemented in a set of software tools, namely OptiMist for synthesis of low-latency control path, and VeriMap for synthesis of hazard-free data path. These tools are successfully integrated in the BESST design flow to provide a front-end

to high-level HDLs (Verilog, VHDL, System-C). The use of a high-level language for input of a specification allows a hardware designer to start using the design flow even without deep knowledge of the asynchronous circuit theory. The conventional HDL input and output also facilitates an interface to industrial EDA tools, e.g. Cadence, which is required for seamless timing analysis, simulation, place and routing and other architecture independent tasks.

Both software tools, OptiMist and VeriMap, were tested on a number of benchmarks. Several chips designed by using the VeriMap tool kit were fabricated and evaluated by a semiconductor company Atmel Inc. Also a secure design of AES with computable Sbox obtained by VeriMap was implemented in a chip using the fabrication facilities of Europractice.

1.4 Organisation of thesis

This thesis is organised as follows:

Chapter 1 Introduction briefly outlines the scope and contribution of the thesis.

Chapter 2 Background describes the Petri nets modelling language, defining its basic properties, subclasses and extensions. The Petri nets can play a pivotal role in future synthesis tools for self-timed systems, exhibiting advanced concurrency and timing paradigms. This role can be as important as that of a Finite State Machine (FSM) in designing clocked systems.

Chapter 3 Automated synthesis of asynchronous circuits reviews the existing methods of synthesis of self-timed circuits from high-level HDLs. Two main design approaches are reviewed: syntax-driven translation and logic synthesis. The advantages and problems of both approaches are pointed out. A new framework addressing those problems is proposed. It is based on the existing logic synthesis design flow and enriches it by a method for design of a secure, hazard-free data path, and a computationally simple technique for synthesis of low-latency control logic.

Chapter 4 Synthesis of control path presents a method for the direct mapping of control circuits from STGs. In this method a specification is converted into a form convenient for subsequent mapping into a circuit netlist. The efficient interface to the environment, which is based

on a two-level architecture, allows to achieve low and predictable latency of the circuit. The optimisation and mapping techniques presented in this chapter have low algorithmic complexity.

Chapter 5 Synthesis of data path presents a method for converting a conventional RTL data path into a hazard-free circuit. The hazard-free logic is obtained by use of a dual-rail encoding with a return-to-spacer signalling. A new alternating-spacer protocol with two spacers alternating in time is presented in this chapter. This switching discipline makes all gates switch per computation cycle, which can be used for security circuits, online testing and refreshing of dynamic logic.

Chapter 6 Synthesis of security circuits studies the effect of the alternating-spacer protocol to balancing the energy consumption of dual-rail circuits. Two security metrics, energy imbalance and exposure time, are introduced in this chapter. These metrics are used to estimate the ability of circuits switching in alternating-spacer protocol to resist power analysis attacks. For this a set of cryptographic benchmarks are studied with different switching protocols.

Chapter 7 Conclusions summarises the major results achieved in this work and points the areas for future research.

Appendix A OptiMist user manual explains how to use the OptiMist toolkit for the direct mapping of low-latency asynchronous controllers from STGs. The ASTG language, which is used by OptiMist, is also described there.

Appendix B VeriMap user manual illustrates the usage of the VeriMap tool for synthesis of a secure and hazard-free data path. It also introduces a structural Verilog language, which is used to describe circuit netlists.

Appendix C AES designs presents a chip which was designed using the VeriMap tool kit. The chip contains two implementations of Advanced Encryption Standard (AES) with computable Sboxes. The first one is a standard RTL implementation synthesised from a behavioural AES specification, and the other one is a dual-rail implementation obtained from the RTL netlist by using VeriMap.

Chapter 2

Background

This chapter provides an introduction to asynchronous circuits, their delay models, operation modes, classes and common signalling protocols. A behavioural Petri nets model which is widely used for specification, verification and synthesis of asynchronous circuits is also presented in this chapter.

2.1 Asynchronous circuits

A category of circuits containing no global clock is called *asynchronous circuits [116]*. These circuits may make use of timing assumptions both within the circuit and in its interaction with the environment. Based on these assumptions the asynchronous circuits can be divided into several classes. This section overviews the classes of asynchronous circuits using a classification presented in [63, 35].

2.1.1 Delay models

An asynchronous circuit can be considered as an interconnection of two types of components, *gates* and *delay elements*, by means of *wires*. A gate computes a set of output variables (often a single output variable) as a discrete logical function of its input variables. A delay element produces a single output that is a delayed version of its input. Each wire connects an output of a single gate or delay element to inputs of one or more gates or delay elements. Primary inputs and outputs of a circuit can be considered as gates computing the identity function.

There are two major models of a delay element: *pure delay* model and *inertial delay* model. A pure delay element transmits each signal event on its input to its output with some delay regardless the shape of the signal's waveform. On the contrary, an inertial delay element alters the shape of its input waveform by attenuating short pulses, i.e. it filters out pulses of a duration less than some threshold period.

The delay elements are also characterised by their timing models. In a *fixed delay* model, a delay is assumed to have a fixed value. In a *bounded delay* model, a delay may have any value in a given timing interval. In an *unbounded delay* model, a delay may take an arbitrary finite value.

2.1.2 Operation modes

An interaction of a device circuit with its environment can be characterised by circuit operation mode. The *device* and its *environment* together form a *close system*. If the environment is allowed to respond to a device's outputs without any timing constraints, the system is said to interact in *input-output mode*. Otherwise, environmental timing constraints are assumed. The most common example is *fundamental mode* where the environment must wait for the device to stabilise before producing new inputs.

Depending on the restrictions to the input changes, the fundamental mode is divided into several subclasses. If a single input is allowed to change at a time, the operation mode is called *Single Input Change (SIC)* fundamental mode. SIC mode forces the inputs to be sequential, which may restrict the speed of circuit operation. Another approach which allows one or more inputs to change after the circuit stabilisation, is called *Multiple Input Change* (MIC) fundamental mode. The speed of a circuit operating in this mode improves compared to SIC mode, however it may be difficult to implement a circuit operating MIC mode.

A trade-off between SIC and MIC fundamental modes is a *Burst Mode* (BM) which only allows inputs to change in groups, called bursts. Inputs in a burst may arrive in any order and at arbitrary time. A set of inputs in a burst cannot be a subset of another burst. This restriction helps a circuit to distinguish bursts one from another. The circuit waits until all inputs in a burst change before producing its outputs. The outputs must be allowed to settle before another input burst starts.

2.1.3 Classes of asynchronous circuits

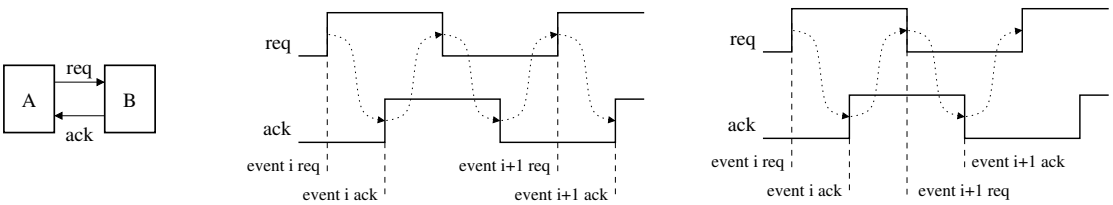
The most obvious model to use for asynchronous circuits is the same as for synchronous circuits. This model is followed in *Huffman* circuits [52], which are designed to work correctly in the fundamental mode of operation. A bounded delay is assumed for both gates and wires.

Delay-Insensitive (DI) circuits are designed to operate correctly in input-output mode with unbounded gate and wire delay. These circuits are most robust with respect to manufacturing processes and environmental variations. The concept of delay-insensitive circuits originates from [27] and is formalised in [115]. The class of DI circuits built out of simple gates is quite limited. It has been proven that almost no useful DI circuits can be built if one is restricted to a class of simple gates [68]. However, many practical DI circuits can be built using complex gates [38]. A complex gate is constructed out of several simple gates. Externally a complex gate operates in a delay-insensitive manner, however internally it may rely on some timing assumptions.

In order to build practical circuits out of simple gates a relaxation of the requirements to the DI circuits is necessary. This can be achieved by introducing an *isochronic fork*, which is a forked wire where the difference in delays between the branches is negligible [9]. Asynchronous circuits with isochronic forks are called *Quasi-Delay-Insensitive* (QDI) circuits [67]. In contrast, in DI circuits, delays on the different fork branches are completely independent, and may vary considerably.

Speed-Independent (SI) circuits are guaranteed to work correctly in input-output mode regardless of gate delays, assuming that wire delays are negligible. This means that whenever a signal changes its value all gates it is connected to will see that change immediately. SI circuits introduced in [77] only considered deterministic input and output behaviour. This class has been extended to include circuits with a limited form of non-determinism in [8].

Self-timed circuits, described in [95], are built out of a group of elements. Each element may be an SI circuit, or a circuit whose correct operation relies on local timing assumptions. However, no timing assumptions are made on the communication between elements and the circuit operates in input-output mode. If both internal and external timing assumptions are used to optimise the designs, then such circuits are called *timed* [82].



(a) Handshake (b) Four-phase protocol (c) Two-phase protocol

Figure 2.1: Signalling protocols

2.1.4 Signalling protocols

Asynchronous circuit signalling schemes are based on a protocol called *handshake*, involving *requests*, which are used to initiate an action, and corresponding *acknowledgements*, used to signal completion of that action. These control signals provide all of the necessary sequence controls for computational events in the system.

For example, consider an interaction of two modules, a sender A and a receiver B shown in Figure 2.1(a). A request is sent from A to B indicating that A is requesting some action from B. When B completes the action, it acknowledges the request by sending an acknowledge signal from B to A. Most asynchronous signalling protocols require a strict alternation of request and acknowledge events. These ideas can be extended to interfaces shared by more than 2 subsystems. There are several ways of how these alternating events are encoded onto specific control wires. The most commonly used handshake protocols are the *four-phase* and *two-phase*.

The *four-phase protocol*, also called *return-to-zero*, is shown in Figure 2.1(b). The dashed arrows indicate the causality of the events. There are no implicit assumptions about the delay between successive events. In this protocol there are four signal transitions (two on the request and two on the acknowledgement) required to complete a particular event transition.

The *two-phase protocol*, also called *non-return-to-zero*, is shown in Figure 2.1(c). The waveforms are the same as for four-phase signalling with the exception that every transition on the request wire, both falling and rising, indicates a new request. The same is true for transition on the acknowledgement wire.

Typically four-phase circuits are smaller than they are for two-phase signalling. The time required for the falling transition on the request and on the acknowledge lines dos not usually cause

performance degradation because these transitions happen in parallel with other circuit operations. Two-phase signalling is better from both a power and a performance standpoint, since every transition represents a meaningful event and no power is consumed for resetting of the handshake link. Whilst this is true, in principle, it is also the case that most two-phase interface implementations require more logic than four-phase equivalents.

So far, only control signalling is addressed. There are also different ways of data encoding. A common choice is the use of a *bundled-data* protocol with either two-phase or four-phase signalling. This protocol requires $n + 2$ wires (n bits of data, a request bit, and an acknowledge bit), to pass an *n*-bit data value from a sender to a receiver. While this choice is conservative in terms of wires, it does contain an implied timing assumption. Namely the assumption is that the propagation of data signals is not faster than propagation of the control signals.

The common alternative to the bundled-data approach is *dual-rail* encoding. In this case, data and control signals are not separated onto distinct wire paths. Instead, using the dual-rail approach, a bit of data is encoded with its own request onto 2 wires. A typical dual-rail encoding has four states: *00* - data is not valid, *10* - valid 0, *01* - valid 1, *11* - illegal. In this case, for an n-bit data value, the link between sender and receiver must contain $3 \cdot n$ wires (2 wires for each bit of data and the associated request plus another bit for the acknowledge). An improvement on this protocol is possible when n-bits of data are considered to be associated in every transaction, as is the case when the circuit operates on bytes or words. In this case it is convenient to combine the acknowledges into a single wire. The resulting wiring complexity is then reduced to $2 \cdot n + 1$ wires ($2 \cdot n$ wires for the data and requests plus an additional acknowledge signal).

In a four-phase variant of this dual-rail protocol, sending a bit requires the transition from the idle state to either the valid 0 or valid 1 state and then, after receiving the acknowledge, it must transition back to the idle state. The acknowledge wire must be reset prior to a subsequent assertion of a valid 0 or 1. The illegal state is not used.

A two-phase dual-rail protocol would signal a valid 0 by a single transition of one bit, while a valid 1 would be signalled by a transition on another bit. Concurrent transitions on both the left and right bits are illegal. Sending a 0 or a 1 must be followed by a transition on the acknowledge wire before another bit can be transmitted.

Dual-rail signalling is insensitive to the delays on any wire and therefore is more robust when bundled-data timing assumptions cannot be guaranteed. The receiver will need to check for validity of all n-bits before using the data or asserting the acknowledge. The downside of the dual-rail approach is often the increased complexity in both wiring and logic.

There exist other communication protocols such as 1-of-4 [2] or even *1-of-n* encodings [42, 43] used in control logic and higher-radix data encodings. If the focus is on communication rather then computation, *m-of-n* encodings [3] may be of relevance.

2.2 Behavioural models

This section introduces the formal models used for the specification and verification of asynchronous circuits. First, the basic concept of Petri nets (PNs) model is presented. PNs extend the Finite State Machines (FSMs) model with a notion of concurrency, which makes them especially convenient for the specification and verification of asynchronous circuits. The formal definitions and notations in this section are based on the work introduced in [30, 78, 84, 90].

2.2.1 Petri nets

A Petri nets model, first defined in [88], is a graphical and mathematical representations of discrete distributed systems. Petri nets are used to describe and study concurrent, asynchronous, distributed, parallel and non-deterministic systems. As a graphical tool, PNs can be used as a visual communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is allows to set up state equations, algebraic equations, and other mathematical models governing the behaviour of systems.

A *Petri Net* (PN) is formally defined as a tuple $PN = \langle P,\, T,\, F,\, M_0 \rangle$ comprising finite disjoint sets of *places* P and *transitions* T, *arcs* denoting the flow relation $F \subseteq (P \times T) \cup (T \times P)$ and *initial marking* M_0.

There is an arc between $x \in P \cup T$ and $y \in P \cup T$ iff $(x, y) \in F$. An arc from a place to a transition is called *consuming arc*, and from a transition to a place - *producing arc*. The *preset* of a node $x \in P \cup T$ is defined as $\bullet x = \{y \mid (y, x) \in F\}$, and the *postset* as $x\bullet = \{y \mid (x, y) \in F\}$.

It is assumed that $\bullet t \neq \emptyset \neq t\bullet$, $\forall\, t \in T$. The *pre-preset* of a node $x \in P \cup T$ is defined as $\bullet\bullet x = \bigcup_{y \in \bullet x} \bullet y$, and the *post-postset* as $x \bullet\bullet = \bigcup_{y \in x\bullet} y\bullet$.

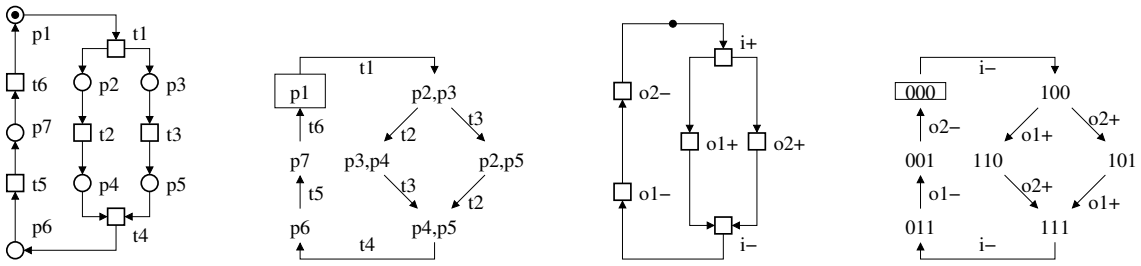
A place p such that $|p\bullet| > 1$ is called *choice place*, i.e. it has more than one transition in its postset. A choice place p is called *free choice* if $\forall t \in p\bullet : |\bullet t| = 1$, i.e. each transition in its postset has exactly one preset place. A choice place p is called *controlled choice* if $\exists t \in p\bullet : |\bullet t| > 1$, i.e. there is at least one transition in its postset which has more than one preset place. Note that a controlled choice whose all postset transitions have the same preset places can be transformed into a free choice. A place p such that $|\bullet p| > 1$ is called *merge place*. A transition t such that $|t\bullet| > 1$ is called *fork* and a transition t such that $|\bullet t| > 1$ is called *join*.

The dynamic behaviour of a PN is defined as a *token game*, changing markings according to the enabling and firing rules its transitions. A *marking* is a mapping $M : P \rightarrow \mathbb{N}$ denoting the number of *tokens* in each place, $\mathbb{N} = \{0, 1\}$ for *1-safe* PNs. A transition t is *enabled* iff $M(p) > 0$, $\forall\, p \in \bullet t$. The evolution of a PN is possible by *firing* the enabled transitions. *Firing* of a transition t results in a new marking M' such that $\forall p \in P : M'(p) = \begin{cases} M(p) - 1 & if\ p \in \bullet t, \\ M(p) + 1 & if\ p \in t\bullet, \\ M(p) & otherwise \end{cases}$, i.e. for an enabled transition t one token is removed from each preset place and one token is produced to each postset place.

A marking M' is *reachable* from a marking M if there exists a *firing sequence* $\sigma = t_0 \dots t_n$ starting at marking M and finishing at M'. A set of reachable markings from M is denoted by $[M\rangle$. A set of markings reachable from the initial marking M_0 is called a *reachability set* of a PN.

The set of markings reachable in a PN from its initial marking can be represented as a reachability graph, whose nodes are labelled with PN markings and arcs are labelled with PN transitions. Formally, a *Reachability Graph* (RG) of a $PN = \langle P, T, F, M_0 \rangle$ is a labelled directed graph $RG = \langle S, A, l, s0 \rangle$, where $S = [M_0\rangle$ is a *reachability set*, $A = S \times T \times S$ is a set of *arcs* between these states, $l : A \rightarrow T$ is a *labelling function* indicating transitions between markings, and s_0 is the *initial state* corresponding to the initial marking of the PN.

Graphically, places of a PN are represented as circles (○), transitions as boxes (□), consuming and producing arcs are shown by arrows (⟶), and tokens of the PN marking are depicted by dots in the corresponding places (⦿). A simple PN is shown using this graphical notation in



(a) PN (b) RG (c) STG (d) SG

Figure 2.2: Simple examples of PN, RG, STG and SG

Figure 2.2(a). This example illustrates that, unlike FSM model, PNs model can capture concurrent actions. If two transitions are enabled in the same marking and the firing of one does not interfere with the enabling of the other, then both transitions will eventually fire. The fact that transitions *t2* and *t3* are concurrent means that both firing sequences *t2, t3* and *t3, t2* are possible. This is captured by the RG in Figure 2.2(b). The RG nodes are labelled with the reachable PN markings, arcs are labelled with the corresponding PN transitions and its initial state is marked with a box.

Transitions in a PN can be involved in different ordering relations. Two PN transitions are in *direct conflict* if there exists a reachable marking in which both of them are enabled but firing of one of them disables the other. *Conflict* relations can be generalised by considering the transitive successors of directly conflicting transitions. If two transitions are enabled in some reachable marking but are not in direct conflict, they are *concurrent*. Transitions which are not concurrent and are not in a transitive conflict are *ordered*.

Important properties of a PN are safeness, liveness and deadlock-freeness. A PN is said to be k-*bounded* if the number of tokens in every place of a reachable marking does not exceed a finite number k. A 1-bounded PN is also called *1-safe*. A PN is *deadlock-free* if, no matter what marking has been reached, it is possible to fire at least one transition of the net. A PN is *live* if for every reachable marking M and every transition t it is possible to reach a marking M' that enables t.

An extension of a PN model is a *contextual net* [73]. It uses additional elements such as *non-consuming arcs*, which only control the enabling of transitions and do not influence their firing. A PN extended with a type of non-consuming arcs, namely *read-arcs,* is defined as $PN =$

$\langle P, T, F, R, M_0 \rangle$. A set of read-arcs R is defined as $R \subseteq (P \times T)$, there is a read-arc between p and t iff $(p,t) \in R$. The *read-preset* of a transition $t \in T$ is defined as $\star t = \{p \mid (p,t) \in R\}$, and the *read-postset* of a place $p \in P$ as $p\star = \{t \mid (p,t) \in R\}$. Place p *controls* transition t by means of a read-arc iff $p \in \star t$. A transition t *reads* the state of a place p iff $t \in p\star$. A transition t is enabled iff $M(p) \neq 0, \forall\, p \in \bullet t \cup \star t$. The rules for firing of the transitions are preserved. A read-arc is depicted as a line without arrows.

The following are three most common subclasses of PNs. A PN is called a *Marked Graph* (MG) iff $\forall p \in P : |\bullet p| \leq 1 \wedge |p\bullet| \leq 1$, i.e. each place has at most one preset and one postset transition. The nets of this subclass represent deterministic concurrent systems. Dually, a PN is called a *State Machine* (SM) iff $\forall t \in T : |\bullet t| = 1 \wedge |t\bullet| = 1$, i.e. each transition has exactly one preset and one postset place. This subclass allows to represent non-deterministic sequential systems. A PN is called *Free Choice* (FC) net iff for any choice place p $\forall t \in p\bullet : |\bullet t| = 1$, i.e. each transition in the postset of a choice place has exactly one preset place. Free choice nets model both non-determinism and concurrency but restricts their interplay. The former is necessary for modelling choice made by the environment whereas the latter is essential for asynchronous behaviour modelling.

The two modelling extensions of PNs are Labelled PNs and Coloured PNs. A *Labelled Petri Net* (LPN) is a PN whose transitions are associated with a labelling function [123]. The extension of non-consuming arcs is also applicable to the LPN definition. A *Coloured Petri Net* (CPN) is a formal high-level net where places are associated with data types, tokens are associated with the data values and transitions denote the operations on that data [55]. This allows the representation of data path in a compact form, where each token is equipped with an attached data value.

2.2.2 Signal transition graphs

The Signal Transition Graph (STG) model was introduced independently in [26] and [91] to formally model both the circuit and the environment. The STG can be considered as a formalisation of the widely used timing diagrams. It describes the causality relations between transitions on the input and output signals of a specified circuit. It also allows the explicit description of data-

dependent choices between various possible behaviours. STGs are interpreted Petri nets, and their close relationship to Petri nets provides a powerful theoretical background for the specification and verification of asynchronous circuits.

An STG is a 1-safe LPN whose transitions are labelled by signal events, i.e. $STG = \langle P, T, F, M_0, \lambda, Z, v_0 \rangle$, where λ is a *labelling function*, Z is a set of *signals* and $v_0 = \{0, 1\}^{|Z|}$ is a *vector of initial signal values*.

The set of signals Z is divided into two disjoint sets of *input signals* Z_I and *output signals* Z_O, $Z = Z_I \cup Z_O$, $Z_I \cap Z_O = \emptyset$. Input signals are assumed to be generated by the environment, whereas output signals are produced by the logic gates of the circuit. *Internal signals* may also be included in the Z_O set.

The labelling function $\lambda : T \rightarrow Z\pm \cup \Theta$ maps transitions into *signal events* $Z\pm = Z \times \{+, -\}$ and *dummies* Θ, $Z \pm \cap \Theta = \emptyset$. The signal events labelled $z+$ and $z-$ denote the transitions of signals $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. Dummy transitions are *silent events* that do not change the state of any signal. The labelling function does not have to be 1-to-1, i.e. transitions with the same label may occur several times in the net. In order to distinguish between transitions with the same label and refer to them from the text an index $i \in \mathbb{N}$ is attached to their labels as follows: $\lambda(t)/i$, where i differs for different transitions with the same label.

In order to be implementable as a circuit an STG must satisfy the property of consistency. An STG is *consistent* if for each signal $z \in Z$ transitions labelled $z-$ and $z+$ alternate in any firing sequence starting from M_0. In this work it is assumed that all the considered STGs are consistent.

A *vector of signal change* $v_\sigma = \left(v_\sigma^1, ..., v_\sigma^{|Z|}\right)$ can be associated with a finite sequence of transitions σ, so that each v_σ^i is the difference between the number of rising and falling edges of signal z_i in σ. The *vector of signal values* $v = v_0 + v_\sigma$ defines the states of all STG signals after some sequence of transitions σ. Note that for consistent STGs the vectors v_0, $v_\sigma$and v are binary.

A *projection* of a firing sequence σ onto a set of signals $X \subseteq Z$ is defined as $\sigma \downarrow X = \{t \in \sigma : \lambda(t) \in X\pm\}$, i.e. it only includes transitions of signals in X. A *silent sequence* θ is a firing sequence (possibly empty) such that $\theta \downarrow Z = \emptyset$, i.e. its projection on the set of signals is empty. Similarly, a firing sequence whose the projection on the set of output (input) signals is empty and

projection on the set of input (output) signals is not empty is called *input (output) sequence*.

STGs inherit the operational semantics of their underlying PNs, including the notations of transition enabling and firing. Likewise, STGs also inherit the various structural (marked graph, free-choice, etc.) and behavioural properties (boundedness, liveness, etc.). Note that a set of read-arcs can be included into the model of STG, which is an enhancement w.r.t. [91].

For graphical representation of STGs a short-hand notation is often used, where a transition can be connected to another transition if the place between those transitions has one incoming and one outgoing arc as illustrated in Figure 2.2(c).

A state of an STG without dummies is a pair $\langle M,\, v\rangle$, where M is a reachable marking and v is vector of signal values corresponding to this marking. Note that the vector of signal values along does not uniquely identify the STG state. Also in general case the same marking can correspond to different states of the STG (e.g., if it is not live or not consistent). The extension of an STG with the notion of dummy transitions complicates the definition of its state. As a dummy transition does not correspond to a signal event, firing of a dummy does not actually change the state of the system described by the STG. That is why the states of an STG before and after firing a dummy are considered equal, though the marking is different. In this work we assume that a dummy (or in a more general case a silent sequence) is a part of preceding signal transition.

In the same way as an STG is an interpreted PN with transitions associated with binary signals, a state graph is the corresponding binary interpretation of an RG in which the events are interpreted as signal transitions. Formally, a *State Graph* (SG) of an $STG = \langle P,\, T,\, F,\, M_0,\, \lambda,\, Z\rangle$ is a quadruple $SG = \langle S,\, A,\, l,\, C,\, s_0\rangle$, where S is a set of *reachable states*, $A = S \times T \times S$ is a set of *arcs* between these states, $l :\ A \rightarrow T$ is a *labelling function* for the arcs, $C :\ S \rightarrow \{0,\, 1\}^{|Z|}$ is a *state assignment function*, which is defined as $C\left(\langle M,\, v\rangle\right) = v$, and $s_0 = \langle M_0,\, v_0\rangle$ is the *initial state*. Note that dummies are

The SG of the STG in Figure 2.2(c) is shown in Figure 2.2(d). Each SG state corresponds to a marking of the STG and is assigned a binary vector. Each SG arc corresponds to firing of a signal transition. For readability the SG arcs are indicated by the transition labels. The signal order in the binary vectors is $\langle i,\, o1,\, o2\rangle$. The initial state (marked with a box) corresponds to the marking $\{p1\}$ with the signal values vector 000.

A property of an STG which simplifies its hardware implementation is persistency. An STG is *persistent* if no transition can be disabled by another transition unless they both are events of different input signals. This means that all non-deterministic behaviour is part of the environment and the arbitration is avoided in the device.

Properties of an STG specific for a logic synthesis approach are *unique state coding* and *complete state coding*. The former is sufficient condition and the latter is necessary condition of a circuit implementability by logic synthesis. Two distinct states of an SG are in a *Unique State Coding* (USC) conflict if they are assigned to the same code. Two distinct states of a SG are in a Complete State Coding (CSC) conflict if they are assigned to the same code and the set of enabled output signals is different in these states. An STG satisfies the USC (CSC) property if no two states of its SG are in USC (CSC) conflict. Note that neither USC nor CSC is required in a direct mapping approach. The properties of an STG which are specific for the proposed direct mapping method are considered in Section 4.1.1.

2.2.3 Bisimulation

Bisimulation, originally introduced in [49, 72], is an equivalence relation between STGs, associating systems which behave in the same way, in the sense that one system simulates the other and vice-versa. Intuitively two systems are bisimular if they match each other's moves, i.e. each of the systems cannot be distinguished from the other by an observer.

Two systems described by $STG = \langle P, T, F, R, M_0, \lambda, Z, v_0 \rangle$ and $STG' = \langle P', T', F', R', M'_0, \lambda', Z', v'_0 \rangle$ are *(strongly) bisimular*, notation $STG \sim STG'$, iff:
(i) $M_0 \sim M'_0$;
(ii) if $M \sim M'$ and $M\,[t\rangle\,M_1$ then $\exists\, t' \in T'$ such that $\lambda\,(t) = \lambda'\,(t')$, $M'\,[t'\rangle\,M'_1$ and $M_1 \sim M'_1$;
(iii) as (ii) but with roles of STG and STG' reversed.

The notion of strong bisimulation requires a system to be capable of matching each transition that an equivalent system may perform. However, sometimes internal and external (observable) behaviour of a system are distinguished. In this sense two systems are equivalent if they exhibit the same external behaviour, irrespective of any intermediate internal behaviour that may occur. For example, if the system STG includes a notion of silent actions (dummies), then bisimulation

Figure 2.3: Observation bisimulation

can be relaxed to ignore these dummies.

Two systems represented by $STG = \langle P, T, F, R, M_0, \lambda, Z, v_0 \rangle$ and $STG' = \langle P', T', F', R', M_0', \lambda', Z', v_0' \rangle$ are *weakly (observationally) bisimular*, notation $STG \approx STG'$, iff:

(i) $M_0 \approx M_0'$;

(ii) if $M \approx M'$ and $M\,[t\rangle\,M_1$ then either $\lambda(t) = \tau$ and $M_1 \approx M'$ or $\exists\, t' \in T'$, $\lambda(t) = \lambda'(t')$ and silent sequences θ_1, θ_2 such that $M'\,[\theta_1\rangle\,M'_{\bullet t}\,[t'\rangle\,M'_{t\bullet}\,[\theta_2\rangle\,M'_1$ and $M_1 \approx M'_1$;

(iii) as (ii) but with roles of STG and STG' reversed.

Still, the notion of weak bisimulation cannot be regarded as the natural generalisation of strong bisimulation for STGs with silent events. The reason for this is that an important feature of bisimulation is missing for weak bisimulation. Namely the property that any firing sequence in one STG corresponds to a firing sequence in the other, in such a way that all intermediate states of these STGs correspond as well. However, according to the definition of the weak bisimulation one may fire arbitrary many silent transitions in an STG without worrying about the markings that are passed through in the meantime. For example, the STGs in Figure 2.3 are weakly bisimilar, however in the right STG there is a trace which does not enable the $b+$ transition, while $b+$ is enabled in all traces of the left STG. Thus, the observational equivalence does not preserve the branching structure of STGs and hence lacks one of the main characteristics of bisimulation semantics.

An alternative definition of observational equivalence which preserves the branching structure of STGs was proposed in [45]. This equivalence, called *branching bisimulation*, requires all intermediate markings in silent sequences θ_1 and θ_2 of STG' to be related with markings M and M_1 of STG respectively. Note that STGs in Figure 2.3 are not branching bisimilar.

Obviously, branching bisimulation is stronger than weak bisimulation. However, one can see that for persistent STGs weak bisimulation becomes equivalent to branching bisimulation.

Chapter 3

Automated synthesis of asynchronous circuits

In 1965 a co-founder of Intel Gordon Moore noticed that the number of transistors doubled every year since the invention of the integrated circuit. He predicted that this trend would continue for the foreseeable future [74]. In subsequent years the pace slowed down and now the functionality of the chip doubles every two years [75]. However, the growth of circuit integration level is still faster than the increase in the designers productivity. This creates a design gap between semiconductor manufacturing capability and the ability of Electronic Design Automation (EDA) tools to deal with the increasing complexity [39], Figure 3.1.

One of the ways to deal with the increasing complexity of logic circuits is to improve the

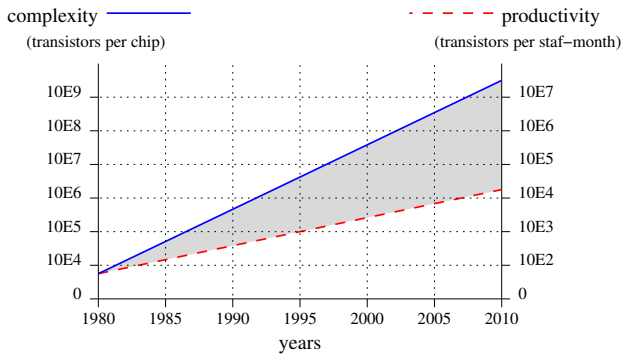


Figure 3.1: Design complexity and designer productivity

efficiency of the design process. In particular, design automation and component reuse help to solve the problem. Systems-on-Chip (SoC) synthesis has proved to be a particularly effective way in which design automation and component reuse can be facilitated. An important role in the synthesis of SoCs is given to the aspects of modelling concurrency and timing [54]. These aspects have traditionally been dividing systems into *synchronous* (or *clocked*) and *asynchronous* (or *self-timed*). The division has recently become fuzzier because systems are built in a mixed timing style: partly clocked and partly self-timed. The argument about the way how the system should be constructed, synchronously or asynchronously, is moving to another round of evolution. It is accepted that the timing issue should only be addressed in the context of the particular design criteria, such as speed, power, security, modularity, etc. Given the complexity of the relationship between these criteria in every single practical case, the design of an SoC is increasingly going to be a mix of timing styles. While industrial designers have a clear and established notion of how to synthesise circuits with a global clock using EDA tools, there is still a lot of uncertainty and doubt about synthesis of asynchronous circuits. The latter remains a hot research field captivating many academics and graduate students. In the last two decades there have been dozens of research publications on asynchronous circuit synthesis, and it would be impossible to embrace them all in a single review. Readers without prior experience are invited to study them at an introductory level (e.g. [48] and *http://www.cs.man.ac.uk/async/background/*) while the more experienced audience can delve into such methods in more detail by addressing monographs and papers (e.g. [31, 81] and *http://www.cs.man.ac.uk/async/pubwork/*).

The main goal of this chapter is to review a coherent subset of synthesis methods for self-timed circuits based primarily on a common underlying model of computation and using a relatively simple example in which these methods can be compared. Such a model is Petri nets, used with various interpretations. The Petri nets can play a pivotal role in future synthesis tools for self-timed systems, exhibiting advanced concurrency and timing paradigms. This role can be as important as that of a Finite State Machine (FSM) in designing clocked systems. To make this review more practically attractive the use of Petri nets is considered in the context of a design flow with a front-end based on a hardware description language.

The rest of the chapter is organised as follows. Firstly, the advantages and drawbacks of

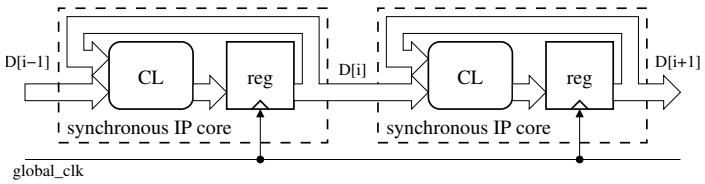


Figure 3.2: Synchronous system architecture

common system architectures are named in Section 3.1. An overview of syntax-driven design flows is given in Section 3.3 and the logic synthesis methods are discussed in Section 3.4. Then, the tools for synthesis of control and data paths are reviewed in Sections 3.5-3.8. Finally, the state of the asynchronous circuits design automation is summarised and the ways of improvement are pointed out in Sections 3.9-3.11 .

This chapter is based on results developed in [107]. The BEhavioural Synthesis of Self-Timed Systems (BESST) design flow proposed as a result of this review in Section 3.24, has been presented in [101].

3.1 System architectures

3.1.1 Synchronous systems

Most of the SoCs produced by industry are synchronous. The components of a *synchronous system* share a discrete notion of time determined by a global clock signal, see Figure 3.2. The traditional design flow for synchronous systems is supported by EDA tools, e.g. Cadence, Mentor Graphics, Synopsys, etc. However, a globally clocked SoC assembled from existing Intellectual Property (IP) cores suffers from several drawbacks.

The first problem with the clocked SoCs is the *timing closure*. Each IP core is designed for a certain clock period, assuming that the clock signal is delivered at the same time to all parts of the system. Finding a common clocking mode for the whole system is a very complex obstacle on the way to component reuse.

The difference in arrival times of the clock signal to IP cores is also difficult to avoid. This phenomenon called *clock skew* is caused by interconnect delays. In the past the transistors were the limiting factor of the circuit speed. The increase of the circuit integration level resulted in the

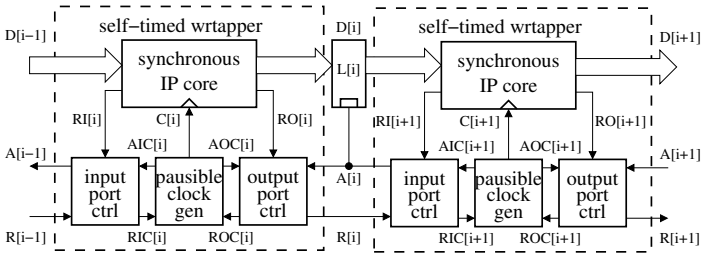


Figure 3.3: GALS system architecture

improvement of the transistor size and speed. However, the interconnect speed is not keeping the pace. Narrower wires have higher resistance for the same length, leading to slower signal edges and longer interconnect delays. Being proportional to interconnect delay, the clock skew becomes an increasing portion of the clock period. This means that eventually large circuits will need to get rid of the global clocking in order to provide high speed.

3.1.2 Globally asynchronous locally synchronous systems

A promising method of composing systems from predesigned components is *Globally Asynchronous Locally Synchronous* (GALS) approach [22]. In such systems the components are synchronous IP cores operating at their own clock speed, which allows the proven synchronous design methodologies to be employed. The interface between the components is converted to asynchronous style by putting them inside self-timed wrappers, as shown in Figure 3.3. This eliminates the need for a global clock with all of its associated problems.

The GALS self-timed wrapper, whose basic structure is captured in Figure 3.3, is proposed in [79]. It contains a pausible clock generator and an asynchronous controller for each port. The data lines between two GALS modules are bundled with a pair of request-acknowledge signals. Any data transfer is initiated by the locally synchronous island on the transmitting side by activating the $RO[i]$ request to the output port controller. The output port controller in turn instructs the clock generator to delay the next clock edge by using the $ROC[i]$ request. After the clock of the module has been frozen, the local clock generator acknowledges it by the $AOC[i]$ signal. Then the communication partner is notified by the request signal $R[i]$. Once the other GALS module has halted its clock using $RIC[i+1]$ and $AIC[i+1]$ handshake, it sets the acknowledge port signal

$A[i]$ and enables the input buffer latch $L[i]$. At this point, both modules have halted their clocks and can exchange data without any risk of timing violations. Once the data transfer is complete, the local clocks are released and the locally synchronous islands continue to operate in a normal synchronous mode.

Note an important timing assumption on the output port controller. The request $ROC[i]$ for clock pausing must be issued in the same clock cycle when $RO[i]$ signal is received from the synchronous island. This is necessary to prevent generation of an additional clock edge before the data transfer. High-speed IP cores may have difficulty with this assumption.

The greatest advantage of the GALS system architecture is the reuse of the existing synchronous IP cores and the employment of the conventional EDA tools for design and verification of new IP cores. Being able to run SoC components at different frequencies, GALS systems also contribute to power savings.

However, GALS systems have their own drawbacks, for example metastability problem, when an asynchronous signal is sampled by a clock. In order to avoid metastability several methods are used.

One of the ways to minimise the probability of metastability is to path each asynchronous signal through a synchroniser, which is typically a pair of back-to-back connected flip-flops. Still, in a GALS system the number of connections between its synchronous blocks is large, which creates a non-negligible probability of system failure. The synchronisers also add extra latency to the signals which significantly impacts the system performance.

The other strategy to avoid metastability is the dynamic alteration of the local clock rate [126, 80]. For this, a *pausible clock* generator is employed in each synchronous island. The clock generator is a ring oscillator with a control input for its stopping and starting. If some asynchronous channel of the synchronous island is not ready, then the inactive phase of the local clock is stretched until all channels are ready.

Several methods to ensure that metastability never occurs in a GALS system with pausible clocking are proposed in [37]. However, the alteration of the local clock may cause a deadlock when all components are waiting for the output of some other component. It is not trivial to guarantee that the system is deadlock free. It should be also noted, that pausing the local clock

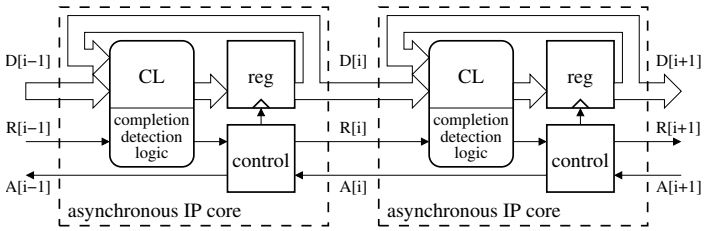


Figure 3.4: Asynchronous system architecture

slows down the entire synchronous island, and the slowdown may be exacerbated with multi-port GALS modules, where the probability of pausing the clock is higher. Furthermore, the local clock alteration may cause problems if the dynamic logic is used, as the length of the clock period becomes important there (as opposed to static logic). Finally, the ring oscillators which are used to form the pausible clocks (as opposed to crystal oscillators) suffer from significant jitter and frequency variation, which may result in a performance degradation.

There also other approaches to the design of GALS systems, such as module stalling via clock gating [19, 97] and fine-grain synchronous handshaking [83, 87, 53].

3.1.3 Asynchronous systems

The GALS approach minimises the designer work in the asynchronous domain, but does not completely escape it. Some of the components, particularly self-timed wrappers, are still asynchronous. At the same time, purely *asynchronous systems* whose architecture is shown in Figure 3.4 offer a set of qualitative advantages which GALS systems do not have. In addition to better modularity and avoidance of clock distribution problem, self-timed systems can exhibit higher robustness, greater performance, power saving, lower electromagnetic noise, etc. [35, 48, 56, 108, 7, 69]

The major obstacle on the mainstream use of asynchronous design techniques is the lack of a coherent design flow, compatible with conventional EDA tools and libraries. The compatibility issue is essential because a large part of the design flow is the mapping of the circuit netlist into silicon. For this task the traditional place and route tools can be reused. It is also possible to inherit the timing analysis and simulation tools. However, synthesis and verification tools intended for synchronous systems omit important features of asynchronous components. These parts of the

traditional design flow have to be replaced.

The other impediment is that industry adheres to existing specification languages. The majority of industry designers think in terms of high-level Hardware Description Languages (HDL), such as System-C, Verilog and VHDL, which were created for synchronous designs. These languages require much more code to be written in order to specify an asynchronous component, compared to synchronous logic. Several new languages were developed for efficient asynchronous design [12, 4, 36]. However, adoption of a unique language in industry involves significant changes in the design flow and retraining the designers. These procedures are extremely costly and take valuable time, which makes the new languages difficult to accept for commercial companies.

Finally, all existing synchronous IP cores have to be abandoned in the asynchronous world. It will take years before all those components are replaced by asynchronous counterparts.

Even though the asynchronous techniques involve significant changes to the conventional design flow, the companies realise that this is the promising route to cover the design productivity gap. Such industry giants as IBM, Infineon, Intel, Philips, Sun etc. invest in synthesis and verification tools for asynchronous circuit design. They also replace parts of their new systems by asynchronous components, gradually replenishing design libraries with asynchronous IP cores.

3.2 Asynchronous circuit design flows

For a designer it is convenient to specify the circuit behaviour in a form of a high-level HDL, such as Verilog, VHDL, SystemC, Balsa, etc. The choice of HDL is based on personal preferences, EDA tool availability, commercial, business and marketing issues [100].

There are two main approaches to synthesis of asynchronous circuits from high-level HDLs: *syntax-driven translation* and *logic synthesis*.

In *syntax-driven translation* the language statements are mapped into circuit components and the interconnect between the components is derived from the syntax of the system specification. This approach is adopted by Tangram [12, 86, 57] and Balsa [5, 4, 6] design flows. The initial circuit specification for these tools is given in the languages based on the concept of processes, variables and channels, similar to Communicating Sequential Processes (CSP) [50].

In *logic synthesis* the initial system specification is transformed into an intermediate be-

havioural format convenient for subsequent verification and synthesis. This approach is used in PipeFitter [13, 14], TAST [36], VeriSyn [16, 96, 15], where Petri nets are used for intermediate design representation. Other examples are MOODs [92] and CASH [120]. The former starts from VHDL and uses a hardware assembly language ICODE for intermediate code. The latter starts from ANSI-C and uses Pegasus dataflow graph for intermediate representation, which is further synthesised into control logic for Micropipelines [110].

Some tools do not cover the whole design, but can be combined with the other tools to support the coherent design flow. For example, Gate Transfer Level (GTL) [98, 99], Theseus Logic NCL-D and NCL-X [62] are developed for synthesis of asynchronous data path from Register Transfer Level (RTL) specifications. Other tools, such as Minimalist [44], 3D [25] and Petrify [28] are aimed at asynchronous controller synthesis from intermediate behavioural specifications. In turn, controller synthesis tools, can be combined with decomposition techniques [121, 124] to reduce the complexity of the specification.

3.3 Syntax-driven translation

The basic design flow diagram for the syntax-driven translation approach is shown in Figure 3.5. The initial system specification is compiled into a parsing tree, which is subsequently mapped into a network of handshake components. The network can be used for behavioural simulation of the asynchronous system. The mapping of the network of handshake components into a gate netlist is performed by a back-end tool, which may vary for different technologies. The obtained gate netlist is mapped into silicon by conventional place and route tools. The timing information extracted from the layout can be used together with the gate netlist for timing simulation.

The syntax-driven approach was initially used in the Tangram group at Philips Research [12, 57]. The Tangram design flow depends on a proprietary CSP-based language and private tool set. While being successfully used in the Philips research environment, the proprietary nature of the tools made practical widespread adoption of this methodology problematic.

The syntax-driven design flow became available for public use after the Manchester Amulet Group developed the Balsa design kit [4, 40]. Similar to Tangram, it relies on the paradigm of handshake components [10, 87] as an intermediate representation of an asynchronous system.

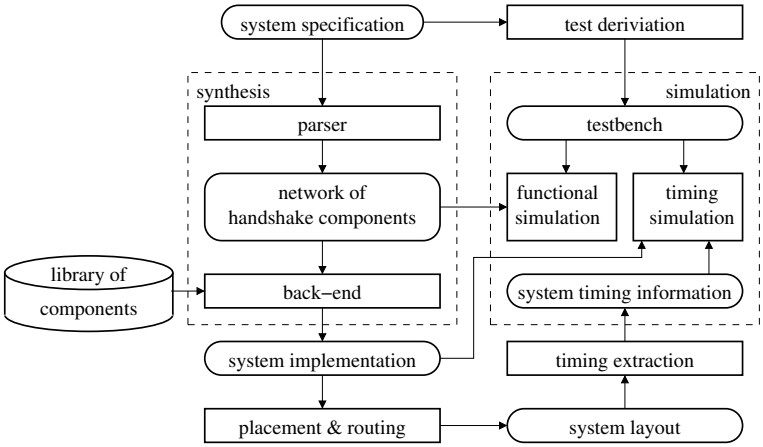


Figure 3.5: Syntax-driven design flow

The Balsa language is created to provide a source for compiling handshake components and is also very similar to Tangram. In Balsa the circuits are described by procedures which contain the specification of processes. Procedures communicate by means of handshake ports. Most procedures consist of a body command whose behaviour is perpetually repeated using a loop.

Consider the operation of the Balsa design flow on the example of the *Greatest Common Divisor* (GCD) of two integers, which is a popular benchmark in the literature about digital circuit design. The GCD of two non-zero integers is the largest integer which divides both numbers with no remainder. It can be found by iteratively subtracting the smaller number from the greatest and replacing the greatest number by the result of subtraction. The procedure stops when both numbers are equal, the GCD is equal to either of these numbers.

The description for the GCD problem in Balsa language is shown in Algorithm 1. The line numbers in the left column are shown for reference only and do not belong to the Balsa language. For simplicity reason the input values are assumed to be greater than zero. The first line of the code contains an inclusion of a pre-compiled module `[balsa.types.basic]`, which only defines some common types, for example `byte`. The second line starts the procedure declaration with 8-bit input ports `x`, `y` and an 8-bit output port `z`, which are declared in lines 03-05. The local 8-bit variables `a` and `b` are declared in line 06. The procedure body is enclosed in an infinitely repeating loop. Inside the loop the concurrent communication on input channels `x` and `y` is expected, line 09. The concurrent operations are separated by concurrency statement (`||`).

Algorithm 1 GCD algorithm in Balsa language

```
01  import [balsa.types.basic]
02  procedure GCD (
03    input x :  byte;
04    input y :  byte;
05    output z :  byte) is
06  local variable a, b :  byte
07  begin
08    loop
09      x -> a || y -> b;
10      loop
11        while a /= b then
12        begin
13          if a > b then
14            a := (a - b as byte)
15          else
16            b := (b - a as byte)
17          end
18        end
19      end;
20      z <- b
21    end
22  end
```

The values of the input channels are saved into local variables using *channel -> variable* statements. After that the `while` loop with a *x /= y* condition is started, where `/=` means 'not equal'. Note that sequential operations are separated by sequence statement (`;`). Inside the `while` loop the `if...then...else...end` statement is exploited, lines 13-17. In both its branches the assignment of an expression to a variable with type casting to `byte` is executed. In line 20, sequentially to the `while` loop, the output communication is synchronised using *channel <- variable* statement. The handshake circuit obtained by compilation of this source code is shown in Figure 3.6.

A handshake circuit consists of handshake components (circles with the operation name inside) linked by channels (solid arcs). Each handshake component has one or more ports with which it can be connected point-to-point to a port of another handshake circuit by means of a channel. Each channel carries request and acknowledgement signalling as well as an optional data payload. The requests flow from the active component ports (filled circles) towards passive component ports (open circles). Acknowledgements flow in the opposite direction to requests. Where a channel carries data, the direction of the data is indicated by an arrow on that channel's arc. The direction of data may be different from the direction of signalling to support push and pull port and channels.

A handshake component can be activated by sending request to its passive port. When ac-

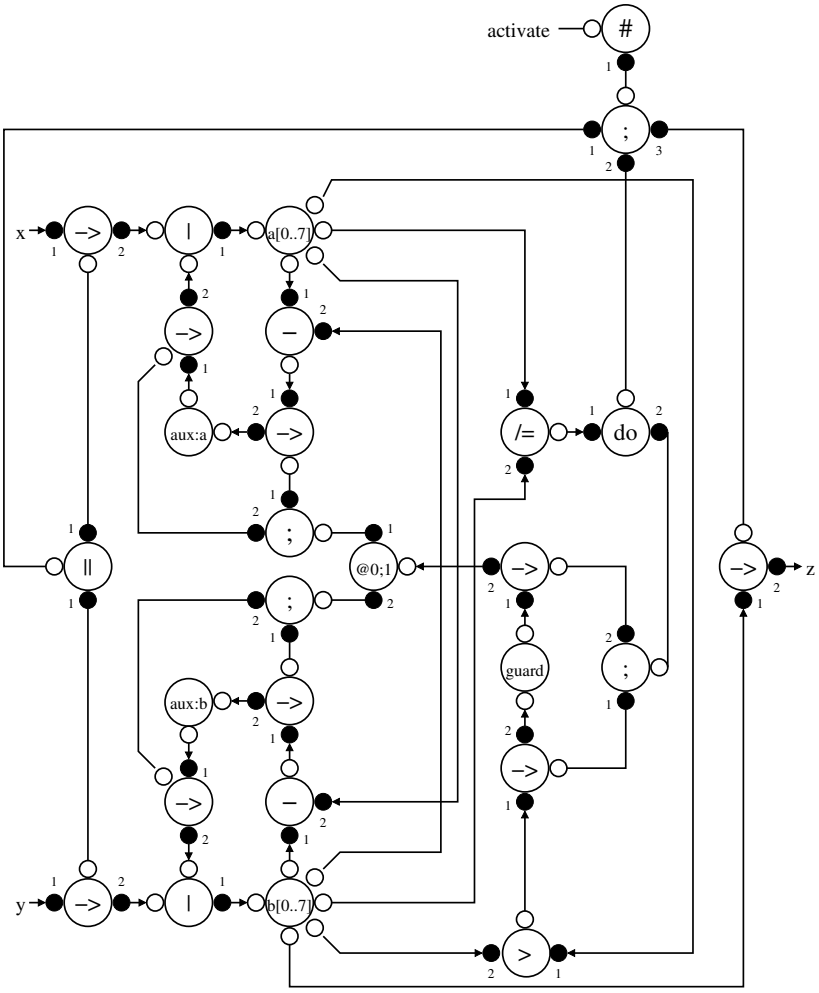


Figure 3.6: Handshake circuit for GCD

tivated, it sends requests to a subset of its active ports and waits for acknowledgements. The subset of the ports activated by the component is determined by its function and may be data-dependent. The order in which the component activates its ports is shown by small numbers next to the ports. The ports of a handshake component which are marked with the same number are activated concurrently. When all activated ports are acknowledged, the handshake component sends an acknowledgement to the passive port from which it was activated and finishes its operation until the next activation.

One can notice correspondence between the syntax of the Balsa program and the structure of the GCD handshake circuit in Figure 3.6. The operation of the GCD circuit starts with the request on the channel marked as `activate`. It activates the loop-component (`#`), which in turn sends a request to sequence-component (`;`).

First, the sequence-component activates the concur-component (`||`). The concur-component controls the fetching operation (`->`) for input channels `x` and `y`. The data from input channels `x` and `y` is pushed through the multiplexers (`|`) to the variables `a` and `b` respectively. When data is stored, the variables send acknowledgements back to the sequence-component (`;`), which then activates the while-component (`do`).

The while-component (`do`) requests the guard, which is the not-equal comparison (`/=`) between `a` and `b` variables. If the guard returns `true`, the while-component sends a request to the sequence-component, which controls the fetching of the `a>b` comparison result to the case-component (@ 0;1). If the result is true, the case-component activates the `a-b` function. The fetching of the subtraction result into a variable is performed using an intermediate `aux:a` variable and two fetch-components to avoid parallel reading and writing of `a`. Similarly, if the comparison returns `false`, the result of the `b-a` is fetched into the `b` variable.

The while-component continues to request the guard and activate the subtraction procedure (described in the previous paragraph) until the guard value becomes `false`. After that, an acknowledgement is sent back to the sequence-component, which then activates the fetching (`->`) of the `b` variable to the output channel `z`.

The syntax-driven translation is attractive from the productivity point of view, as it avoids computationally hard global optimisation of the logic. Instead some local peephole optimisation

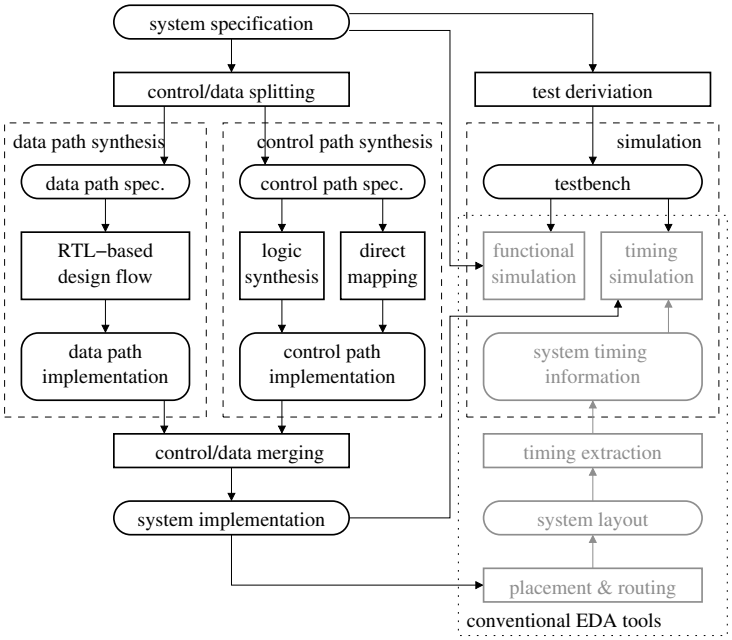


Figure 3.7: Logic synthesis design flow

is applied at the level of handshake circuits using burst mode synthesis tools, e.g. Minimalist [24]. However, the direct translation of the parsing tree into a circuit structure may produce very slow control circuits. The lack of global optimisation may not meet the requirements for high-speed circuits.

3.4 Logic synthesis

The design flow diagram for the logic synthesis approach to asynchronous system design is shown in Figure 3.7. The initial specification in a high-level HDL (System-C, Verilog or VHDL) is first split into two parts: the specification of control path and specification of the data path. Both parts are synthesised separately and subsequently merged into the system implementation netlist. The industrial EDA place and route tools can be used to map the system netlist into silicon. The existing simulation EDA tools can be reused for the behavioural simulation of the initial system specification. These tools can be also adopted for timing simulation of the system netlist back-annotated with timing information from the layout.

The variations in the design flow appear in the way of:

- extracting the specifications of control and data paths from the system specification;
- synthesis of the data path;
- synthesis of the control path either by direct mapping or by explicit logic synthesis .

The following sections consider each of these issues separately.

3.5 Splitting of control and data paths

The first step in the logic synthesis of a circuit is the extraction of control path and data path specifications from the high-level description of the system. Often the partitioning of the system is performed manually by the designers. However, this might be impracticable for a large system or under a pressure of design time constraints. At the same time, the tools automating the extraction process are still immature and require a lot of investment to be used outside a research lab.

For example, the PipeFitter tool [14], which is based on Verilog HDL and PNs as an intermediate format, supports only a very limited subset of Verilog constructs: `module, function, initial, always, wait, if, case, fork, join`. Any high-level specification which contains a loop or a conditional jump cannot be processed by this tool. A simple GCD benchmark could not even be parsed because of the `while` loop it contains. An attempt to process the specification where the loop behaviour is modelled by the `always` statement has also been unsuccessful.

A more mature VeriSyn tool has front-ends to Verilog [16], VHDL [96] and SystemC [15] languages. In addition to the constructs accepted by PipeFitter this tool supports loop statements with the Verilog front-end: `for`, `while`, `repeat`, `forever` . The following constructs are supported in the VHDL front-end: `entity`, `architecture`, `process`, `procedure`, `variable`, `wait`, `if`, `case`, `when`, `loop`, `while`, `call`, `block`. The SystemC front-end supports the following language constructs: `sc_main`, `SC_MODULE`, `SC_METHOD`, `SC_CTOR`, `sc_signal`, `sc_types`, `if`, `switch`, `for`, `while`, `repeat`, `break`, `class`.

The primary output of the VeriSyn tool is a *global net*, which is a PN whose transitions are associated with the atomic operations and whose places divide the system functioning into separate

stages. Initially the global net transitions represent complex operations corresponding to the high-level functional blocks (modules in Verilog, processes in VHDL, functions in SystemC). Then the global net transitions are refined iteratively, until all transitions represent atomic operations. This global net is used to derive a Labelled Petri net (LPN) for control path and a Coloured Petri net (CPN) for data path. The interface between control and data paths is modelled by a *local control net*, which connects the generated LPN and CPN. These PNs are subsequently passed to the synthesis tools for optimisation and implementation.

The derivation of a global net and the extraction of an LPN for control path and a CPN for the data path is illustrated on the GCD benchmark. For this the VeriSyn tool is applied to the Verilog specification of GCD algorithm, see Algorithm 2. The line numbers in the left column are shown for reference only and do not belong to the Verilog language. The first line of the code starts the `gcd` module declaration. The module has two 8-bit input ports `x`, `y` and one 8-bit output port `z`, which are declared in lines 02-04. Two internal 8-bit variables `x_reg` and `y_reg` are declared in line 05. The module consists of one infinitely repeating `always` statement which is activated by either `x` or `y` input, see line 06. Initially the input values `x` and `y` are saved into local variables `x_reg` and `y_reg` in lines 08,09. After that the `while` loop with a `x_reg != y_reg` condition is started, where `!=` means 'not equal'. Inside the `while` loop the `if...else` statement is exploited, lines 12-15. In both its branches the assignment of an expression to a variable is executed. In line 17, after the `while` loop is finished, the `z` output is generated from the value of `x_reg` variable.

As the GCD module contains one `always` statement only, the global net initially consists of one transition as shown in Figure 3.8(a). This transition is refined by using an as-soon-as-possible (ASAP) scheduling algorithm. The assignments of inputs *x* and *y* to registers *x_reg* and *y_reg* are scheduled concurrently. The parallel execution of these two statements is possible because they do not share any register, i.e. they are independent. The whole input operation, the `while` loop and the output of the result are scheduled in sequence because all of them share *x_reg* register, see Figure 3.8(b). The `while` loop is refined into *x_reg!=y_reg* condition and two branches for *true* and *false* result of the comparison, see in Figure 3.8(c). The nested `if` statement is also refined into condition *x_reg>y_reg* and two branches for *true* and *false* result of the comparison as shown in

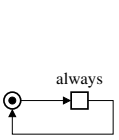
Algorithm 2 GCD algorithm in Verilog HDL

```
01  module gcd(x, y, z);
02    input [7:0] x;
03    input [7:0] y;
04    output reg [7:0] z;
05    reg [7:0] x_reg, y_reg;
06    always @(x or y)
07    begin
08      x_reg = x;
09      y_reg = y;
10      while (x_reg != y_reg)
11      begin
12        if (x_reg > y_reg)
13          x_reg = x_reg - y_reg;
14        else
15          y_reg = y_reg - x_reg;
16      end
17      z = x_reg;
18    end
19  endmodule
```

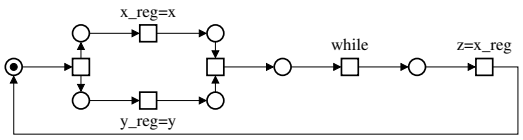
Figure 3.8(d). The conditions of both `if` and `while` statements are basic comparison operations between *x_reg* and *y_reg* registers. The sequence *x_reg!=y_reg* and *x_reg>y_reg* operators is automatically merged into a three-way comparison operation *x_reg?y_reg*, which gives one of the following results *greater_than*, *equal* or *less_than*, see in Figure 3.8(e). The refined model of the system is shown in Figure 3.8(f). At the final stage of refinement the transitions *x_reg=x_reg-y_reg* and *x_reg=x_reg-y_reg* are split into the subtraction operations (*sub_gt* and *sub_lt*) and the storage of the result (*store_x* and *store_y*). The transitions of the global net are given short and distinctive labels convenient for further reference. The global net is ready for the extraction of the control path LPN and the data path CPN.

The LPN for the control path is obtained by expanding each global net transition representing an atomic operation *x* into a transition-place-transition sequence as shown in Figure 3.9. The first transition *x_start* denotes the beginning of the operation, the place *p_x* represents the operation being executed and the last transition *x_end* denotes the end of the operation. Transition *x_start* produces the operation request *x_req* to the data path via local control net and transition *x_end* is synchronised with the operation acknowledgement *x_ack* from the data path via local control net.

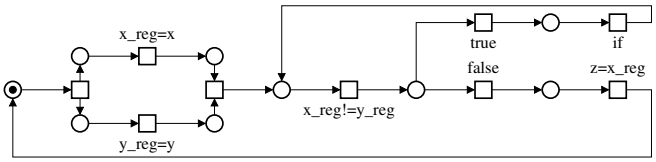
The LPN obtained by the transition expansion technique is shown in Figure 3.10(a) using solid arcs, places and transitions. The dashed arcs and places represent the local control net. Each part of the LPN highlighted with a gray box is merged into one transition, thus eliminating the redundant



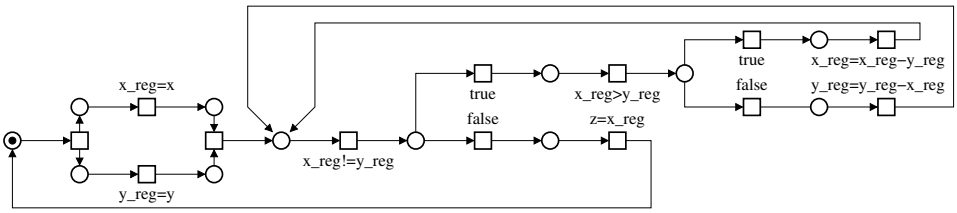
(a) Initial global net



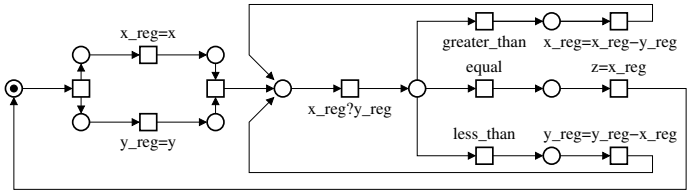
(b) Refinement of always statement



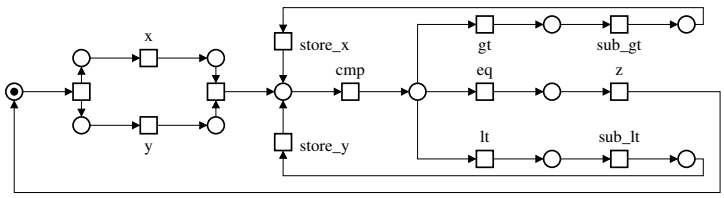
(c) Refinement of while loop



(d) Refinement of if statement



(e) Optimisation of comparison operation



(f) Resultant global net

Figure 3.8: Global net for GCD algorithm

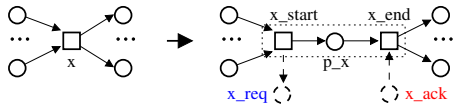


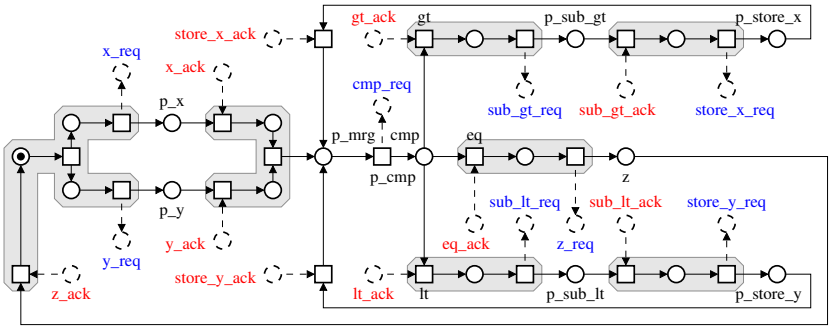
Figure 3.9: Obtaining control path LPN from global net

places which do not correspond to any operation in the data path. The result of this optimisation is shown in Figure 3.10(b). This LPN can be further optimised by leaving the acknowledgement *sub_x_ack* for subtraction operation and the request *store_x_req* for storage of the result in the data path. The subtraction action followed by the storage operation are both acknowledged by *x_ack* signal. Similarly, *sub_lt_req* can be acknowledged directly by *y_ack* leaving *sub_lt_ack* and *store_y_req* in the data path. Finally, places *p_x* and *p_y* are merged into one place *p_in*, which denotes the stage of data input. The optimised LPN is shown in Figure 3.10(c).

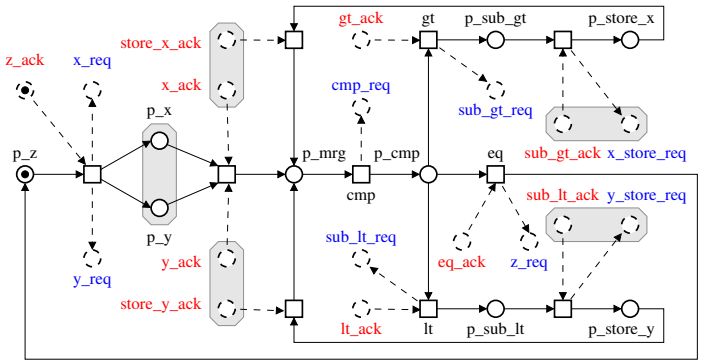
Signals *z_ack* and *z_req* compose the handshake interface to the environment. When set, the *z_req* signal means the computation is complete and output data is ready to be consumed. The *z_ack* signal is set when the output of the previous computation cycle is consumed and the new input data is ready to be processed.

The data path CPN generated by VeriSyn is presented in Figure 3.11 using the solid arcs, places and transitions. The dashed arcs and places represent the local control net. Transitions *MUX_x_0* and *MUX_x_1* are used for multiplexing the *x* input and the output of *SUB_gt* operation to *REG_x* register. Similarly, *MUX_y_0* and *MUX_y_1* are multiplexing the *y* input and the output of *SUB_lt* operation to *REG_y* register. The *CMP_xy* block of the net, framed by the dotted rectangle, is used for comparing the values of *REG_x* and *REG_y* registers. Depending on the comparison result one of the transitions $x>y$, $x=y$ or $x<y$ is fired.

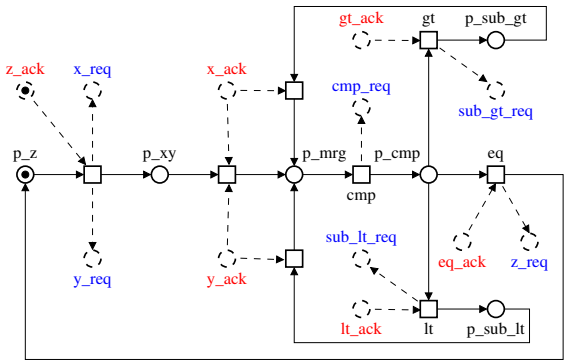
In Figures 3.10, 3.11 the dashed arcs and places belong to the local control net. All the communication between the control and data paths is carried out by means of this net, as shown in Figure 3.12. For example, when the *z_ack* signal is received, the control generates *x_req* and *y_req* signals which enable the *MUX_x_0* and *MUX_y_0* transitions in the data path. When the multiplexing is finished the values of *x* and *y* are stored using *REG_x* and *REG_y* respectively. The data path acknowledges this by *x_ack* and *y_ack* signals. The acknowledgement signals enable the *dum1* transition in the control path LPN. After that, the control path requests the comparison op-



(a) Initial LPN and local control net



(b) Optimisation of LPN



(c) Resultant LPN

Figure 3.10: LPN for GCD control path

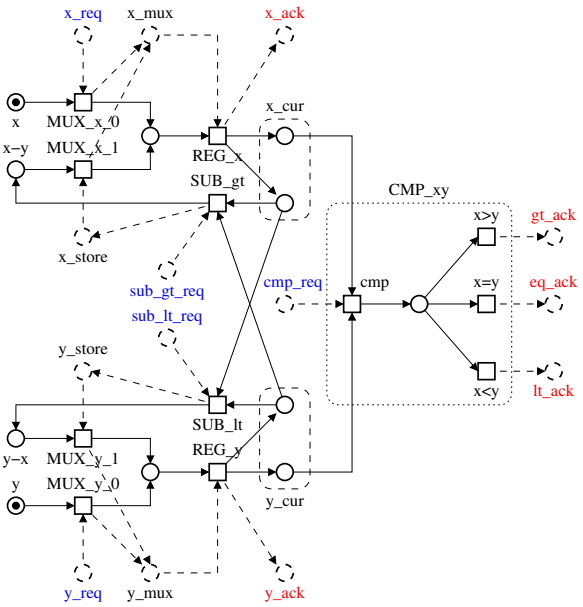


Figure 3.11: CPN for GCD data path

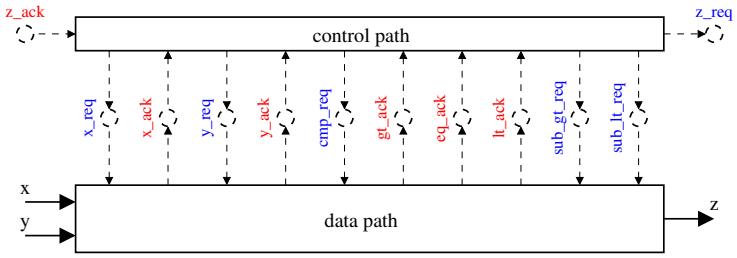


Figure 3.12: GCD interface between control and data paths

eration by means of the *cmp_req* signal. When the comparison is complete in the data path, one of the signals *gt_ack*, *eq_ack* or *lt_ack* is returned to the control. If *gt_ack* is received, the control path generates *sub_gt_req* request, which activates *SUB_xy* transition in the data path. This results in subtracting the current value of *y* from *x* and storing the difference using *REG_x* transition. The data path acknowledges this by *x_ack* and the comparison operation is activated again. If the *lt_ack* signal is issued by the data path then the operation of the system is analogous to that of *gt_ack*. However, as soon as *eq_ack* is generated, the control path issues the *z_req* signal to the environment, indicating that the calculation of GCD is complete.

Note that the local control net *x_mux* between *MUX_x_0* and *REG_x* does not leave the data path, thereby simplifying the control path. Similarly, other signals, *y_mux*, *x_store* and *y_store*, in

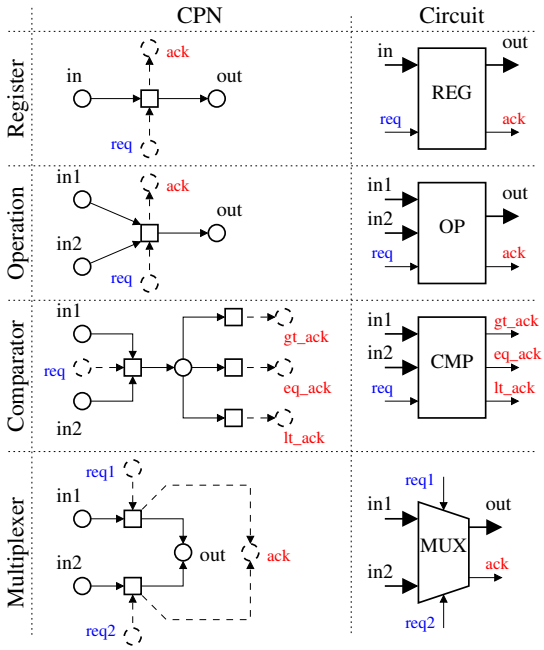


Figure 3.13: Mapping from CPN into circuit

the local control net are kept inside the data path.

The same control path LPN, data path CPN and local control net can be obtained from VHDL or SystemC specifications of GCD algorithm. These PNs are passed to synthesis tools for deriving the implementation of control and data paths.

3.6 Synthesis of data path

The method of data path synthesis employed in PN2DCs is based on the mapping of CPN fragments into predesigned hardware components. A part of the library of such components and corresponding CPN fragments are shown in Figure 3.13. The solid places and arcs in the CPN column correspond to data inputs and outputs; the dashed arcs and places denote the control signals (request and acknowledgement).

A block diagram for the GCD data path is presented in Figure 3.14. It is mapped from the CPN specification shown in Figure 3.11. The CPN is divided into the following fragments, which have hardware implementations in the library shown in Figure 3.13: 2 multiplexers, 2 registers, 1 comparator and 2 subtracters. These hardware components are connected according to the arcs

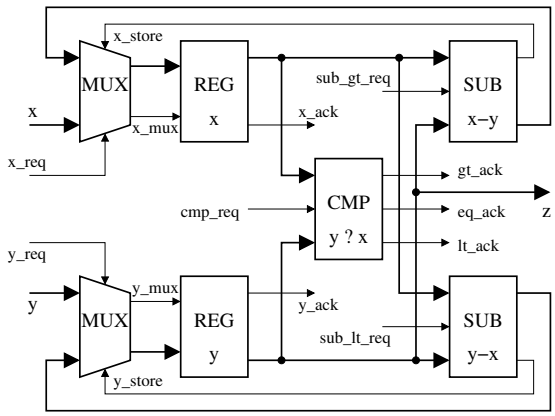


Figure 3.14: GCD data path

between the corresponding fragments of the CPN. To save the hardware, the output z is not latched in its own register. Instead it is taken from the register y and is valid when the controller sets the *z_req* signal.

If the library of data path components does not have an appropriate block, the latter should be either manually constructed or automatically generated from RTL, for example, using Theseus NCL [64] or GTL [98].

3.7 Direct mapping of control path

The main idea of the *direct mapping* is that a graph specification of a circuit can be translated directly (without computationally hard transformations) into the circuit netlist in such a way that the graph nodes correspond to the circuit elements and graph arcs correspond to the interconnects. The direct mapping approach originates from [52], where a method of *the one-relay-per-row* realisation of an asynchronous sequential circuit is proposed. This approach is further developed in [116] where the idea of the *1-hot state assignment* is described. The 1-hot state assignment is then used in the method of concurrent circuit synthesis presented in [51].

The underlying model for circuits described in [51] is an Augmented Finite State Machine (AFSM), which is an FSM with added facilities, including timing mechanisms for the delay of state changes. These circuits have inputs that are logic values (signal levels as opposed to signal transitions), which is advantageous for low-level interfacing. These circuits use a separate set-

reset flip-flop for every local state, which is set to 1 during a transition into the state, and which in turn resets to 0 the flip-flops of all its predecessor's local states. The main disadvantages of this approach are the fundamental mode assumptions and the use of local state variables as outputs. The latter are convenient for implementing event flows but require an additional level of flip-flops if each of those events controls just one switching phase of an external signal (either from 0 to 1 or from 1 to 0).

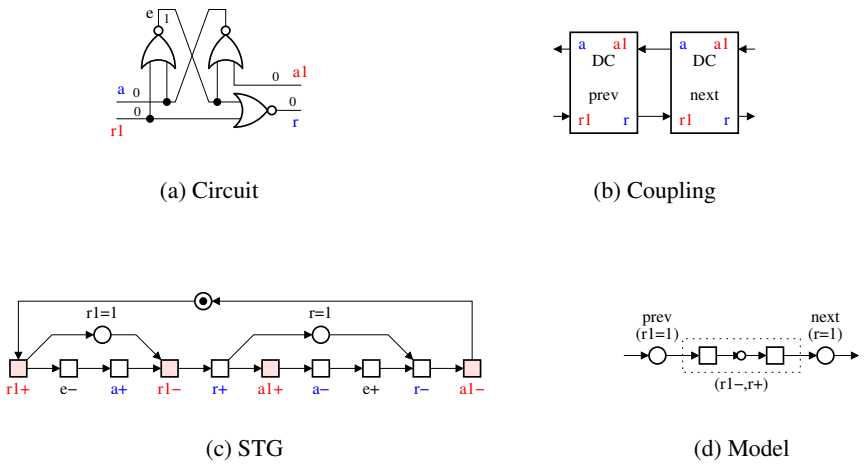
The direct mapping method proposed in [85] works for the whole class of 1-safe PNs. However, it produces control circuits whose operation uses a 2-phase (no-return-to-zero) signalling protocol. This results in lower performance than what can be achieved in 4-phase circuits.

The approach of [59] is based on *distributors* and also uses the 1-hot state assignment, though a different implementation of local states. In this method every place of a Petri net is associated with a *David cell* [34].

A circuit diagram of a *David cell* (DC) is shown in Figure 3.15(a). DCs can be coupled using a 4-phase handshake protocol, so that the interface $\langle a1,\ r\rangle$ of the previous stage DC is connected to the interface $\langle a,\ r1\rangle$ of the next stage as shown in Figure 3.15(b). Output r of a DC is used to model the marking of the associated PN place. If the output r is low the corresponding place is empty and if it is high then the corresponding place is marked with a token. The operation of a single DC is illustrated in Figure 3.15(c). The transitive places $prev$ and $next$ represent the high level of signals $r1$ and r respectively. Their state denotes the marking of the places associated to the previous stage and next stage DCs, see Figure 3.15(d). The dotted rectangle depicts the transition between $prev$ and $next$ places. This transition contains an internal place, where a token 'disappears' for the time $t_{r1-\!\!\rightarrow r+}$. In most cases this time can be considered as negligible, because it corresponds to a single two input NOR-gate delay.

The circuits built of DCs are speed-independent [77] and do not need fundamental mode assumptions. On the other hand, these circuits are autonomous (no inputs/outputs). The only way of interfacing them to the environment is to represent each interface signal as a set of abstract processes, implemented as request-acknowledgement handshakes, and to insert these handshakes into the breaks in the wires connecting DCs. This restricts the use of DCs to high-level design.

The PN2DCs tool [96] uses the *direct mapping from LPNs* approach based on [59]. In this



(a) Circuit

(b) Coupling

(c) STG

(d) Model

Figure 3.15: David cell

approach the places of the control path LPN are mapped into DCs. The request and acknowledgement functions of each DC are generated from the structure of the LPN in the vicinity of corresponding place as shown in Figure 3.16. The request function of each DC is shown in its top-left corner and the acknowledgement function in its bottom-right conner.

The GCD control path described by the LPN in Figure 3.10(c) is mapped into the netlist of DCs shown in Figure 3.17. Each DC in this netlist corresponds to the LPN place with the same name. The requests to the data path (*x_req*, *y_req*, *cmp_req*, *sub_gt_req*, *sub_lt_req*, *z_req*) and the acknowledgements from the data path (*x_ack*, *y_ack*, *gt_ack*, *lt_ack*, *eq_ack*, *z_ack*) are defined by the local control net places with the same names.

There is a space for further optimisation of the obtained control path circuit. For example, two DCs *p_mrg* and *p_cmp* could be merged. Also the request function of the *p_mrg* DC could be simplified by introducing additional memory elements. However, the obtained circuit is already implementable in standard libraries e.g. AMS. The circuit exhibits good size and speed characteristics: it consists of 120 transistors and has the worst case latency of 4 negative gates in the *x_ack*-→*cmp_req*+ trace.

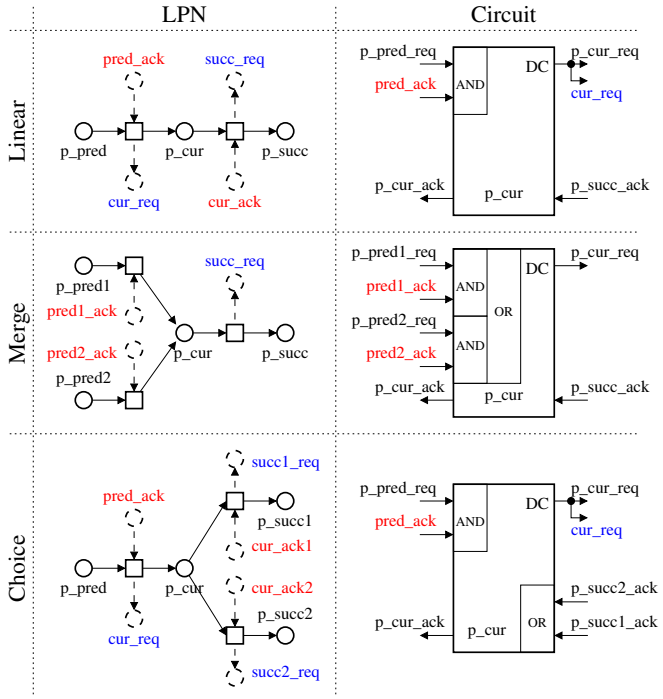


Figure 3.16: Mapping of LPN places into DCs

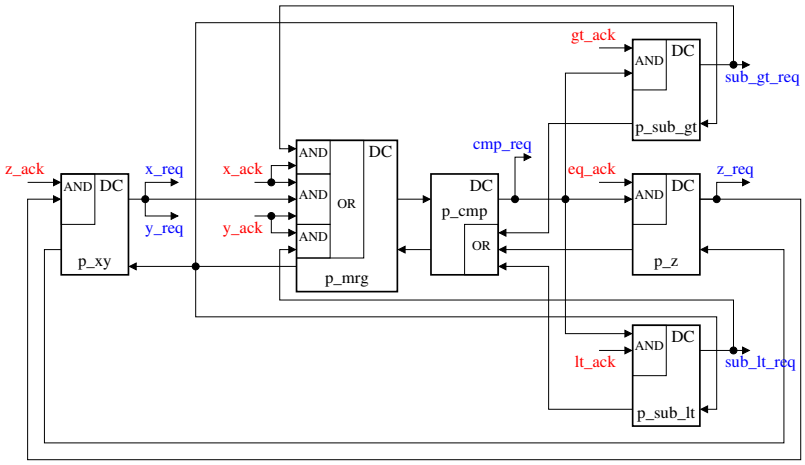


Figure 3.17: GCD control path scheme

3.8 Explicit logic synthesis

The *explicit logic synthesis* methods work with the low-level system specifications which capture the behaviour of the system at the level of signal transitions, such as STGs. These methods usually derive boolean equations for the output signals of the controller using the *next state functions* obtained from STGs [26].

An STG is a succinct representation of the behaviour of an asynchronous control circuit that describes the causality relations among the events. In order to find the next state functions all possible firing orders of the events must be explored. Such an exploration may result in a state space which is much larger than the STG specification. Finding efficient representations of the state space is a crucial aspect in building synthesis tools.

The synthesis method based on state space exploration is implemented in the Petrify tool [28]. It represents the system state space in form of a *State Graph* (SG), which is a binary encoded reachability graph of the underlying PN. Then the theory of regions [29] is used to derive the boolean equations for the output signals.

The control path STG for the GCD benchmark is shown in Figure 3.18. It is obtained from global net shown in Figure 3.8 by expanding its transitions to a 4-phase handshake protocol. After that, the GCD data path schematic shown in Figure 3.14 is taken into account to manually adjust the control path STG to the data path interface. In the modified STG the request to the comparator *cmp_req* is acknowledged in 1-hot code by one of the signals: *gt_ack*, *eq_ack* or *lt_ack*. The request to the subtracter *sub_gt_req* is acknowledged by *x_ack*. This is possible because the procedure of storing the subtraction result into the register is controlled directly in the data path and does not involve the control path. Similarly *sub_lt_req* is acknowledged by *y_ack*.

Figure 3.19 presents the SG for the GCD control path obtained from the STG in Figure 3.18. The SG consists of vertexes and directed arcs connecting them. Each vertex corresponds to a state of the system and is assigned a binary vector that represents the value of all signals in that state. The sequence of the signals in the binary vector is the following: <*x_req, y_req, x_ack, y_ack, cmp_req, gt_ack, eq_ack, lt_ack, sub_gt_req, sub_lt_req, z_req, z_ack*>. The initial state is marked with a box. The directed arcs are assigned with the signal events which are enabled in the preceding states.

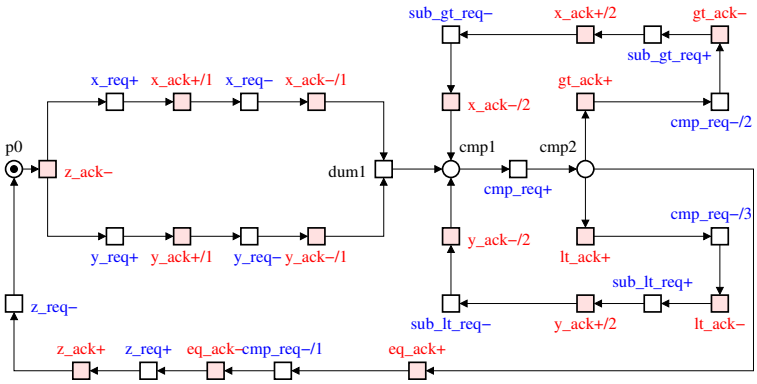


Figure 3.18: STG for GCD control path

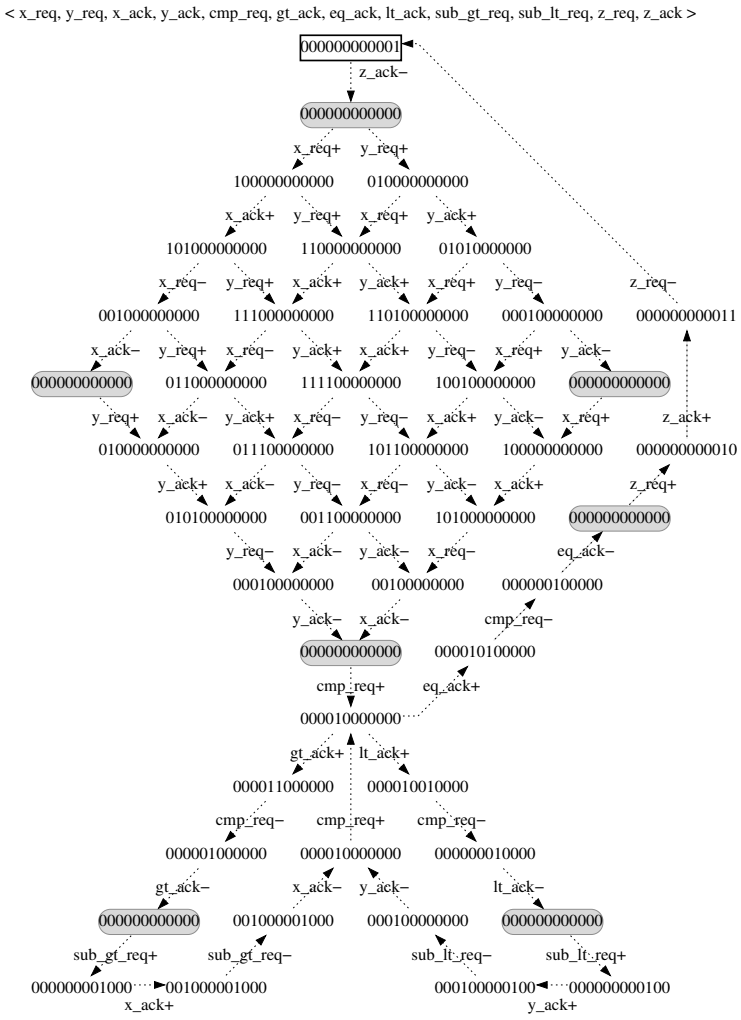


Figure 3.19: State graph for GCD control path

Note that all possible combinations of the events in two concurrent branches *x_req+*→*x_ack+*→*x_req-*→*x_ack-* and *y_req+*→*y_ack+*→*y_req-*→*y_ack-* are expressed explicitly in the SG. The explicit representation of concurrency results in a huge SG for a highly concurrent STG. This is known as the *state space explosion* problem, which puts practical bounds on the size of control circuits that can be synthesised using state-based techniques.

The other interesting issue is the unambiguous state encoding. The shadowed states in Figure 3.19 have the same binary code, but they enable different signal events. This means that the binary encoding of the SG signals alone cannot determine the future behaviour of the system. Hence, an ambiguity arises when trying to define the next-state function. Roughly speaking, this phenomenon appears when the system does not have enough memory to 'remember' in which state it is. When this occurs, the system is said to violate the *Complete State Coding* (CSC) property. Enforcing CSC is one of the most difficult problems in the synthesis of asynchronous circuits. The general idea of solving CSC conflicts is the insertion of new signals, that add more memory to the system. The signal events should be added in such a way that the values of inserted signals disambiguate the conflicting states.

3.8.1 Automatic CSC conflict resolution

One of the possibilities to resolve the CSC conflicts is to exploit the Petrify tool and the underlying theory of regions. In Petrify all calculations for finding the states in conflict and inserting the new signal events are performed at the level of SG. The tool relies on the set of optimisation heuristics when deciding how to insert new transitions. However, the calculation of regions involves the computationally intensive procedures which are repeated when every new signal is inserted. This may result in long computation time.

When the system becomes conflict-free, the SG is transformed back into STG. Often the structure of the resultant STG differs significantly from the original STG, which might be inconvenient for its further manual modification. Actually, the STG may look different even after a simple transformation into SG and back to STG, because the structural information is lost at the level of SG.

The conflict-free STG for the GCD control path is shown in Figure 3.20. There are two changes to the structure of the STG which are not due to new signal insertion. Firstly, the transition

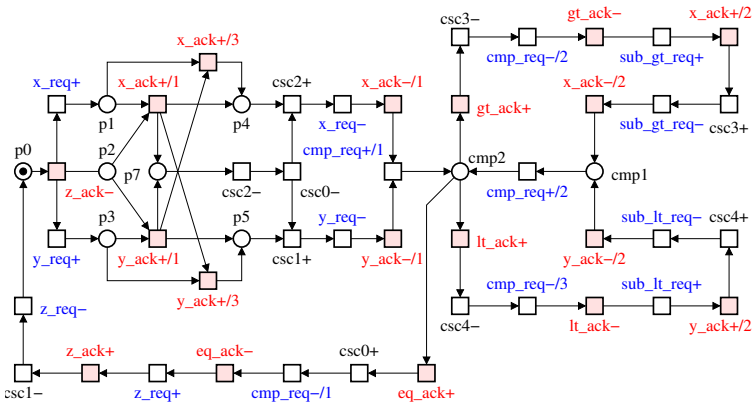


Figure 3.20: Resolution of CSC conflicts by Petrify

cmp_req+ is split into *cmp_req+/1* and *cmp_req+/2*; Secondly, the concurrent input of *x* and *y* is now synchronised on *cmp_req+/1* instead of a dummy transition.

Petrify resolves the CSC conflicts in the GCD control path specification by adding five new signals, namely *csc*0, *csc1*, *csc2*, *csc3*, *csc4*. The insertion of signals *csc0*, *csc3* and *csc4* is quite predictable. They are inserted in three conflicting branches (one in each branch) in order to distinguish between the state just before *cmp_req+/1* and just after *eq_ack-*, *gt_ack-*, *lt_ack-* respectively.

For example, the state of the system before and after the following sequence of transitions is exactly the same: *cmp_req+/1*→*eq_ack+*→*cmp_req-/1*→*eq_ack-*. In order to distinguish between these states transition *csc0+* is inserted inside the above sequence. As the behaviour of the environment must be preserved, the new transition can only be inserted before the output transition *cmp_req-/1*. The are two possibilities for its insertion: sequentially or concurrently. The former type of insertion is usually (but not always) preferable for smaller size of the circuit, the latter for lower latency. Relying on its sophisticated heuristics Petrify decides to insert *csc0+* sequentially. Signal *csc0* is reset in the same branch outside the above sequence of transitions.

Similarly, signals *csc1* and *csc2* are inserted to distinguish the states before and after the sequence of transitions *x_req+*→*x_ack+*→*x_req-*→*x_ack-* and *y_req+*→*y_ack+*→*y_req-*→*y_ack-* respectively. However the reset of *csc2* is not symmetrical to the reset of *csc1* (as expected) and involves a significant change of the original STG structure, see Figure 3.20.

The synthesis of the conflict-free specification with logic decomposition into gates with at

most four literals results in the following equations:

```
[x_req] = z_ack' (csc0 csc1' + csc2');
[y_req] = z_ack' csc1';
[z_req] = csc0 eq_ack' csc1;
[3] = csc4' + csc3' + csc0 + csc2';
[cmp_req] = [3]' x_ack' y_ack' csc1;
[sub_gt_req] = csc3' gt_ack';
[sub_lt_req] = csc4' lt_ack';
[csc0] = csc2 csc0 + eq_ack;
[csc1] = csc0' y_ack + z_ack' csc1;
[9] = csc0' (csc2 + x_ack);
[csc2] = x_ack' csc2 y_ack' + [9];
[csc3] = gt_ack' (csc3 + x_ack);
[csc4] = lt_ack' (csc4 + y_ack);
```

The estimated area is 432 units and the maximum and average delay between the inputs is 4.00 and 1.75 events respectively. The worst case latency is between the input *x_ack+/1* and the output *x_req-*. The trace of the events is *x_ack+/1*→*csc_2-*→*csc_0-*→*csc_*2+→*x_req-*. Taking into account that CMOS logic is built out of negative gates these events correspond to the following sequence of gates switching: [*x_ack*↑]→ [*x_ack'*↓]→ [*csc2'*↑]→ [*csc2*↓]→ [*csc0'*↑]→ [*9'*↓]→ [*9*↑]→ [*csc2'*↓]→ [*x_req'*↑]→ [*x_req*↓]. This gives the latency estimate equal to the delay of 9 negative gates. Note that redundant inverters are removed during the technology mapping optimisation and do not introduce excessive delay.

3.8.2 Semi-automatic CSC conflict resolution

A semi-automatic approach to CSC conflict resolution is adopted in the ConfRes tool [65]. The main advantage of the tool is its interactivity with the user during CSC conflict resolution. It visualises the cause of the conflicts and allows the designer to manipulate the model by choosing where in the specification to insert new signals.

The ConfRes tool uses STG unfolding prefixes [58] to visualise the coding conflicts. An unfolding prefix of an STG is a finite acyclic net which implicitly represents all the reachable

states of the STG together with transitions enabled at those states. Intuitively, it can be obtained by successive firings of STG transitions under the following assumptions:

- for each new firing a fresh transition (called an *event*) is generated;
- for each newly produced token a fresh place (called a *condition*) is generated.

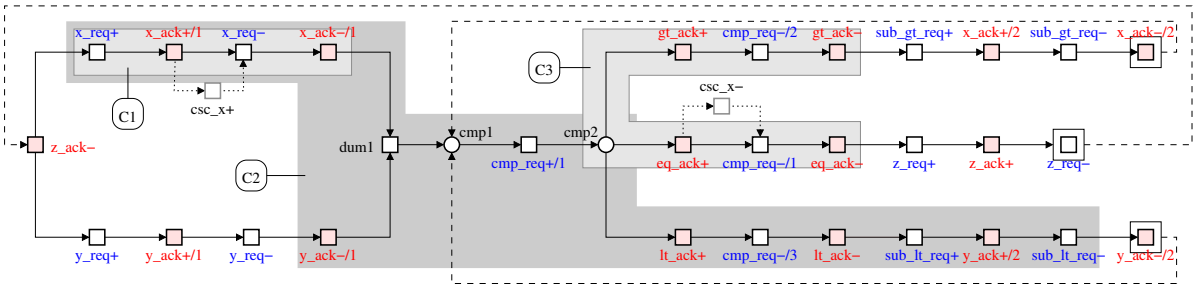
If the STG has a finite number of reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off events*) without loss of information, yielding a finite and complete prefix.

In order to avoid the explicit enumeration of coding conflicts, they are visualised as *cores*, i.e. the sets of transitions 'causing' one or more of conflicts. All such cores must eventually be eliminated by adding new signals.

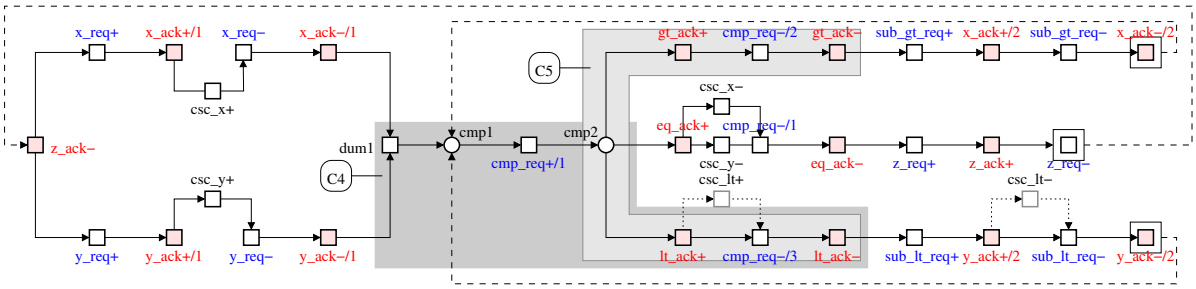
The process of core resolution in the GCD control path is illustrated in Figure 3.21. Actually, there are ten overlapping conflict cores in the STG. The ConfRes tool shows them in different colours similar to a geographical height-map. However, all ten cores would be hardly distinguishable on a gray-scale printout. That is the reason why only those cores whose resolution is currently discussed are shown. The cores are depicted as gray polygons covering the sets of sequential transitions. Each core has different brightness and is labelled with a name in a rounded box to refer from the text.

The basic rules for signal insertion are the following:

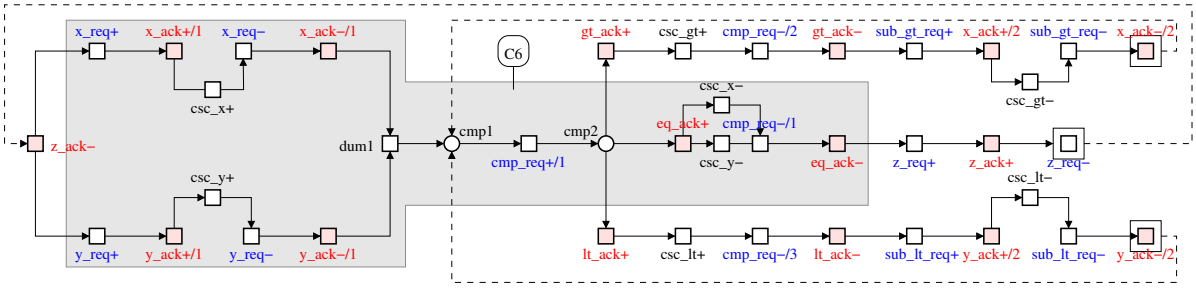
- In order to destroy a core one phase of a new signal should be inserted inside the core and the other phase outside the core.
- A new signal should be inserted into the intersection of several cores whenever possible, because this minimises the number of inserted signals, and thus the area and latency of the circuit.
- A new signal transition cannot be inserted before an input signal transition, because it would change the device-environment interface.
- Usually (but not always) the sequential insertion of a transition is preferred for smaller circuit size and concurrent insertion is advantageous for lower circuit latency.



(a)



(b)



(c)

Figure 3.21: Visualisation of conflict cores in ConfRes

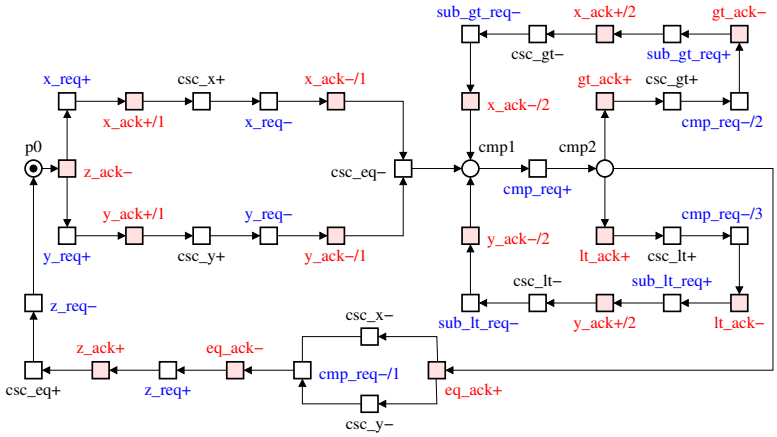


Figure 3.22: Resolution of CSC conflicts by ConfRes

Consider this procedure on the example of GCD controller, Figure 3.21. Two experiments are conducted. In the first one the strategy of sequential signal insertion is exploited in order to compete with automatic conflict resolution in circuit size. In the second experiment the new signals are inserted concurrently (where possible) in order to achieve lower latency.

In the experiment with sequential signal insertion, firstly, the cores *C1* and *C2* shown in Figure 3.21(a) are destroyed by inserting *csc_x+* transition sequentially before *x_req-*. The reset phase of *csc_x* is inserted between *eq_ack+* and *cmp_req-/1* thereby destroying the core *C3*. Similarly, two other cores, symmetrical to *C1* and *C2* (not shown in the diagram for readability), are eliminated by inserting transition *csc_y+* before *y_req-*. The reset phase of *csc_y* is inserted the same way as *csc_x-* (between *eq_ack+* and *cmp_req-/1*) and destroys the core that is symmetrical to C3.

Secondly, cores *C4* and *C5* are eliminated by inserting *csc_lt+* sequentially before *cmp_req-/3*, see Figure 3.21(b). The reset phase of *csc_lt* is inserted outside *C4* and *C5*, in sequence with *sub_lt_req-*. Likewise, the core which is symmetrical to *C4* (not shown for simplicity) is destroyed by inserting *csc_gt+* before *cmp_req+/2* and *csc_gt-* before *sub_gt_req-*.

Finally, only one core *C6* is left, see Figure 3.21(c). It is destroyed by replacing transition *dum1* by *csc_eq-*. The set phase of *csc_eq* is inserted outside the core before *z_req-*. The resultant conflict-free STG of the GCD controller is shown in Figure 3.22.

Petrify synthesises this STG with logic decomposition into gates with at most four literals into

the following equations:

```
[x_req] = csc_x' z_ack' csc_eq;
[y_req] = csc_y' z_ack' csc_eq;
[z_req] = csc_x' eq_ack' csc_eq';
[3] = csc_y' csc_x' + csc_gt + csc_lt;
[cmp_req] = [3]' y_ack' x_ack' csc_eq';
[sub_gt_req] = csc_gt gt_ack';
[sub_lt_req] = csc_lt lt_ack';
[csc_x] = eq_ack' (x_ack + csc_x);
[csc_y] = csc_y eq_ack' + y_ack;
[csc_gt] = x_ack' csc_gt + gt_ack;
[10] = csc_eq (csc_x' + csc_y') + z_ack;
[csc_eq] = csc_eq (x_ack + y_ack) + [10];
[csc_lt] = y_ack' (lt_ack + csc_lt);
```

The estimated area is 432 units, which is the same as when the coding conflicts are resolved automatically. However, the maximum and average delays between the inputs are significantly improved: 2.00 and 1.59 events respectively. The worst case latency of the circuit is between *gt_ack+* and *cmp_req-*/2 (or between *eq_ack+* and *cmp_req-*/1). If the circuit is implemented using CMOS negative gates then this latency corresponds to the following sequence of gates switching: [*gt_ack*↑]→ [*csc_gt*'↓]→ [*csc_gt*↑]→ [*3*'↓]→ [*cmp_req*'↑]→ [*cmp_req*↓]. This gives the latency estimate equal to the delay of 5 negative gates, which is significantly better than in the experiment with automatic coding conflict resolution.

The other experiment with semi-automatic CSC conflict resolution aims at lower latency of the GCD control circuit. Now the new signal transitions are inserted as concurrently as possible. Namely, *csc_x+* is concurrent to *x_ack+/1*; *csc_y+* is concurrent to *y_ack+/1*; *csc_gt-* is concurrent to *x_ack+/2*; and *csc_lt-* is concurrent to *y_ack+/2*. The other transitions are inserted the same way as in the previous experiment. Synthesis of the constructed conflict-free STG produces the following equations:

```
[0] = csc_x' z_ack' csc_eq;
[x_req] = x_req map0' + [0];
```

```
[2] = csc_y' z_ack' csc_eq;
[y_req] = y_ack' y_req + [2];
[z_req] = csc_y' eq_ack' csc_eq';
[5] = csc_y' csc_x' + map0 + csc_eq;
[cmp_req] = sub_lt_req' [5]' (map1 + eq_ack);
[sub_gt_req] = gt_ack' (sub_gt_req map1 + csc_gt);
[sub_lt_req] = sub_lt_req map1 + csc_lt lt_ack';
[csc_x] = eq_ack' (csc_x + x_req);
[csc_y] = eq_ack' (csc_y + y_req);
[csc_lt] = sub_lt_req' csc_lt + lt_ack;
[csc_gt] = sub_gt_req' (gt_ack + csc_gt);
[csc_eq] = map1' (csc_eq + z_ack);
map0 = sub_gt_req + csc_gt + csc_lt + x_ack;
[15] = csc_x' + x_req + csc_y';
map1 = [15]' y_ack' y_req' x_ack';
```

Two new signals, *map0* and *map1*, are added by Petrify in order to decompose the logic into library gates with at most four literals. This results in larger estimated circuit size, 592 units. The average input-to-input delay of the circuit becomes 1.34 events, which is smaller than in the previous experiment. However, the maximum latency of the circuit is 7 negative gate delay. It occurs, for example, between *gt_ack+* and *cmp_req-* transitions. The gates switched between these transitions together with the direction of switchings are: [*gt_ack*↑]→ [*csc_gt'*↓]→ [*csc_gt*↑]→ [*map0'*↓]→ [*map0*↑]→ [*5'*↓]→ [*cmp_req'*↑]→ [*cmp_req*↓]. The worst case latency in this implementation is greater than the latency in the previous design due to the internal *map0* and *map1* signals, which are used for the decomposition of non-implementable functions. Note that the input-output latency of the Petrify solutions is estimated after the technology mapping optimisation, thus the delay of the redundant inverters is not included in the worst case latency.

The complex gate implementation of the GCD controller, where the CSC conflict is resolved manually by inserting new signals in series with the existing ones is shown in Figure 3.23. This is the best solution (in terms of size and latency) synthesised by Petrify with the help of the ConfRes tool. It consists of 120 transistors and exhibits the latency of 5 negative gates delay.

Clearly, semi-automatic conflict resolution gives the designer a lot of flexibility in choosing

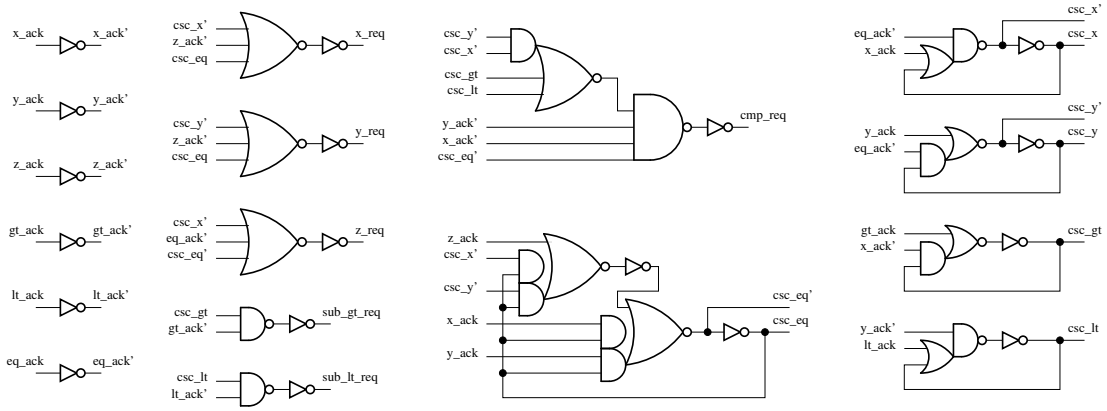


Figure 3.23: Complex gates implementation of GCD controller

between the circuit size and latency. The visual representation of conflict cores distribution helps the designer to plan how to insert each phase of a new signal optimally, thus possibly destroying several cores by one signal. The diagram of core distribution is updated after every new signal insertion. As all the modifications to the system are performed on its unfolding prefix, there is no need to recalculate the state space of the system, which makes the operation of ConfRes extremely fast.

Another approach to CSC conflict resolution, which avoids the expensive computation of the system state space, is proposed in [21, 20]. The approach is based on structural methods, which makes it applicable for large STG specifications. Its main idea is to insert a new set of signals in the initial specification in a way that unique encoding is guaranteed in the transformed specification. The main drawback of this approach is that the structural methods are approximate and can only be exact for well-formed PNs.

3.9 Tools comparison

In this section the tools are compared using GCD benchmark in two categories: system synthesis from high-level HDLs and synthesis of the control path from PNs.

Table 3.1 presents the characteristics of asynchronous GCD circuits synthesised by Balsa and PN2DCs tools from high-level HDLs. Both solutions are implemented using the AMS-0.35μm technology and dual-rail data path components. The size of each circuit is calculated using Ca-

Tool	Area (μm^2)	Speed (ns)		computation time (s)
		x=y	x=12, y=16	
Balsa	119,647	21	188	< 1
PN2DCs	100,489	14	109	< 1
Improvement	16%	33%	42%	0

Table 3.1: Comparison between Balsa and PN2DCs

Tool		number of transistors	latency (units)	computation time (s)
PN2DCs		120	6.0	<1
Petrify	automatic	116	13.0	18
	sequential	120	8.0	2
	concurrent	142	11.0	4

Table 3.2: Comparison between PN2DCs and Petrify

dence Ambit tool and the speed is obtained by circuit simulation in SPICE analog simulator.

The benchmark shows that the circuit generated by PN2DCs is 16% smaller and 33-42% faster than the circuit synthesised by Balsa. The size and speed improvements in PN2DCs compared to Balsa solution are due to different control strategies. Note that the intermediate controller specification for the PN2DCs tool is manually optimised by removing redundant places and transitions. This reduces the control path area by four DCs ($732\mu m^2$). However, the optimisation algorithm is straightforward, the redundant places and transitions removal can be further automated.

The time spent by Balsa and PN2DCs to generate the circuit netlists is negligible. This is because both tools use computationally simple mapping techniques, which allow processing of large system specifications in acceptable time.

The characteristics of the circuits synthesised from the control path specification are shown in Table 3.2. The number of transistors for the circuits generated by Petrify is counted for complex gate implementation. The technology mapping into the library gates with at most four literals is applied.

In all experiments, the input-output latency is counted after technology mapping optimisation. The latency is estimated as the cumulative delay of negative gates switched between an input and the next output. The following dependency of a negative gate delay on its complexity is used. The latency of an inverter is associated with a unit delay. Gates which have maximum two transistors

in their transistor stacks are associated with 1.5 units; 3 transistors - 2.0 units; 4 transistors - 2.5 units. This approximate dependency is derived from the analysis of the gates in the AMS $0.35\mu m$ library. The method of latency estimation does not claim to be very accurate. However, it takes into account not only the number of gates switched between an input and the next output, but also the complexity of these gates.

The Petrify tool was used to synthesise the circuits with three alternatives for CSC conflict resolution. In the first circuit the coding conflict is solved by inserting new signals automatically. In the second and the third circuits the semi-automatic method of conflict resolution is employed by using the ConfRes tool. In the second circuit the transitions of new signals are inserted sequentially, and in the third one concurrently.

The experiments show that the automatic coding conflict resolution may result in a circuit with high output latency which is due to non-optimal insertion of the new signals. The smallest circuit is synthesised when the coding conflicts are resolved manually by inserting the new signals sequentially. This solution also exhibits lower latency than in the case of automatic and concurrent signal insertion. The circuit with the new signals inserted concurrently lacks the expected low latency because of its excessive logic complexity.

3.10 BESST design flow

Between syntax driven-translation and logic synthesis of asynchronous circuits the latter produces smaller circuits with faster control path. The existing logic synthesis design flow, however, has several drawbacks:

- The design flow lacks an automatic synthesis of hazard-free data path components.
- The data path synthesis is unacceptable for security applications due to dependency between processed data and power consumption.
- Synthesis of the control path described at STG level exhibits high algorithmic complexity.
- The control path obtained from STG exhibits high input-output latency.
- Unpredictable, often high latency of the control circuits synthesised by existing techniques.

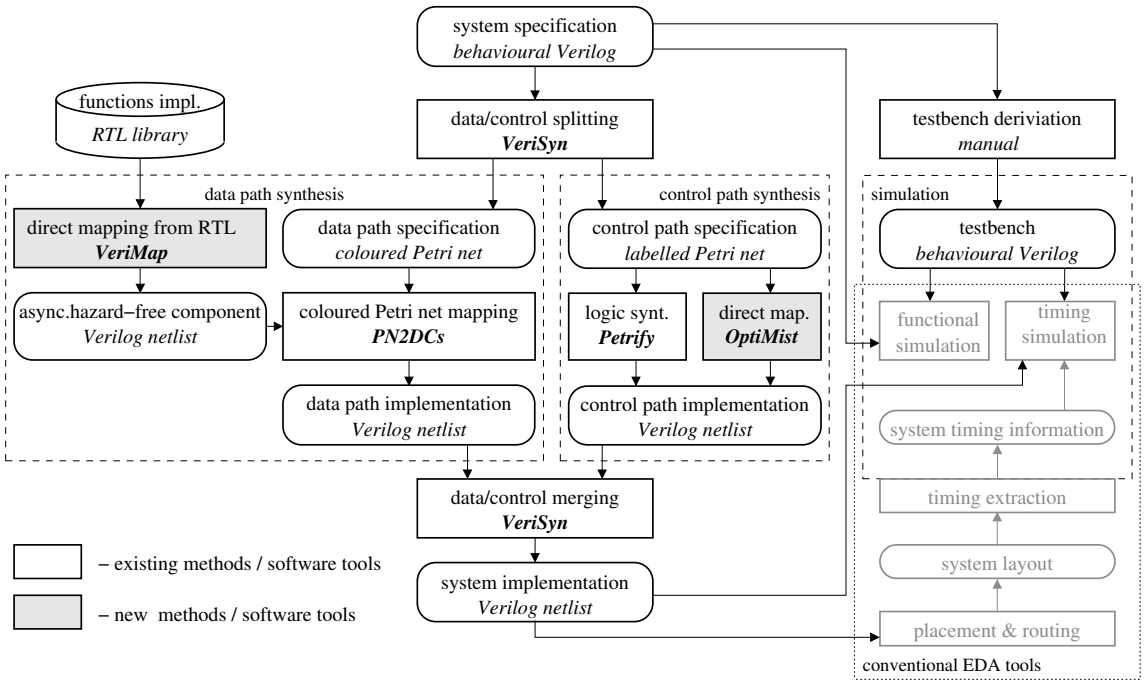


Figure 3.24: BESST design flow

These problems are addressed in the BEhavioural Synthesis of Self-Timed Systems (BESST) design flow [101] which is based on logic synthesis design flow reviewed in Section 3.4. The BESST design flow uses PNs as an intermediate representation of circuit specification. It incorporates software tools for high-level partitioning, scheduling, direct mapping and logic synthesis. These are used to generate efficient speed-independent circuits with optional security features. A diagram of the BESST design flow is shown in Figure 3.24, where the shaded boxes denote the new methods and software tools developed in the scope of this thesis.

The conventional behavioural Verilog is used as an initial specification of the system. The system is first partitioned into subsystems and then divided into control and data paths by the VeriSyn software tool [16], which operates as a front-end. VeriSyn performs scheduling to produce an LPN for the control path and a CPN for the data path, see Section 3.5. These nets are passed to the tools, which perform synthesis of control and data paths respectively.

The data path is mapped directly from the CPN into a netlist of hardware components using the PN2DCs tool as describe in Section 3.6. In the existing design flow the library of the data path components is either manually designed or obtained by conventional RTL tools. The

manual development of the components is resource consuming and prone to human errors. Being optimised for the synchronous circuit architecture the RTL-based design flow does not guarantee hazard-free implementation of the components. The single-rail nature of the RTL-based components also causes problems with completion detection which is necessary to support the request-acknowledgement handshake protocol essential for the asynchronous circuit architecture. A common solution for the completion signal is the delay element insertion between the request input and the acknowledgement output. The properties of the delay element are chosen by careful timing analysis of the component because the delay of the acknowledgement signal must be greater than the worst case computation time of the component. This results in the worst case performance of each data path component. The worst case performance can be partially avoided by inserting an adjustable delay element in the request-acknowledgement line, its delay is chosen depending on the input values of the component.

A method and a software tool (VeriMap) for automatic synthesis of hazard-free data path components are presented in Chapter 5. Verimap tool is integrated into conventional RTL design flow. Its input is a single-rail circuit synthesised from behavioural specification by an RTL tool. The method employs dual-rail encoding and monotonic switching, which facilitate the hazard-free logic. The completion detection built on dual-rail logic provides the average case performance of data path components. The method extends the traditional single-spacer dual-rail encoding with two spacers (all-zeros and all-ones) alternating in time. The alternating spacers provide strictly periodic refreshing of all wires in the data path which is beneficial for testing, dynamic logic and security applications. The VeriMap tool is compatible with the conventional EDA design flow at the level of Verilog netlists.

The control path can be obtained by either logic synthesis or direct mapping. In the logic synthesis approach boolean equations for the control path are derived using next-state functions, which are obtained from its STG. A state-based synthesis tool Petrify derives equations by exploring all possible orders of STG events, i.e. building its state space. The state space of a system grows exponentially with the increase of concurrency in the specification. This might result in unacceptable computation time or memory usage, which is the main drawback of the explicit logic synthesis method. The other problem of the logic synthesis approach is the resolution of CSC

conflicts. A CSC conflict arises when semantically different states of an STG have the same binary encoding. The automatic resolution of CSC conflicts might be computationally hard or even impossible for STGs with high concurrency, see Section 3.8.

An approach for STG synthesis which avoids high computational complexity and CSC conflicts is direct mapping. However, the existing logic synthesis design flow is only capable of direct mapping from high-level LPNs, see Section 3.7. In order to fill this gap a method and software tool (OptiMist) for the direct mapping of control path from STGs are developed, see Chapter 4. This method avoids building the whole state space of the system, all optimisations are performed locally by means of heuristics. The latency reduction is obtained by using a tracker-and-bouncer architecture. The tracker computes the state of the system concurrently to the environment operation. The bouncer produces the outputs based on the tracker's state as soon as the inputs are received from the environment. However, the circuits produced by OptiMist are usually larger than Petrify's solutions.

A combination of logic synthesis and direct mapping techniques in a single design flow might prove most advantageous. For example, at first, each output signal which has a complete state coding is synthesised individually by Petrify. Then, all the remaining outputs, whose CSC resolution is hard or impossible, are mapped into logic at once using the OptiMist direct mapping tool. Thus the best trade-off between area and performance may be achieved.

The data path and control path implementations are finally merged into the system netlist in the structural Verilog format. Conventional EDA tools are applied to the netlist for place-and-route and simulation of the system.

3.11 Summary

The state of the art in the synthesis of asynchronous systems form high-level behavioural specifications has been reviewed. Two main approaches of circuit synthesis have been considered: syntax-driven translation and logic synthesis.

Firstly, the syntax-driven approach is studied in the example of Balsa design flow. It uses a CSP-based language for the initial system specification. Its parsing tree is translated into a handshake circuit, which is subsequently mapped to the library of hardware components. This

approach enables the construction of large-size asynchronous systems in a short time, due to its low computational complexity. However, the speed and area of the circuit implementations may not be the best possible. Therefore this approach benefits from peep-hole optimisations, which apply logic synthesis locally, to groups of components, as was demonstrated in [24].

Secondly, the logic synthesis approach is reviewed using the VeriSyn, PN2DCs, Petrify and ConfRes tools. The VeriSyn tool partitions the VHDL system specification on control and data paths. Petri nets are used for their intermediate behavioural representation. The data path PN is subsequently mapped into a netlist of data path components using the PN2DCs tool. The controller PN can be either mapped into a David cell structure by PN2DCs tool or further refined to an STG. The STG control path specification can be synthesised by one of the above mentioned tools. Logic synthesis approach is computationally harder than the syntax-driven translation. The direct mapping of Petri nets in PN2DCs helps to avoid state space explosion involved in the state encoding procedures used in Petrify. At the same time, this comes at the cost of more area.

It should be clear that tools like Petrify and ConfRes can only be used for relatively small control logic, for instance in interfaces and pipeline stage controllers (see [31]), rather than complex data processing controllers, where PN2DCs is more appropriate. The latter is however not optimal for speed because it works at a relatively high-level of abstraction.

The GCD benchmark is then used to evaluate all of the above mentioned tools. The size and speed of the resultant circuits are compared. They demonstrate the various possible enhancements in the design flow, such as the use of an interactive approach to state encoding in logic synthesis.

Finally, the drawbacks of the existing design flows are pointed out and an improved version of logic synthesis design flow is proposed. The control path synthesis is enriched by a computationally simple method and software tool for direct mapping of low-latency control path from STGs. These are described in Chapter 4. The other improvement concerns the automatic synthesis of hazard-free data path components. The method and a software tool implementing this approach are presented in Chapter 5.

Chapter 4

Synthesis of control path

Two main approaches to design of asynchronous controllers are logic synthesis [28] and direct mapping [51, 59]. The review of the state of the art in Chapter 3 exposed the weak spots of both approaches. The logic synthesis approach is well developed and supported by tools, however it suffers from excessive computation complexity and cannot be applied to large specifications. The direct mapping approach is computationally simple, however it is insufficiently studied and existing techniques for direct mapping often produce large circuits with an inefficient interface to the environment.

The controllers and interfaces are traditionally specified by timing diagrams and STGs. However, the majority of direct mapping techniques work with high-level Petri nets and cannot process low-level specifications. An attempt to apply direct mapping method at a low-level, where the circuit behaviour is captured at the level of signal events, is made in [119]. In this approach DC structures are used to capture the state of the system and to control flip-flops which are associated to each output signal. Inputs, however, are still represented as abstract processes and free-choice nets are not supported.

Direct mapping from STGs and the problem of device-environment interface are addressed in [18]. In proposed method a system specification is, firstly, split into a device STG and an environment STG. These are synchronised via a communication net, which model wires. The device STG is considered separately. It consists of a *tracker* and a *bouncer*. The *tracker* follows the state of the environment and is used as a reference point by the device outputs. The *bouncer*

interfaces the environment and generates output events in response to the input events according to the state of the tracker. This two-level device architecture provides an efficient interface to the environment and is convenient for subsequent mapping into a circuit netlist.

The work presented in this chapter is based on the idea of [18]. The method is extended by a set of optimisation algorithms and heuristics. These are implemented in a software tool called OptiMist. The speed-independent circuits obtained by this method have a two-level architecture, which contributes to a low-latency interface to the environment. The OptiMist tool exhibits the computation time growth linear to the specification size which allows to apply the method to large STGs. A way of combining logic synthesis and direct mapping to employ the advantages of both approaches is also discussed.

This chapter is based on the results developed in [102, 103, 104].

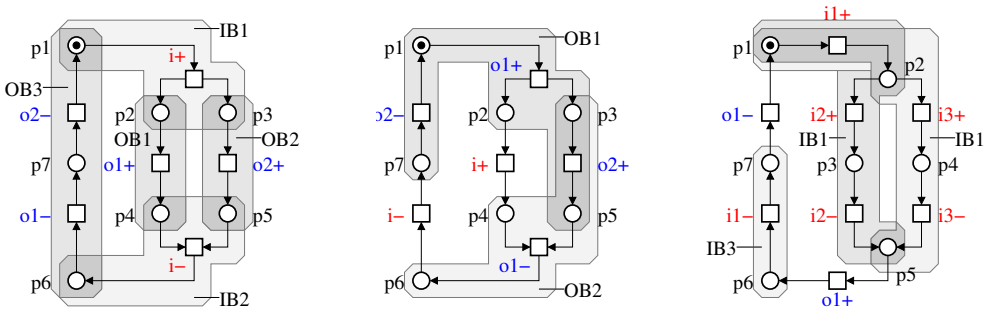
4.1 Method

A distinctive characteristic of the proposed direct mapping technique is that the system STG is converted into a form convenient for mapping into circuit netlist. It is achieved by associating groups of places and transitions to the state holding elements and by modelling connections between circuit components with arcs.

The initial specification describes the behaviour of both device and environment as a complete system. Usually, only the device needs to be synthesised which requires the extraction of the device model from the system STG. In order to do this the system STG is split into a device model and an environment model, which are connected by an intermediate net. Only the device model is subsequently optimised and mapped into a circuit netlist.

4.1.1 Requirements to the initial specification

There are several limitations on the class of STGs which can be synthesised using our direct mapping method. Similarly to the requirements of logic synthesis methods the STG must be consistent and persistent. The STG consistency is essential for any hardware implementation due to the nature of binary signals whose rising and falling transitions alternate. Persistency is required to avoid arbitration in the device by letting the environment make all the choices.



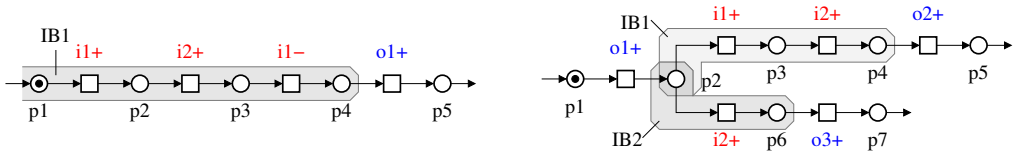
(a) Simple bursts (b) Concurrency (c) Choice

Figure 4.1: Input and output bursts

Unlike logic synthesis methods in our approach neither USC nor CSC is necessary for the whole STG. The limitation on the state encoding is more relaxed and is defined using the notion of bursts: a maximally connected subgraph of an STG which only includes transitions of an input (output) sequence and places incident to them is called *input (output) burst*. Two bursts are said to be in *conflict* if there is a transition in one burst which is in a direct conflict with a transition from another burst. A burst *B1* is said to be *covered* by burst *B2* if they are in conflict and all signal events of *B1* also exist in *B2* possibly in different order. Note that in a persistent STG only input bursts can be in conflict and covered.

The notion of bursts is illustrated in Figure 4.1. The STG in Figure 4.1(a) contains two input and two output bursts (*IB1*, *IB2* and *OB1*, *OB2* respectively). Note that even though an output sequence $o1+$, $o2+$ is possible from a reachable marking $\{p2, p3\}$, the output bursts *OB1* and *OB2* are separate because their graphs are not connected. The example in Figure 4.1(b) shows two output bursts *OB1* and *OB2* (input bursts are trivial and are hidden for simplicity). These output bursts are overlapping, however they cannot be merged into one burst because there is no output sequence which contains both $o1+$ and $o1-$. In Figure 4.1(c) three input bursts are shown (trivial output bursts are hidden). Note that conflicting input bursts *IB1* and *IB2* are separated in this STG even though their graphs are overlapping because these bursts belong to different input sequences.

All the requirements to encoding of the system states in the proposed direct mapping method are due to the delay-insensitive nature of the device-environment interface. Usually a designer can



(a) Non-unique burst encodding

(b) Conflicting bursts

Figure 4.2: Bursts in a delay-insensitive interface

control the wire delays inside a relatively small device and build it speed-independent. However, the delays of connections between the device and the environment often cannot be guaranteed. The uncontrollable interconnect delays on the device-environment interface may result in a situation when two signals issued in sequence by a sender reach a receiver simultaneously or even in a reversed order. This means that while the device itself can be speed-independent, its interface to the environment should be built under delay-insensitive timing assumptions.

One of the ways to ensure the delay-insensitive interface is to apply *order relaxation* [93] to the initial STG. This approach, however, may complicate the structure of the STG, which is disadvantageous for a direct mapping technique. In our direct mapping approach the order relaxation is not applied explicitly. Instead, a device distinguishes the end of an input burst by catching an encoding in which all inputs comprising the burst have switched. The unique identification of such encoding is only possible if all states corresponding to the input burst are coded uniquely. The opposite is also true for output bursts: in order to uniquely identify the end of an output burst the encodings of its states should be unique. For example, the STG in Figure 4.2(a) has an input burst *IB1* which can cause problems. If due to interconnect delays *i2+* reaches the device before *i1+* then the device can produce *o1+* by mistake even without waiting for *i1+* and *i1-*.

Thus, in our method each input and output burst of the system STG must have USC. An STG is said to satisfy *burst USC* property if there is no USC conflict in any input or output burst, i.e. the state encoding is unique within each individual burst. In order to satisfy the burst USC property it is sufficient for a consistent and persistent STG to have no more than one transition of each signal in every input and output burst. The uniqueness of a signal transition in each burst implies monotonic change of the code and hence no repetition of the state encoding within the burst.

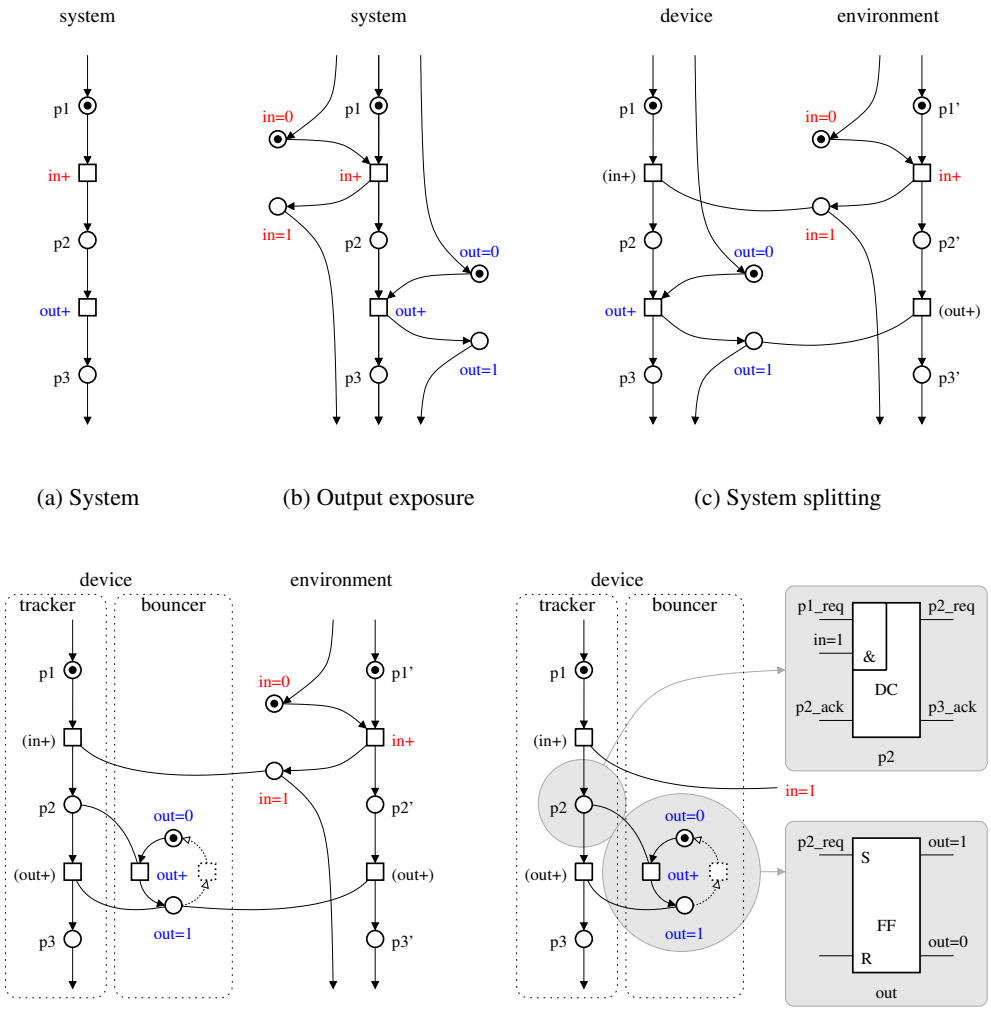
Another type of ambiguity introduced by the delays in the device-environment interconnect may occur when choosing between conflicting branches. For example, the STG in Figure 4.2(b) has two input bursts *IB1* and *IB2* which are in conflict. The transitions in the direct conflict are different (*in1*+ and *in2*+) and the choice is unambiguous. However, if the transitions of the burst *IB1* reach the device in the reverse order (*in2*+ first), then the device will be confused which conflicting branch the environment selected. The device still can recognise the branch selection if the following condition holds: the state encodings after each input burst is different from all encodings in the bursts it conflicts with. In order to satisfy this condition it is sufficient for a consistent and persistent STG with burst USC to contain only non-covered bursts; such an STG is called *non-covered*. Indeed, the state of all STG signals is the same before conflicting bursts and the change of encoding is monotonic within each burst. If none of the bursts is covered by the others then the encoding after a burst is not repeated in any burst it conflicts with.

To summarise, in order to be mappable into a circuit using our direct mapping method a system STG must be consistent, persistent, non-covered and must have burst USC. Checking these properties is computationally hard problem which does not feet into a direct mapping design flow aiming at low algorithmic complexity. Instead it is assumed that the control path STG is supplied by a high-level synthesis tool which insures the above properties by construction. All the transformations presented in the following sections preserve the behavioural equivalence if the original STG satisfying these properties.

4.1.2 Transformation

The idea of the our direct mapping method is illustrated on a basic example whose STG is partially shown in Figures 4.3(a). The depicted slice of the specification contains the *in*+ input event causing the *out*+ output event.

The first step in extracting the device model is the exposure of the signal states as shown in Figure 4.3(b). For this each signal z is associated with a pair of complementary places $z = 0$ and $z = 1$ representing low and high levels of the signal. These places are inserted as transitive places between positive and negative transitions of z, thus expressing the property of signal consistency. Note that the transitive places do not change the behaviour of the system and weak bisimulation is



(a) System (b) Output exposure (c) System splitting

(d) Tracker-bouncer architecture (e) Mapping into circuit

Figure 4.3: Method for the direct mapping from STGs

preserved on this stage of transformation.

The second step of the transformation is splitting the system specification into *device* and *environment* parts as shown in Figure 4.3(c). For this the STG obtained in the first step is duplicated. In the first copy, corresponding to the device, the transitive places associated to the inputs are removed. Similarly, in the second copy, corresponding to the environment, the transitive places associated to the outputs are removed. The behaviour of the device and the environment parts is synchronised by means of read-arcs as follows. In the environment part, each transitive place associated to low (high) level of an input signal z_I is connected by read-arcs to all negative (positive) transitions of z_I in the device. After that the transitions of input signal z_I in the device part are replaced by dummies. This way the device follows (or tracks) the behaviour of environment. Similar procedure applies to all output signals but with the roles of device and environment changed. In the device part, each transitive place associated to low (high) level of an output signal z_O is connected by read-arcs to all negative (positive) transitions of z_O in the environment. The transitions of z_O in the environment part are replaced by dummies. Now the environment also tracks the operation of the device. For convenience each dummy introduced in this step are labelled by the original transition name put in parenthesis.

The transformation of the second step splits each transition of an output (input) signal into the signal transition itself which belongs to the device (environment) and a dummy in the environment (device). The firing of these two transitions are ordered by read-arcs, so that the interface signal transition is enabled first and only after this transition fires the corresponding dummy is enabled. The dummy transition cannot be disabled until it fires because of the burst USC property of the original STG. Thus the behavioural equivalence is preserved on this step of transformation.

The third step of the transformation is splitting the device into *tracker* and *bouncer* parts as shown in Figure 4.3(d). There is no need to further transform the environment part as only the device will be subsequently implemented. The tracker-bouncer splitting starts from representing each output signal by an elementary cycle. An *elementary cycle* of a signal z consists of two places $z = 0$ and $z = 0$ (these are transitive places added in the first transformation step), and several positive and negative transitions of z connecting these places. The positive transitions of z are inserted after $z = 0$ and before $z = 1$. Similarly, the negative transitions of z are inserted

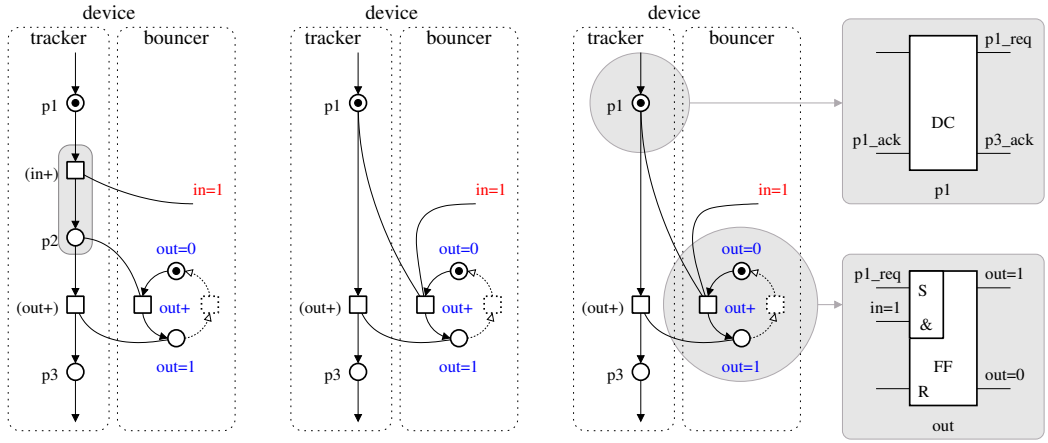
after $z = 1$ and before $z = 0$. The number of positive and negative transitions of z is equal to the number of corresponding events in the device specification and can be more than one. The set of elementary cycles for all output signals forms the device bouncer and the rest is the device tracker. The elementary cycles of the bouncer are synchronised with the tracker part by means of read arcs as shown in Figure 4.3(d). Each positive (negative) transition t_B of signal z in the bouncer is uniquely associated to a positive (negative) transition t_T of the same signal z in the tracker. A transition t_T is called a *prototype* of t_B. All places in the preset of t_T are connected by read-arcs to t_B and the only place in the postset of t_B is connected by a read-arc to t_T. After that the prototype transition t_T in the tracker is replaced by a dummy which is labelled by the original transition name in parenthesis.

The transformation described in the third step basically splits each output signal transition $z\pm$ into the signal transition $z\pm$ itself (in the bouncer) and a dummy $(z\pm)$ (in the tracker). The transition $z\pm$ is enabled only when all the places in the preset of $(z\pm)$ have tokens. These tokens cannot propagate further because the dummy is disabled by a read-arc from the postset of $z\pm$. The only place p_z in the post of $z\pm$ is either $z = 0$ or $z = 1$ depending on the polarity of $z\pm$. As soon as $z\pm$ fires the dummy $(z\pm)$ becomes enabled and the tokens continue their move in the tracker. It is also necessary that the token does not leave p_z until $(z\pm)$ fires. This condition is ensured by the signal consistency of the initial STG. Thus the transformation of this step preserves the behavioural equivalence of the system.

From this point the device model is considered separately and the environment is assumed to produce inputs in response to device outputs according to the system protocol. The elementary cycles of the device bouncer are subsequently implemented as set-reset Flip-Flops (FF) and the places of the device tracker are mapped into DCs, see Figure 4.3(e).

4.1.3 Optimisation

It is often possible to control outputs by the directly preceding interface signals without using intermediate states. Many places and preceding dummies can thus be removed, provided that the system behaviour is preserved w.r.t. input-output interface (weak bisimulation). Such places are called *redundant*. Note that the notion of redundant places in our method is different from the



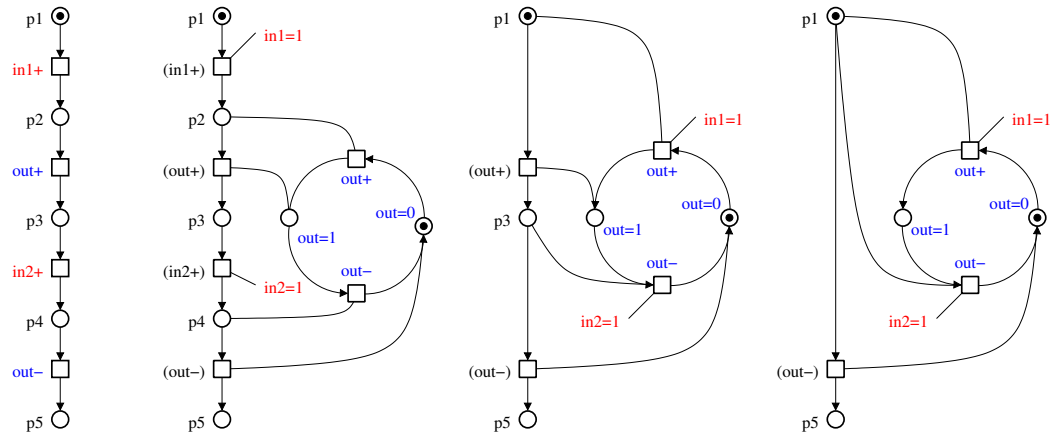
(a) Redundancy (b) Optimisation (c) Mapping into circuit

Figure 4.4: Optimisation of the device specification

redundant transitive places in the structural theory of Petri nets, thus the structural theory cannot be applied to remove them. This way $p2$ is redundant in the considered example, Figure 4.4(a). It can be removed from the device tracker together with the preceding dummy $(in+)$ as shown in Figure 4.4(b). Now the input $in = 1$ controls the output $out+$ transition directly, which results in latency reduction when the STG is mapped into the circuit, see Figure 4.4(c). Before the optimisation the output flip-flop was set by the $p2\_req$ signal, which was generated in response to the input in, see Figure 4.3(e). In the optimised circuit the output flip-flop is triggered directly by the in input and the context signal $p1\_req$ is calculated in advance, concurrently with the environment action.

4.1.4 Coding conflicts

The elimination of places is restricted by potential *coding conflicts* which may cause tracking errors. There are two types of conflicts: *Marked Graph-specific* and *State Machine-specific*. The former conflicts may appear in a non-conflicting branch of an STG, the latter may appear in the conflicting branches after a choice place.



(a) System (b) Output exposure (c) Correct optimisation (d) Coding conflict

Figure 4.5: Preventing coding conflicts

Marked Graph-specific coding conflicts

For the idea of an Marked Graph-specific coding conflict, consider the system whose STG is depicted in Figure 4.5(a). The device specification extracted from this STG by applying the above method is shown in Figure 4.5(b). The tracker part of the device can be further optimised. The removal of redundant places $p2$ and $p4$ does not cause any conflicts of the tracker, Figure 4.5(c). However, if the place $p3$ is eliminated as shown in Figure 4.5(d), then the tracker cannot distinguish between the output having not yet been set and the output already reset. Note the specifics of this direct mapping approach: only those signals whose switching directly precedes the given output are used in its support.

It is computationally simpler to detect redundant places by processing the original specification. For this the set of all STG places P is divided into three non-intersecting subsets: P_U, P_R, P_M such that $P_U \cup P_R \cup P_M = P$ and $P_U \cap P_R \cap P_M = \emptyset$. The set P_R consists of redundant places which can be safely removed from the device STG, the set P_M holds the mandatory places which must be preserved in the device model, and the set P_U contains the places which have not been considered yet (undefined places). In the following figures the undefined places are depicted as ordinary circles (○), redundant places are drawn as small circles (∘), and the mandatory places are shown as bold circles (**O**). Initially all STG places belong to P_U, both sets P_R and P_M are

empty. Then, each place in P_U is tested for being redundant. If the place removal does not cause a coding conflict then the place is redundant and is moved into P_R, otherwise it is mandatory and is moved into P_M.

A coding conflict for a place p is detected by intersecting two sets of signals. The first set contains the signals whose transitions are fired in the *forward neighbourhood* of place p limited by places and transitions in $P_R \cup T$. The second set consists of the signals whose transitions are fired in the *backward neighbourhood* of place p limited by places and transitions in $P_R \cup T$.

The *forward neighbourhood* $\varepsilon_f\,(p,\,X)$ of place p limited by nodes (places and transitions) in X is defined as the minimal (w.r.t.$\subseteq$) set such that:

$$\begin{aligned} \varepsilon_f\,(p,\,X):\quad & p \in \varepsilon_f\,(p,\,X)\,; \\ & \forall\, x \in X\ if\ \exists\, y \in \varepsilon_f\,(p,\,X):\ x \in y\bullet,\ then\ x \in \varepsilon_f\,(p,\,X) \end{aligned} \tag{4.1}$$

The set of signals $Z_f(p,\,X)$ whose transitions are fired in the forward neighbourhood of place p limited by the set of nodes X are defined using labelling function λ:

$$Z_f(p,\,X) = \{z:\ \exists t \in \varepsilon_f\,(p,\,X) \cap T:\ \lambda(t) \in \{z{+},\,z{-}\}\} \tag{4.2}$$

Similarly, the *backward neighbourhood* $\varepsilon_b\,(p,\,X)$ of place p limited by the set of nodes X is defined as the minimal (w.r.t.$\subseteq$) set such that:

$$\begin{aligned} \varepsilon_b\,(p,\,X):\quad & p \in \varepsilon_b\,(p,\,X)\,; \\ & \forall\, x \in X\ if\ \exists\, y \in \varepsilon_b\,(p,\,X):\ x \in \bullet y,\ then\ x \in \varepsilon_b\,(p,\,X) \end{aligned} \tag{4.3}$$

$$Z_b(p,\,X) = \{z:\ \exists t \in \varepsilon_b\,(p,\,X) \cap T:\ \lambda(t) \in \{z{+},\,z{-}\}\} \tag{4.4}$$

For the detection of a coding conflict the forward and backward neighbourhoods are calculated on a set of transitions and redundant places. If $Z_f\,(p,\,P_R \cup T)\ \cap\ Z_b\,(p,\,P_R \cup T)\ =\ \emptyset$, then the removal of the place p does not cause coding conflicts. However, if there is a signal z whose transitions belong to both the forward and the backward neighbourhoods of place p limited by transitions and redundant places, then the removal of this place causes a coding conflict for signal z. The state of the signal z is the same before its transition in the backward neighbourhood and

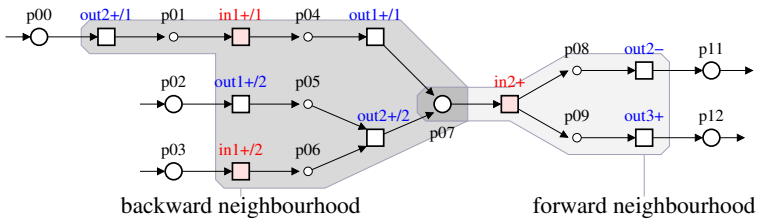


Figure 4.6: Forward and backward neighbourhoods

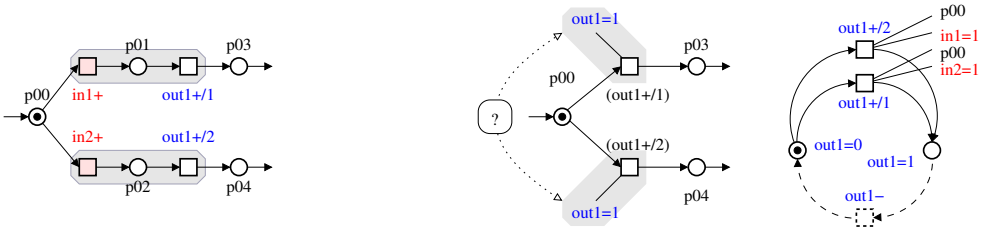
after its transition in the forward neighbourhood of place p. As the place p is the only undefined place between these transitions (the others are redundant) it must be preserved in order to separate the same state of the signal z in different parts of the system specification.

Consider the detection of coding conflict on the example shown in Figure 4.6. The place under question is *p07*. Its forward neighbourhood limited by redundant places is *{p07, in2+, p08, p09, out2-, out3+}* and its backward neighbourhood is *{p07, out1+/1, out2+/2, p05, p06, p04, in1+/1, out1+/2, in1+/2, p01, out2+/1}*. Places *p00*, *p02*, *p03*, *p11*, *p12* are not redundant and form a border for the place *p07* neighbourhoods. The signals whose transitions are fired in the forward and backward neighbourhoods are *{in2, out2, out}* and *{in1, out1, out2}* respectively. The intersection of these sets is *{out2}* which means that place *p07* separates the different states of the signal *out2* and removal of this place will cause a coding conflict.

State Machine-specific coding conflicts

The situation becomes more complex if a place under test belongs to a post-postset of a free-choice place. For example, places *p01* and *p02* in Figure 4.7(a) are not redundant. If these places are removed, then the choice between the conflicting branches is controlled by the same condition *out1=1*, which is ambiguous, see Figure 4.7(b). This is an State Machine-specific coding conflict. The forward and backward neighbourhoods do not help here because the places whose removal cause a coding conflict are in mutually exclusive branches.

There are several strategies to avoid the ambiguity in choice branches after choice place p_{choice}. The first extreme is to preserve all post-postset places of the choice place. This approach is computationally simple, however the latency reduction might be sacrificed if there is an input signal transition in the postset of the choice place (which is the case for a free-choice). The other ex-



(a) Initial STG

(b) An ambiguouty caused by removal of places p01 and p02

Figure 4.7: Mandatory places after choice

treme is to traverse each conflicting branch from p_{choice} and find all signals whose transitions are present in more than one branch. At least one mandatory place must precede the first transition of such signals in each branch. This approach helps to reduce the input-output latency, but it may be computationally hard if loops or nested choices are found.

A trade-off between the computation speed and latency optimisation is the following. First, each undefined place p_{succ} in the post-postset of p_{choice} is checked. The removal of place p_{succ} does not improve the input-output latency if all preset transitions of p_{succ} are non-input signal events. Such a place is made mandatory to help reducing the computation complexity of the next step. After that, for each transition $t \in p_{choice}\bullet$ a set of transitions $T_{seq}(t) = \{t\} \cup t\bullet\bullet$ is built. It contains the transitions in a choice branch starting from t for the depth of two transitions counting from p_{choice}. The joint set of transitions in choice branches of place p_{choice} limited by the depth of two transitions is $T_{choice}(p_{choice}) = \bigcup_{t \in p_{choice}\bullet} T_{seq}(t)$. If for a transition $t \in p_{choice}\bullet$ there is a signal whose transition belongs to both $T_{seq}(t)$ and $T_{choice}(p_{choice}) \setminus T_{seq}(t)$ then the places in the postset of t must be preserved. Otherwise all places in the postset of $t\bullet\bullet$ transitions are made mandatory to reduce the depth of traversing. The trade-off approach benefits from the input-output latency reduction and low computation complexity, however the size optimisation might suffer.

Consider the application of the trade-off approach to the example in Figure 4.7(a). For the choice place $p00$, which has transitions $in1+$ and $in2+$ in its postset, $T_{seq}(in1+) = \{in1+, out1+/1\}$, $T_{seq}(in2+) = \{in2+, out1+/2\}$ and $T_{choice}(p00) = \{in1+, out1+/1, in2+, out1+/2\}$. As a transition of $out1$ belongs to both $T_{seq}(in1+)$ and $T_{choice}(p00) \setminus T_{seq}(in1+) = T_{seq}(in2+)$ the place $p01$is mandatory. Similarly, a transition of

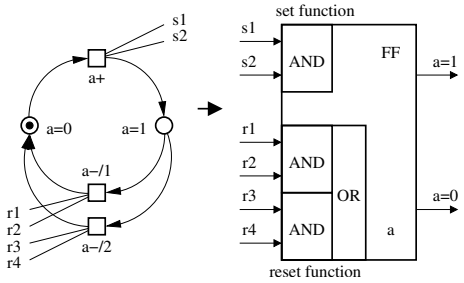


Figure 4.8: Mapping of elementary cycles into FFs

$out1$ belongs to both $T_{seq}\,(in2+)$ and $T_{choice}\,(p00)\setminus T_{seq}\,(in2+) = T_{seq}\,(in1+)$, which makes place $p02$ mandatory.

4.1.5 Mapping into circuit

In this method the places of the tracker STG are mapped into DCs and the elementary cycles of the bouncer are mapped into set-reset FFs. Read-arcs connecting the elementary cycles to the tracker and to the environment model the behaviour of wires and are directly mapped into wires between DCs and FFs. The set and reset functions of a FF are mapped from the structure of the set and reset phases of the corresponding elementary cycle as shown in Figure 4.8.

The traditional DC circuit is described in Section 3.7, where the method of LPN place mapping is reviewed. However, the mapping of high-level LPN places is different from the mapping of STG places. The former approach uses a place preset transition to request for an operation in the environment and to acknowledge the completion of the previous stage operation. The latter approach uses the bouncer elementary cycles to interface with the environment. The tracker keeps the state of the system only. Its marked places select the set of output signals which can switch in the current state of the system. The preset transitions of each tracker place are used to request the tracker to switch into the next state.

The mapping of basic tracker structures into DCs is shown in Figure 4.9. The request and acknowledgement functions of each DC are generated from the structure of the tracker in the preset and postset of the corresponding place. The request function of each DC is shown in its top-left corner and the acknowledgement function in its bottom-right conner.

Faster and more compact solutions for a DC implementation, proposed in [18], are called

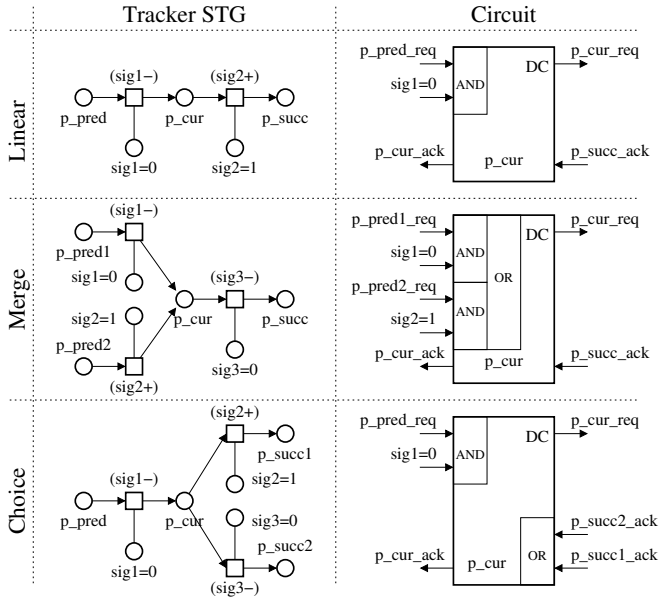
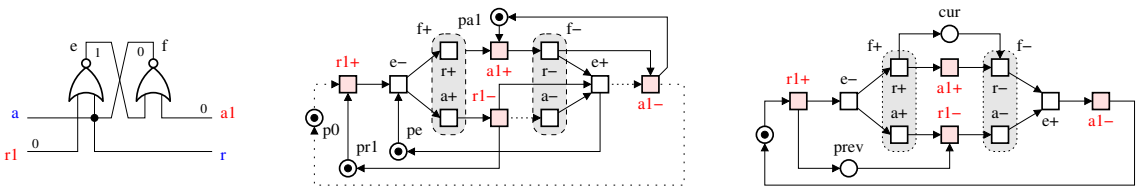


Figure 4.9: Mapping of tracker places into DCs



(a) Circuit (b) STG (c) Timing assumptions

Figure 4.10: Gate-level fast DC

fast DCs. A gate-level implementation of a fast DC and its STG are shown in Figure 4.10(a,b). The input *r1* is the request from the previous stage DC to pass the token. Output *a* acknowledges the receipt of the token to the previous stage DC. Similarly, output *r* requests the next stage DC to accept the token and input *a1* acknowledges its receipt. Signals *e* and *f* represent the 'empty' and 'full' states of the state holding element. An interesting feature of the fast DC is that it internally contains a GasP-like interface [111], which uses a single wire to transmit a request in one direction and an acknowledgement in the other.

A fast DC has speed advantages over a traditional DC because the reset phase of its state holding element happens concurrently with the token move into the next stage DC. However, fast DCs rely on timing assumptions. The timing assumptions are depicted in the STG using dotted

arcs, see Figure 4.10(b).

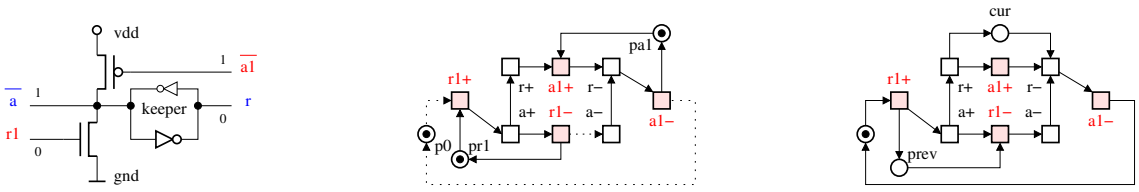
The first timing assumption is represented by the dotted arcs incident to place *p0*. The assumption is that the new token arrives only after the token has left the next stage DC. This assumption is common for both traditional and fast DCs. It results in a limitation of the method to have at least three DCs in every loop [59]. For the original specification it means that any loop of the system STG must contain at least three places. Also the number of places in the loops must be kept above two during the optimisation of the tracker STG.

The second timing assumption is that the token leaves the DC only after the previous stage DC is empty. It is shown by the dotted arc between transition *r1-* and *a-* in Figures 4.10(b). The same timing assumption for the next stage DC is shown by the dotted arc from *e+* to *a1-*. This assumption is easy to meet because the reset of the request *r1-* from the previous stage DC is delayed by a single two-input NOR-gate. The acknowledgement *a1+* from the next stage DC is set with the delay of at least a pair of two-input NOR-gates.

Accepting the above timing assumptions and removing the transitive places the simplified STG of the fast DC is obtained in Figure 4.10(c). The transitive places *prev* and *next* represent the high level of signals $r1$ and r respectively. Their state denotes the marking of places associated with the previous stage and next stage DCs. One can see that both *prev* and *next* are marked for the time $t_{a+ -\!\!\rightarrow r1-}$ in each cycle of DC operation. This inconsistency between the underlying PN model and fast DCs is called the *token spread.*

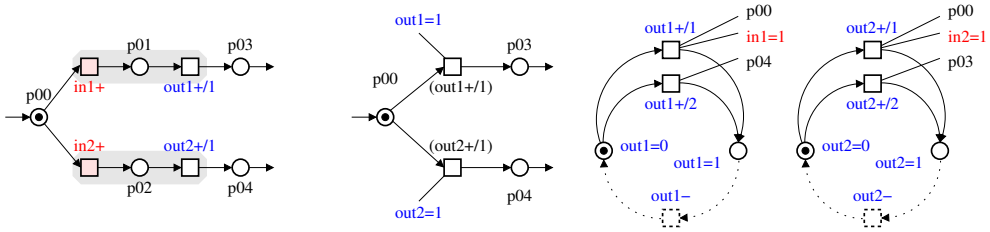
Another implementation of a fast DC and its STG are shown in Figure 4.11(a,b). This implementation uses a *keeper* latch for the state holding element. A *keeper* is a logic level hold circuit which consists of two weak inverters connected back to back. In order to increase the driving ability of the request output, the weak inverter providing this output is replaced by an ordinary inverter. The timing assumptions for the transistor-level implementation are the same as for the gate-level. One can see that the spread of the token is also possible in the transistor-level implementation, see Figure 4.11(c).

The token spread is not modelled by the underlying PN, which may cause problems in the vicinity of the choice place. Consider the example STG shown in Figure 4.12(a). The transitions that directly succeed places *p01* and *p02* are different signal events, and the removal of these



(a) Circuit (b) STG (c) Timing assumptions

Figure 4.11: Transistor-level fast DC



(a) Initial STG (b) Resultant STG

Figure 4.12: Redundant places after choice

places does not cause any coding conflict. The optimised STG shown in Figure 4.12(b) can be safely mapped into traditional DCs. However, the direct mapping into fast DCs is problematic due to their token spread feature.

For example, if the STG shown in Figure 4.12(b) is implemented using fast DCs, then the following scenario is possible: *in1+*→*out1+/1*→*(out1+/1)*→*out2+/1* resulting in the token spread over places *p00* and *p03* for a short time interval. It leads to the incorrect state when transitions *(out2+/1)* and *(out2+/2)* in conflicting branches are enabled simultaneously. The firing of the enabled transition *(out2+/1)* results in the malfunction of the system: both conflicting branches are active at the same time.

A possible solution for the token spread problem is to restrict the propagation of a token in conflicting branches until the token leaves the choice place. It can be done by mapping the first places in the conflicting branches into traditional DCs. Such a DC does not rise its request output until the request of the previous stage DC is low. The application of this approach to the example shown in Figure 4.12(b) forces places *p03* and *p04* to be mapped into traditional DCs.

DC type	min period (ns)	max frequency (MHz)	size (transistor count)	token spread
traditional	3.6	277.8	12	no
fast gate-level	1.2	833.3	8	yes
fast transistor-level	2.1	476.2	6	yes

Table 4.1: Comparison of DC implementations

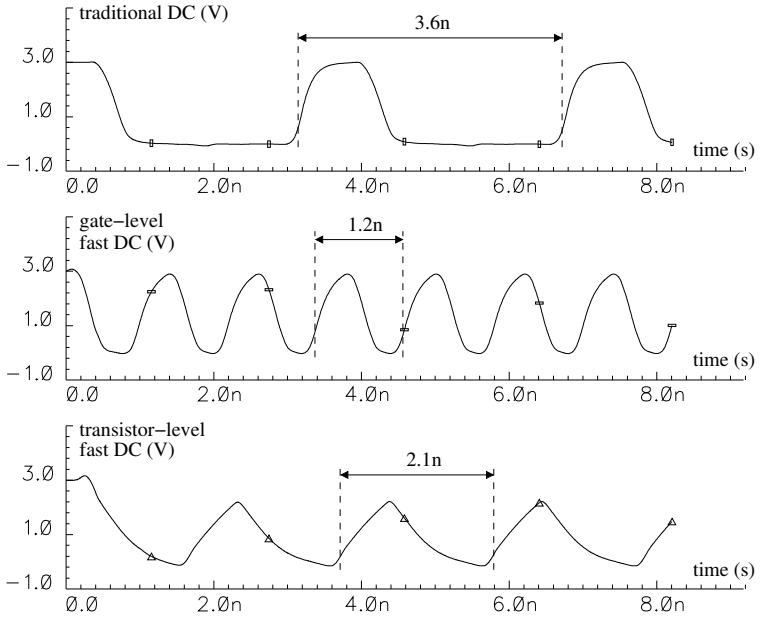


Figure 4.13: Speed of DC implementations

The advantages and drawbacks of different DC implementations are summarised in Table 4.1. Only a traditional DC is free of the token spread problem. The smallest (6 transistors) is the transistor-level fast DC. The fastest (up to 833.3MHz) is the gate-level fast DC.

The maximum frequency of DC operation is measured by SPICE analog simulations using the AMS-0.35μ design kit. For this, traditional DC, gate-level fast DC and transistor-level fast DC have been implemented in AMS-0.35μ library. Then the DCs of each type have been connected in loops of tree DCs and the oscillation of each loop have been captured as shown in Figure 4.13.

The shortest period is exhibited by gate-level fast DCs. These DCs are the best for the synthesis of fast control circuits. Transistor-level fast DCs are recommended when the circuit size is crucial, however they require extra effort for the layout of the library of custom cells. Both types of fast DCs rely on timing assumptions and have certain token spread problems. That is why the

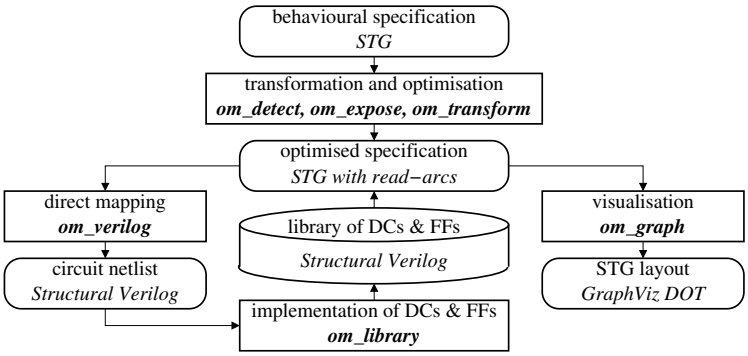


Figure 4.14: OptiMist design flow

traditional DCs should be used for the design of speed-independent circuits and to avoid the spread of a token in the vicinity of choice places.

4.2 Algorithms

This section describes the algorithms employed in the STG optimisation for mapping. The algorithms are implemented in a package of software tools called OptiMist whose design flow is presented in Figure 4.14.

The package consists of separate tools solving the following tasks:

- detection of redundant places;
- exposure of the outputs;
- elimination of redundant places;
- visualisation of an STG with read-arcs extension and tracker-bouncer structure;
- mapping of the optimised specification into a circuit;
- generation of a library of required DCs and FFs either at transistor- or gate-level.

The algorithms for STG optimisation (detection of redundant places, exposure of the outputs and elimination of redundant places) are described in detail in the following subsections. The algorithms for the other three tools are trivial and are not presented here.

Algorithm 3 Detection of redundant places

```
01  procedure detect_redundant_places
02  input: STG = ⟨P, T, F, M_0, I, O⟩, optimisation_level ∈ {0,1,2,3}
03  output: P_R ⊆ P
04  P_R:=∅; P_M:=∅; P_U:=P
07  if optimisation_level > 0 then
08      optimise_choice (STG, P_U, P_R, P_M)
09      if optimisation_level > 1 then
10          optimise_latency (STG, P_U, P_R, P_M)
11          if optimisation_level > 2 then
12              optimise_size (STG, P_U, P_R, P_M)
```

4.2.1 Detection of redundant places

The order in which the redundant places are detected affects the optimisation result. The order is defined by the heuristics presented in Algorithm 3. The *detect_redundant_places* procedure takes *STG* and *optimisation_level* as the input parameters and returns the set P_R of redundant places (lines 01-03). Initially, all STG places are undefined (line 04). The *optimisation_level* parameter defines which optimisation heuristics to apply to the STG (lines 07-12).

Choice optimisation

The heuristic *optimise_choice* whose pseudo-code is shown in Algorithm 4 prevents the State Machine-specific coding conflicts. The algorithm implements a trade-off between the computation speed and latency optimisation as described in Section 4.1.4. The input of the heuristic is the system STG and initial partitioning of its places into undefined, redundant and mandatory; its output is a new partitioning of the places (lines 01-03).

First, for each choice place p the set T_{choice} containing its postset and post-postset transitions is created (lines 05-07). Then, for each transition t in the postset of choice place p a set T_{conf} is computed (lines 08-09). It contains all transitions in the conflicting branches which are in the postset or post-postset of the choice place p. For each place p_{succ} in the postset of transition t a set T_{seq} containing t and all transitions in the postset of p_{succ} (lines 10-11). If there is a signal whose transition belongs to both T_{seq} and T_{conf} then place p_{succ} is mandatory (lines 12-14). Otherwise place p_{succ} is redundant and all places in its post-postset are made mandatory to reduce the computation complexity (lines 15-18).

Algorithm 4 Choice optimisation

```
01  procedure optimise_choice
02  input: STG = ⟨P, T, F, M0, I, O⟩, PU ⊆ P, PR ⊆ P, PM ⊆ P
03  output: PU ⊆ P, PR ⊆ P, PM ⊆ P
04  for_each p ∈ P : |p•| > 1 do
05      Tchoice := ∅
06      for_each t ∈ p• do
07          Tchoice := Tchoice ∪ {t} ∪ t••
08      for_each t ∈ p• do
09          Tconf := Tchoice \ ({t} ∪ t••)
10          for_each psucc ∈ t• : psucc ∈ PU do
11              Tseq := {t} ∪ psucc•
12              if ∃(z ∈ I ∪ O, tseq ∈ Tseq, tconf ∈ Tconf) :
13                  (tseq ∈ {z+, z−}) ∧ (tconf ∈ {z+, z−}) then
14                  PM := PM ∪ {psucc}
15              else
16                  PR := PR ∪ {psucc}
17                  for_each psucc_succ ∈ psucc•• : psucc_succ ∈ PU do
18                      PM := PM ∪ {psucc_succ}, PU := PU \ {psucc_succ}
19              PU := PU \ {psucc}
```

Latency optimisation

The heuristic *optimise_latency* is aimed at latency reduction. Its basic idea is that a place is redundant if all its direct predecessors are input transitions and all direct successors are non-input transitions. Such a place can be considered redundant even without checking for the possibility of a coding conflict. All its surrounding places are undefined yet which means that the backward neighbourhood includes input transitions only and the forward neighbourhood contains non-input transitions only. Thus, the intersection of the sets of signals in these neighbourhoods is always empty which means the place under question is redundant.

The *optimise_latency* pseudo-code is shown in Algorithm 5. The input of the heuristic is the system STG and initial partitioning of its places into undefined, redundant and mandatory obtained by *optimise_choice* algorithm; its output is a new partitioning of the places (lines 01-03). The algorithm finds all undefined places whose preset transitions are input events and postset transitions are not (lines 04-05). Such places are moved from the set of undefined places into the set of redundant places (lines 06-07).

Size optimisation

The *optimise_size* heuristic, whose pseudo-code is presented in Algorithm 6, is aimed at size reduction. The input of the heuristic is the system STG and the partitioning of places into undefined,

Algorithm 5 Input-output latency optimisation

```
01  procedure optimise_latency
02  input: STG = ⟨P, T, F, M0, I, O⟩, PU ⊆ P, PR ⊆ P, PM ⊆ P
03  output: PU ⊆ P, PR ⊆ P, PM ⊆ P
04  for_each p ∈ PU do
05      if (∀t ∈ •p, λ(t) ∈ I × {+,−}) ∧ (∀t ∈ p•, λ(t) ∉ I × {+,−}) then
06          PR:=PR ∪ {p}
07          PU:=PU \ {p}
```

Algorithm 6 Size reduction

```
01  procedure optimise_size
02  input: STG = ⟨P, T, F, M0, I, O⟩, PU ⊆ P, PR ⊆ P, PM ⊆ P
03  output: PU⊆ P, PR ⊆ P, PM ⊆ P
04  // Process sequences of undefined places
05  while ∃p ∈ PU : (εb(p, PR ∪ PM ∪ T) ∩ P = {p}) ∧ (εf(p, PU ∪ T) ∩ P = {p}) do
06      if Af(p, PR) ∩ Ab(p, PR) = ∅ then
07          PR:=PR ∪ {p}
08      else
09          PM:=PM ∪ {p}
10      PU:=PU \ {p}
11  // Process remaining undefined places individually
12  for_each p ∈ PU do
13      if Af(p, PR) ∩ Ab(p, PR) = ∅ then
14          PR:=PR ∪ {p}
15      else
16          PM:=PM ∪ {p}
17      PU:=PU \ {p}
```

redundant and mandatory subsets obtained by *optimise_choice* and *optimise_latency* heuristics; its output is the new partitioning of places (lines 01-03). The heuristic is divided into two steps: first, the redundant places are detected in the chains of undefined places (lines 04-10); then, the undefined places left in the STG are checked for redundancy individually (lines 11-17).

At the first step, for each undefined place its backward neighbourhood limited by undefined places and forward neighbourhood limited by non-undefined places are found. If the number of places contained in these neighbourhoods is equal to one (the place under question itself) then this place is a *boundary* place between non-undefined and undefined places (line 05). Such a place is subject for redundancy check (line 06). If for this place there is no signal whose transitions are fired in both forward and backward neighbourhoods limited by redundant places, then the place is redundant (line 07). Otherwise it is mandatory (line 09). The procedure is repeated until no boundary places left in the STG.

At the second step all undefined places which left in the STG are checked for redundancy without any specific order (lines 11-17). The majority of redundant places are already detected in

the previous heuristics. The places left undefined in the STG are usually those preceding the input signal transitions and their removal does not improve the latency, however the size reduction is still possible.

4.2.2 Exposure of outputs

The conversion of the system STG into a two-level architecture is described by Algorithm 7. The input to the algorithm is a system STG and the initial states S of input and output signals; its output is a modified STG which consists of a tracker and a bouncer (lines 01-03). The initially empty set of read-arcs R connecting the tracker and bouncer is added to the STG (line 04). For each STG signal z a place representing its low level p_z^{low} and a place representing its high level p_z^{high} are created (lines 05-07). The initial marking of these places is chosen according to the initial state S of signal z (lines 08-11). Then, each transition t of signal z is substituted by a dummy transition t_{dummy} in the tracker part (lines 12-18). The signal transition itself is moved into the bouncer part, thus forming the signal elementary cycle (lines 19-23). The tracker and bouncer operation is synchronised by means of read-arcs which are inserted in such way that signal transition t is enabled only when all direct predecessors of the dummy transition t_{dummy} are marked, see read-arcs inserted in lines 24-25. The dummy transition t_{dummy} itself is only enabled when the signal transition t is fired and the marking of signal z elementary cycle is changed, see read-arcs inserted in lines 21 and 23.

4.2.3 Elimination of redundant places

Algorithm 8 describes the procedure of redundant places elimination. This procedure should be used after detection of redundant places and exposure of outputs. It consists of three steps: initial marking optimisation, trigger signals optimisation and context signal optimisation (lines 04-06).

The procedure of initial marking optimisation is shown in Algorithm 9. It changes the initial marking in such way that no redundant places contain tokens. For this the marking of each redundant place is traversed one transition back assuming that all the places in its postset are marked (lines 04-15). The exception is made for merge places because it is hard to compute which conflicting branch produced the token for the merge place (line 07). The back traversal repeats until

Algorithm 7 Conversion of a system STG into a tracker-bouncer architecture

```
01  procedure convert_tracker_bouncer
02  input: STG = ⟨P, T, F, M_0, I, O⟩, S : I ∪ O → {0,1}
03  output: STG = ⟨P, T, F, R, M_0, I, O⟩
04  R:=∅
05  for_each z ∈ I ∪ O do
06      create place p_z^low, create place p_z^high
07      P:=P ∪ {p_z^low} ∪ {p_z^high}
08      if S(z) = 0 then
09          M_0(p_z^low):=1, M_0(p_z^high):=0
10      else
11          M_0(p_z^low):=0, M_0(p_z^high):=1
12      for_each t ∈ T : λ(t) ∈ {z+, z−} do
13          // Substitute signal z transitions by dummies (in the tracker)
14          create dummy transition t_dummy, T:=T∪{t_dummy}
15          for_each p ∈ •t do
16              F:=F ∪ {(p, t_dummy)}, F:=F \ {(p,t)}
17          for_each p ∈ t• do
18              F:=F ∪ {(t_dummy, p)}, F:=F \ {(t,p)}
19          // Move signal z transitions into elementary cycle (in the bouncer)
20          if λ(t) = z+ then
21              F:=F ∪ {(p_z^low, t)}, F:=F ∪ {(t, p_z^high)}, R:=R ∪ {(p_z^high, t_dummy)}
22          else
23              F:=F ∪ {(p_z^high, t)}, F:=F ∪ {(t, p_z^low)}, R:=R ∪ {(p_z^low, t_dummy)}
24          for_each p ∈ •t_dummy do
25              R:=R ∪ {(p,t)}
```

Algorithm 8 Elimination of redundant places

```
01  procedure optimise
02  input: STG = ⟨P, T, F, R, M_0, I, O⟩, P_R ⊆ P
03  output: STG = ⟨P, T, F, R, M_0, I, O⟩
04  optimise_marking (STG, P_R)
05  optimise_bouncer (STG, P_R)
06  optimise_tracker (STG, P_R)
```

Algorithm 9 Re-calculation of the initial marking from redundant places to mandatory

```
01  procedure optimise_marking
02  input: STG = ⟨P, T, F, R, M0, I, O⟩, P_R ⊆ P
03  output: STG = ⟨P, T, F, R, M0, I, O⟩, P_R ⊆ P
04  // Try to recalculate the initial marking to mandatory places
05  repeat
06      done:=true
07      for_each p ∈ P_R : (M0(p) ≠ 0) ∧ (|•p| = 1) do
08          for_each t ∈ •p do
09              if ∀p_conc ∈ t• : M0(p_conc) = 1 then
10                  for_each p_pred ∈ •t do
11                      M(p_pred):=1
12                  for_each p_conc ∈ t• do
13                      M(p_conc):=0
14                  done:=false
15  until done
16  // Make all marked places mandatory
17  for_each p ∈ P_R : M(p) ≠ 0 do
18      P_R:=P_R\{p}, P_M:=P_M ∪ {p}
```

either only mandatory places are marked or there is no such transition preceding a marked redundant place whose postset contains places that are all marked. If some redundant places are still marked after the marking recalculation they are made mandatory (lines 16-18). The recalculation of the initial marking for the merge places can be improved by employing the reachability analysis algorithms. However, they require either building a finite prefix or a reachability graph which is computationally complex for large specifications.

An auxiliary procedure removing an STG node together with its incident arcs is described by Algorithm 10. It is moved to a separate algorithm in order to lighten the *optimise_bouncer* and *optimise_tracker* pseudo-code. The input of the *remove_node* algorithm is an STG and its node which is required to remove. The removal of node *x* starts from the elimination of its producing and consuming arcs (lines 04-07). If *x* is a place, then the read-arcs from this place to all transitions are removed and the node is subtracted from the set of STG places (lines 08-11). If *x* is a transition, then read-arcs from all places to this transition are removed and *x* is removed from the set of STG transitions (lines 12-15).

The pseudo-code for optimisation of context and trigger signals in the bouncer part is shown in Algorithm 11. It changes the read-arcs connecting the tracker with the bouncer in such a way that only mandatory places control the transitions of elementary cycles. In order to do this for each transition t_{read} which is controlled by a redundant place p its copy t_{read}^{dup} is created and its consuming and producing arcs are duplicated (lines 06-09).

Algorithm 10 Node removal together with its incident arcs

```
01 procedure remove_node
02 input: STG = ⟨P, T, F, R, M_0, I, O⟩, P_R ⊆ P, x ∈ P ∪ T
03 output: STG = ⟨P, T, F, R, M_0, I, O⟩, P_R ⊆ P
04 for_each y ∈ •x do
05     F:=F \ {(y, x)}
06 for_each y ∈ x• do
07     F:=F \ {(x, y)}
08 if x ∈ P then
09     for_each y ∈ x⋆ do
10         R:=R \ {(x, y)}
11     P:=P \ {x}, P_R:=P_R \ {x}
12 else
13     for_each y ∈ ⋆x do
14         R:=R \ {(y, x)}
15     T:=T \ {x}
```

Algorithm 11 Optimisation of the bouncer context and trigger signals

```
01 procedure optimise_bouncer
02 input: STG = ⟨P, T, F, R, M_0, I, O⟩, P_R ⊆ P
03 output: STG = ⟨P, T, F, R, M_0, I, O⟩
04 T_trig:=∅
05 for_each t_read ∈ p⋆ : p ∈ P_R do
06     // Duplicate t_read together with its consuming and producing arcs
07     create transition t_read^dup, T:=T ∪ {t_read^dup}
08     for_each p_read_pred ∈ •t_read do F:=F ∪ {(p_read_pred, t_read^dup)}
09     for_each p_read_succ ∈ t_read• do F:=F ∪ {(t_read^dup, p_read_succ)}
10     for_each t ∈ •p do
11         // Form trigger signals for t_read^dup
12         if t_read ∉ T_trig then
13             for_each p_read ∈ ⋆t do R:=R ∪ {(p_read, t_read^dup)}
14             T_trig:=T_trig ∪ {t_read^dup}
15         // Form context signals for t_read^dup
16         create transition t^dup, T:=T ∪ {t^dup}
17         F:=F ∪ {(t^dup, p)}, F:=F \ {(t, p)}
18         for_each p_pred ∈ •t do
19             create place p_pred^dup, P:=P ∪ {p_pred^dup}
20             R:=R ∪ {(p_pred^dup, t_read^dup)}
21             for_each t_pred_succ ∈ p_pred• : t_pred_succ ≠ t do F:=F ∪ {(p_pred^dup, t_pred_succ)}
22             for_each t_pred_pred ∈ •p_pred do F:=F ∪ {(t_pred_pred, p_pred^dup)}
23             for_each t_pred_read ∈ p_pred⋆ do R:=R ∪ {(p_pred^dup, t_pred_read)}
24             F:=F ∪ {(p_pred^dup, t^dup)}
25         // Remove processed transition t and its preset places
26         if |t•| = 0 then
27             for_each p_pred ∈ •t do
28                 remove_node (STG, p_pred)
29             remove_node (STG, t)
30     remove_node (STG, t_read)
```

In our method only those signals whose transitions directly precede an output transition form the set of its triggers. Line 12 checks if transition t_{read} is controlled by a trigger signal yet. If it is not then trigger signals are introduced by means of read-arcs connecting t^{dup}_{read} to each place which controls a transition in the preset of place p (line 13). Transition t^{dup}_{read} is added to the set T_{trig} containing transitions which are already controlled by trigger signals (line 14).

Recalculation of the context signals involves some change in the tracker structure. Each transition t in the preset of the redundant place p is copied to t^{dup} (line 16). Then producing arc $\left(t^{dup},\, p\right)$ is added and arc $(t,\, p)$ is deleted, thus removing redundant place p from the postset of t (line 17) and each place p_{pred} in the preset of transition t is copied to p^{dup}_{pred} (lines 18-19). This place p^{dup}_{pred} is used by the transition t^{dup}_{read} in the elementary cycle as a new context signal instead of place p, see read-arc $\left(p^{dup}_{pred}, t^{dup}_{read}\right)$ in line 20. All arcs incident to place p_{pred} except consuming arc (p_{pred}, t) are copied to similar arcs connected to place p^{dup}_{pred} (lines 21-23). The consuming arc (p_{pred}, t) is mapped into arc $\left(p^{dup}_{pred}, t^{dup}\right)$ (lines 24).

If the redundant place p was the only place in the postset of transition t then this transition is removed together with its preset places and their incident arcs (lines 26-29). Finally the transition t_{read} is removed together with its consuming, producing and read-arcs. If there are other read-arcs from redundant places to t_{read} they have been copied into the read-arcs to t^{dup}_{read} and are processed in the next iterations of the algorithm. If there are redundant places in the pre-preset of place p these are also processed in the next iterations.

After the application of *optimise_marking* and *optimise_bouncer* algorithms to the device STG its redundant places are not marked with tokens and do not control any transition by means of read-arcs. These places can be removed now by the procedure whose pseudo-code is shown in Algorithm 12. Each redundant place p is removed individually with the required change of the tracker structure. For each transition t in the preset of p a copy t^{dup}_{succ} of the each transition t_{succ} in the postset of p is created (lines 06-10). The copy transition t^{dup}_{succ} is added to the T^{dup} which contains all transitions which are duplicated for t. Incident arcs of t_{succ} except of consuming arc (p, t_{succ}) are also copied (lines 11-13). After that each place in the preset of t is connected by a consuming arc with each transition in T^{dup} (lines 14-17). When all transitions in the preset of p are processed, the transitions in the postset of p and the redundant place p itself are removed

Algorithm 12 Optimisation of the tracker by redundant places removal

```
01  procedure optimise_tracker
02  input: STG = ⟨P, T, F, R, M0, I, O⟩, P_R ⊆ P
03  output: STG = ⟨P, T, F, R, M0, I, O⟩
04  for_each p ∈ P_R do
05      // Duplicate p• transitions and their incident arcs
06      for_each t ∈ •p do
07          T^dup := ∅
08          for_each t_succ ∈ p• do
09              create transition t_succ^dup
10              T := T ∪ t_succ^dup, T^dup := T^dup ∪ t_succ^dup
11              for_each p_conc ∈ •t_succ : p_conc ≠ p do F := F ∪ {(p_conc, t_succ^dup)}
12              for_each p_succ ∈ t_succ• do F := F ∪ {(t_succ^dup, p_succ)}
13              for_each p_read ∈ ⋆t_succ do R := R ∪ {(p_read, t_succ^dup)}
14          // Connect •t places with all T^dup transitions
15          for_each p_pred ∈ •t do
16              for_each t_succ^dup ∈ T^dup do
17                  F := F ∪ {(p, t_succ^dup)}
18      // Remove redundant place p, p• transitions and processed •p transitions
19      for_each t ∈ •p do
20          F := F \ {(t, p)}
21          if |t•| = 0 then
22              remove_node(STG, t)
23      for_each t_succ ∈ p• do
24          remove_node(STG, t_succ)
25      remove_node(STG, p)
```

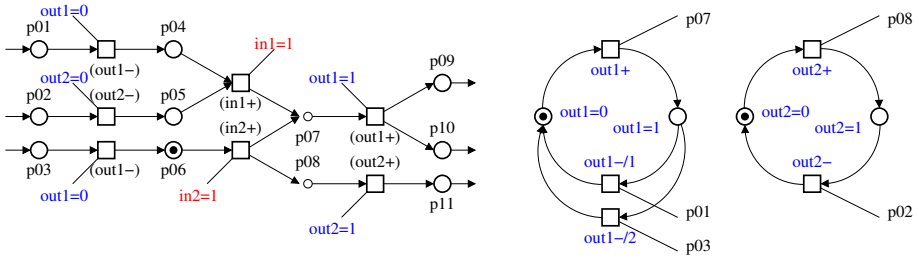
(lines 18-25). If redundant place p was the only place in the postset of t then transition t is also removed.

The procedure of redundant places removal is described using a simple example whose STG is shown in Figure 4.15(a). Only places *p07*, *p08* are redundant and the initial marking does not require recalculation. Redundant places *p07* and *p08* control transitions *out1+* and *out2+* respectively. New context and trigger signals for each transition are found by *optimise_bouncer* algorithm. Its result is shown in Figure 4.15(b).

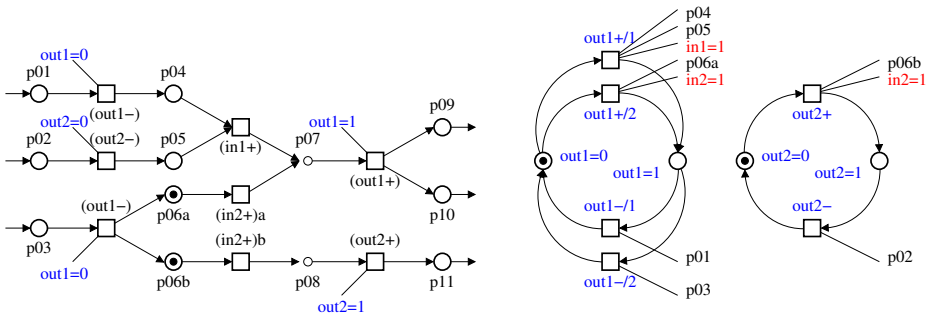
Dummy *(in2+)* is split into *(in2+)a* and *(in2+)b* because it precedes two redundant places. The place *p06* which is in the preset of *(in2+)* is also split into *p06a* and *p06b*, so that *p06a* precedes *(in2+)a* and *p06b* precedes *(in2+)b*.

Only the *(in2+)b* dummy precedes redundant place *p08*. The context and trigger signals of the transition *out2+* are defined by the preset of place *p08*. Its trigger consists of place *in2=1* which controls *(in2+)b* and its context is formed by place *p06b* which is in preset of *(in2+)b*. Read-arcs from *in2=1* to *(in2+)b* and from *p06* to *out2+* are removed.

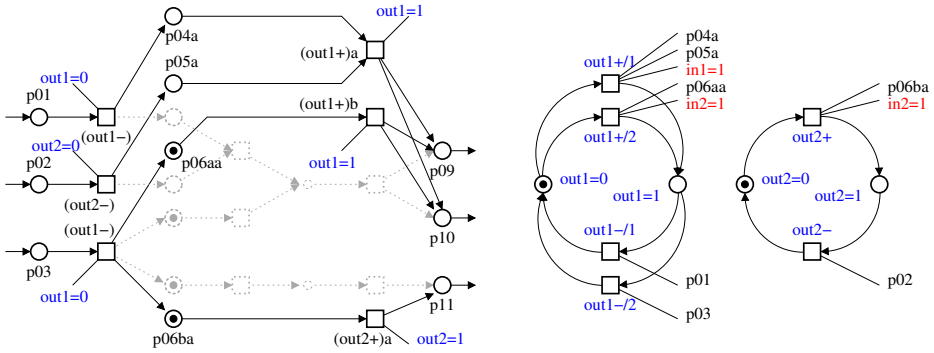
Two dummies *(in1+)* and *(in2+)a* precede redundant place *p07* which means there are two



(a) Initial STG with redundant places



(b) Bouncer optimisation



(c) Tracker optimisation

Figure 4.15: Removal of redundant places

mutually exclusive sets of triggers/context signals for *out1+*. For this reason the *out1+* transition is duplicated. The trigger of its first copy *out1+/1* is the place *in1=1* which controls *(in1+)*; its context is provided by places *p04* and *p05* which are in preset of *(in1+)*. The trigger of the second copy *out1+/2* is place *in2=1* which controls *(in2+)a*; its context signals places *p06a* which is in preset of *(in2+)a*. Read-arcs from *p07* to *out1+,* from *in1=1* to *(in1+)* and from *in2=1* to *(in2+)a* are removed.

Redundant places *p07* and *p08* are then removed from the STG using *optimise_tracker* algorithm whose result is shown in Figure 4.15(c). Note that dummy *(out1+)* is split into *(out1+)a* and *(out1+)b*. There are two transitions in the preset of *p07*, for each of them a copy of *p07* postset is created.

The algorithms presented in this section are implemented in the OptiMist toolkit. The toolkit automates the mapping of STGs into circuits. At the same time it gives a designer full control on the choice of optimisation heuristics and allows manual adjustment of the solution to specific requirements. Appendix A describes the tools comprising the OptiMist toolkit and their command-line parameters. OptiMist can be employed in combination with Cadence to allow simulation and technology mapping of circuits. A basic library of DCs and FFs has been created for Cadence. It can be expanded, if necessary, using a tool from the OptiMist package which generates a Verilog netlist for DCs and FFs at transistor-level or gate-level.

The results presented in Section 4.3 and Section 4.4 are obtained by OptiMist tools.

4.3 GCD controller example

Consider the use of the OptiMist tools on the example of the GCD control unit. Its STG is obtained by refining the LPN generated from the HDL specification by the PN2DCs tool, see Figure 3.10(c). In order to produce the control unit STG shown in Figure 4.16 the events of the LPN are expanded to a 4-phase handshake protocol. After that, the GCD datapath schematic shown in Figure 3.14 is taken into account for manually adjusting the STG to the datapath interface. In the modified STG the request to the comparator *cmp_req* is acknowledged, in a 1-hot code, by one of the signals: *gt_ack*, *eq_ack* or *lt_ack*. The request to the subtracter *sub_gt_req* is acknowledged by *x_ack*. This is possible because the procedure of storing the subtraction result into the register is

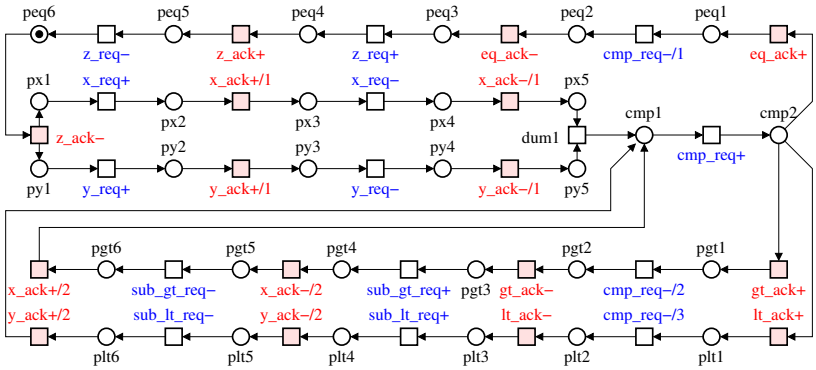


Figure 4.16: STG of GCD controller

controlled directly in the datapath and does not involve the control unit. Similarly *sub_lt_req* is acknowledged by *ack_y*. The obtained STG is stored in the `gcd.g` file.

The redundant places are detected in the original STG before the exposure of outputs by the `om_detect` tool. The first heuristic applied to the GCD example is *optimise_choice*. It prevents coding conflicts from occurring in choice conflicting branches without restricting the latency reduction. The result of this heuristic is shown in Figure 4.17(a). It is obtained by the following command:

```
$ om_detect --level1 --output gcd_1.g gcd.g
```

There are three places *pgt1*, *peq1* and *plt1* which are in the post-postset of the free-choice place *cmp2*. All of them are mandatory because they are preceding the transitions of the same signal *cmp_req*. Making these places mandatory reduces the input-output latency for *eq_ack+*→*cmp_req-/1*, *gt_ack+*→*cmp_req-/2* and *lt_ack+*→*cmp_req-/3* handshakes. However, it is the only way to avoid a coding conflict.

The second heuristic *optimise_latency* reduces both the size and the latency of a circuit. The redundant places detected by this heuristic in the GCD example are *px1*, *py1*, *px3*, *py3*, *px5*, *py5*, *pgt3*, *plt3*, *pgt5*, *plt5*, *peq3* and *peq*5, see Figure 4.17(b). The preset of each of these places contains transitions of input signals only and the postset contains transitions of non-input signals. The command executed for detecting these redundant places is:

```
$ om_detect --level2 --output gcd_2.g gcd.g
```

The last heuristic *optimise_size* detects redundant places *cmp2, pgt2*, *plt2* and *peq2* in the GCD example, see Figure 4.17(c). Removal of place *cmp1* also does not cause a coding conflict, how-

ever it is kept by the `om_detect` tool in order to preserve the simplicity of the *cmp_req* elementary cycle. Without this place the positive phase of the *cmp_req* would be controlled by two context signals from the tracker (read-arcs from *px4* and *py4*) and two trigger signals from the environment (read-arcs from places *x_ack=0* and *y_ack=0*). The trade-off between the complexity of elementary cycles and the number of places in the tracker can be set by command line parameters of the `om_detect` tool:

```
$ om_detect --level3 --join2 --output gcd_3.g gcd.g
```

After the detection of redundant places the `om_expose` tool partitions the STG of GCD control path into tracker and bouncer parts:

```
$ om_expose --output gcd_3e.g gcd_3.g
```

The resultant STG is shown in Figure 4.18. The bouncer consists of elementary cycles representing the outputs of GCD controller, one cycle for each output. The elementary cycles for the inputs are not shown as they belong to the environment. The tracker is connected to inputs and outputs of the system by means of read-arcs, as it is described in the algorithm of outputs exposure.

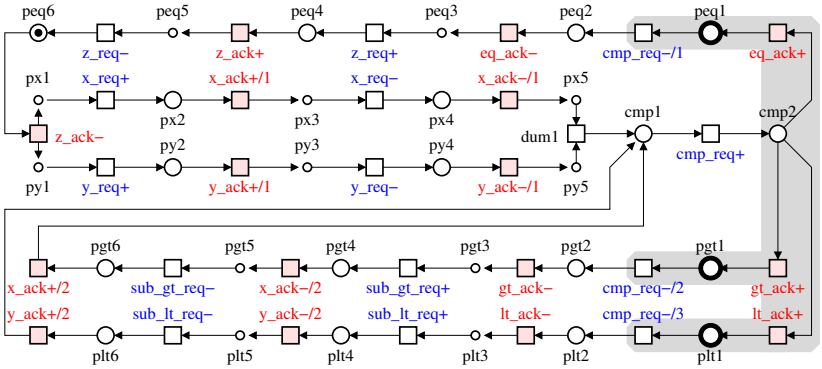
After the redundant places are detected and the outputs are exposed, the STG is optimised by removing the redundant places from the tracker part. The removal of a place involves the change in the STG structure but preserves the behaviour of the system w.r.t. input-output interface. The result of GCD control unit optimisation is presented in Figure 4.19. This operation is automatically performed by the `om_transform` tool:

```
$ om_transform --level5 --output gcd_3et.g gcd_3e.g
```

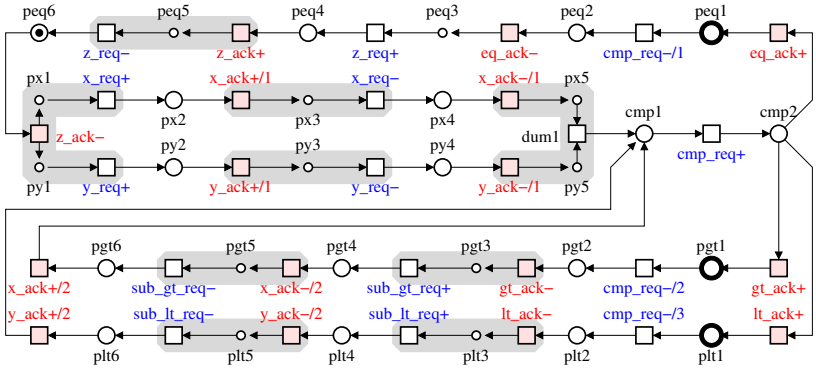
This STG can now be used for circuit synthesis. For this each tracker place is mapped into a DC and each elementary cycle is mapped into a FF. The request and acknowledgement functions of a DC are mapped from the structure of the tracker in the vicinity of the corresponding place. The set and reset functions of a FF are mapped from the structure of the set and reset phases of the corresponding elementary cycle. The GCD controller circuit obtained by this technique is presented in Figure 4.20. The netlist is produced automatically by the following command:

```
$ om_verilog gcd_3et.g --statistics --output gcd.v
```

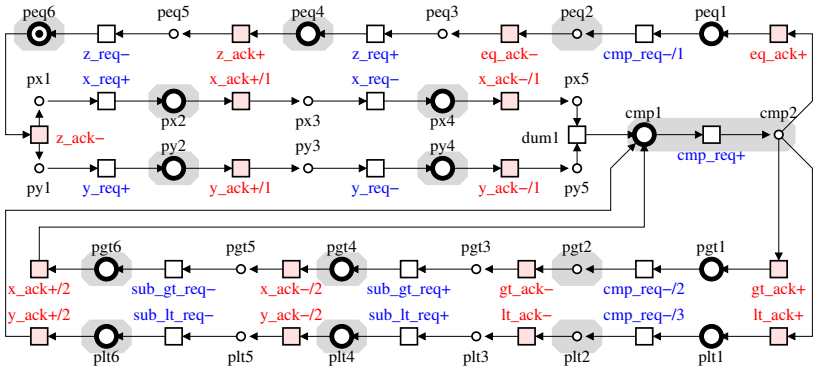
This circuit consists of 15 DCs and 6 FFs. If the DCs are implemented as transistor-level fast DCs then the maximum number of transistor levels in pull-up and pull-down stacks is 4. This transistor stack appears in the request function of the DC for *cmp1* and is formed by the signals



(a) Choice optimisation



(b) Latency optimisation



(c) Size optimisation

Figure 4.17: Detection of redundant places in STG of GCD controller

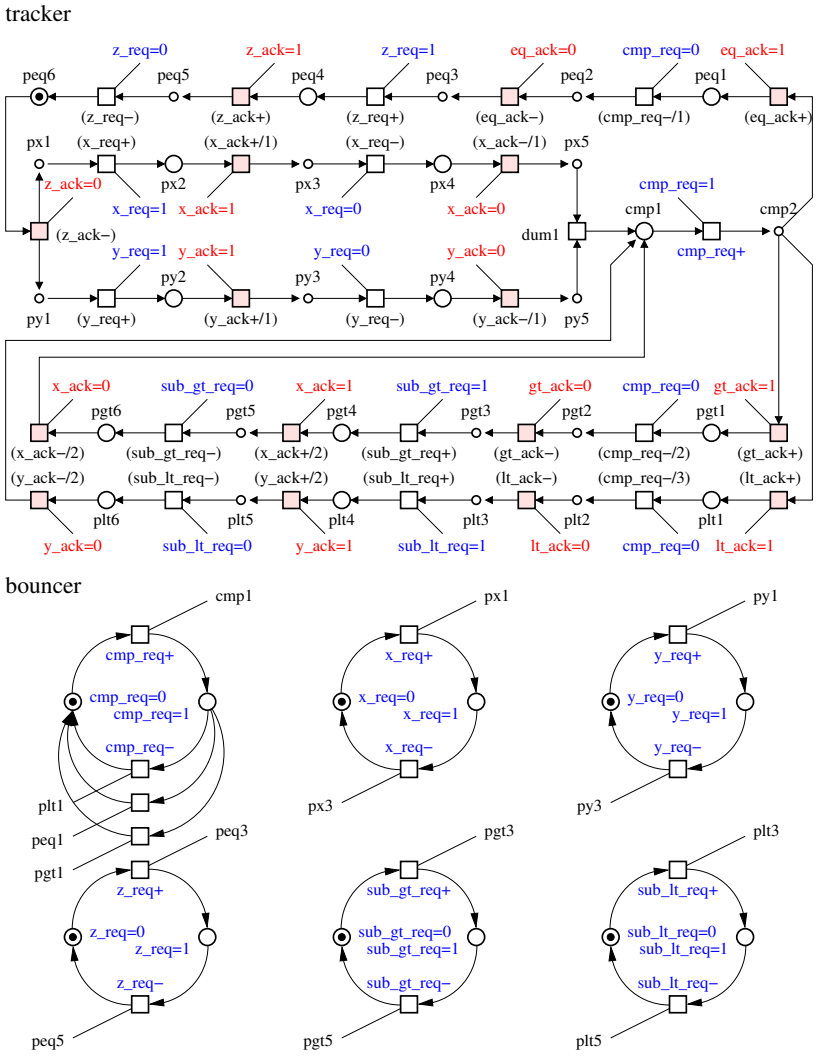


Figure 4.18: Exposure of outputs in STG of GCD controller

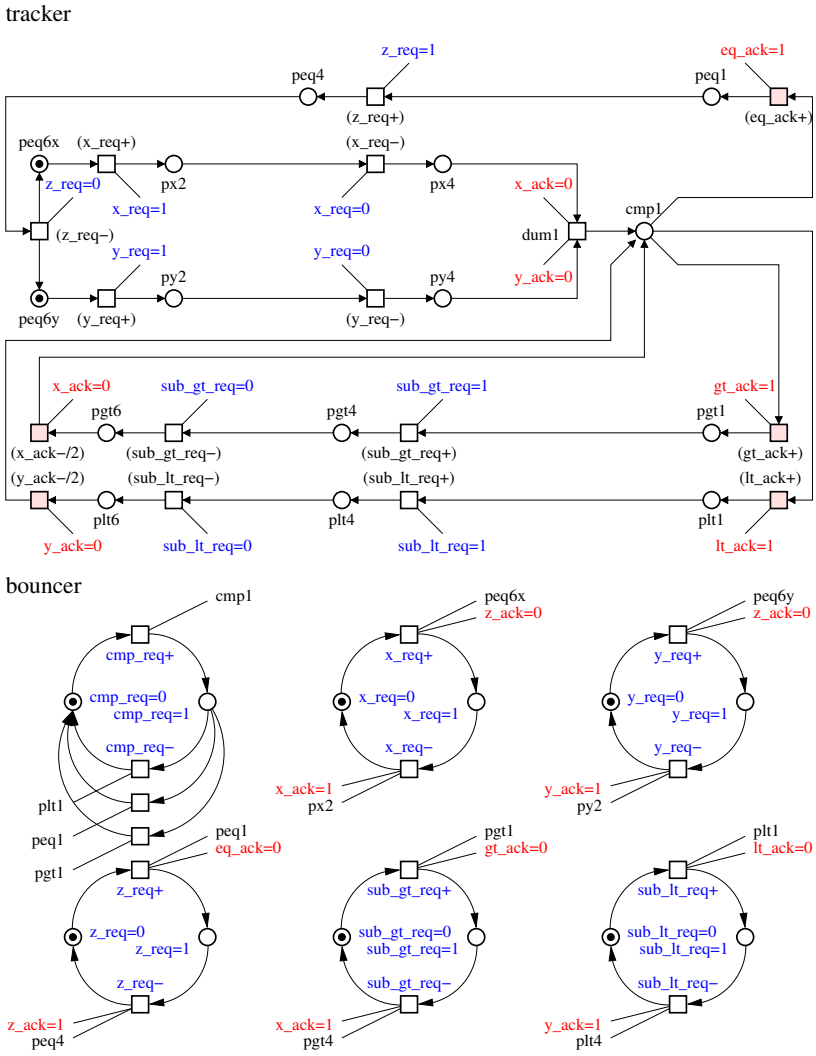


Figure 4.19: Elimination of redundant places in STG of GCD controller

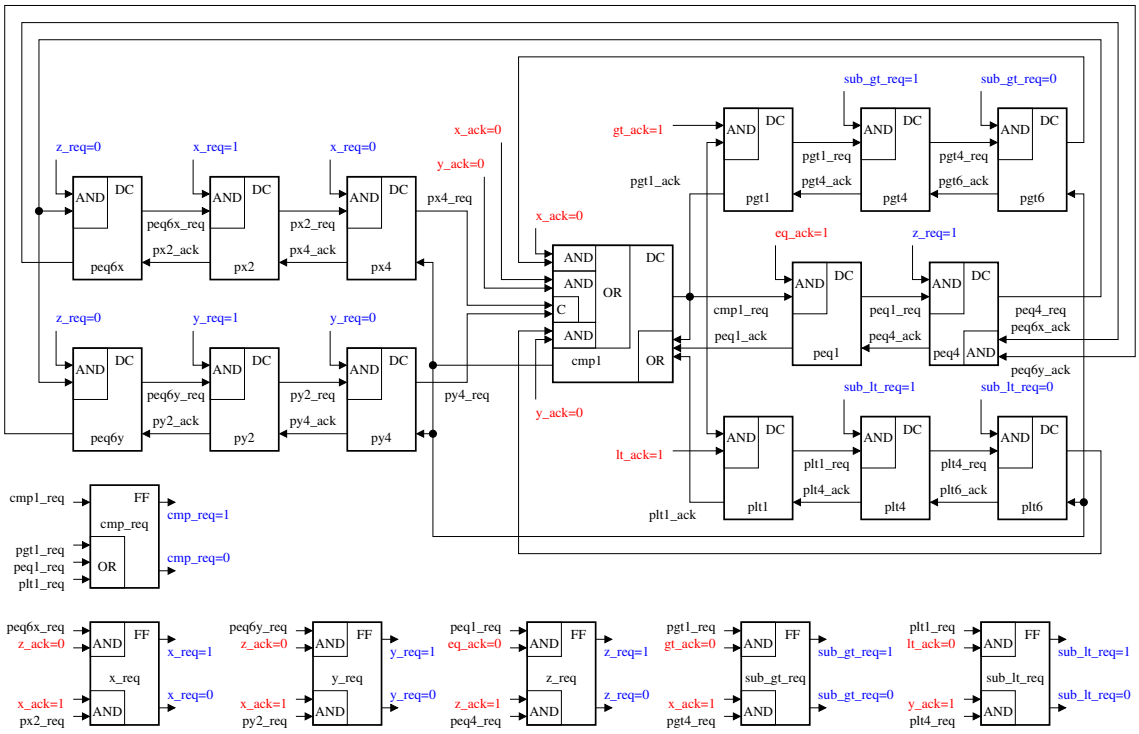


Figure 4.20: GCD controller circuit obtained by the OptiMist tool

x_ack=0, *y_ack=0*, *px4_req* and *py4_req*.

The longest latency, which is the delay between an input change and reaction of the controller by changing some outputs, is exhibited by *cmp_req* signal. The latency of its set and reset phases is equal to the delay of one DC and one FF. The other outputs are triggered directly by input signals which means that their latencies are equal to one FF delay plus the delay of one inverter when the trigger signal requires inversion.

4.4 Benchmarks

This section highlights the advantages and drawbacks of the direct mapping approach implemented in OptiMist for a set of benchmarks. The direct mapping approach is compared against explicit logic synthesis (implemented in Petrify) in terms of circuit size and speed. The complexity of the underlying algorithms is taken into account by measuring the computation time of OptiMist and Petrify on Pentium 3 1GHz, 1Gb RAM computer. The effect of optimisation heuristics on the direct mapping is also analysed. For this comparison each benchmark STG has been synthesised

benchmark name	DC count	max fin	max fout	transistor count	worst-case latency	computation time
gcd						
OptiMist (no optimisation)	30	2	4	255	4.5	0.11s
OptiMist (latency&size optimisation)	14	3	4	174	4.5	0.18s
Petrify				116	11.0	18s
vme-bus						
OptiMist (no optimisation)	17	3	3	155	5.0	0.09s
OptiMist (latency&size optimisation)	10	3	4	121	5.0	0.11s
Petrify				58	8.5	1s
toggle						
OptiMist (no optimisation)	8	2	2	68	4.5	0.06s
OptiMist (latency&size optimisation)	4	2	2	44	3.5	0.07s
Petrify				22	3.5	0.12s
imec-alloc-outbound						
OptiMist (no optimisation)	17	2	2	143	5.0	0.09s
OptiMist (latency&size optimisation)	6	2	5	73	3.0	0.16s
Petrify				46	7.5	6.6s
par3						
OptiMist (no optimisation)	19	3	4	160	6.5	0.07s
OptiMist (latency&size optimisation)	15	4	4	114	4.5	0.09s
Petrify				78	12.5	11s
count						
OptiMist (no optimisation)	19	3	3	150	5.5	0.07s
OptiMist (latency&size optimisation)	11	3	4	98	3.0	0.11s
Petrify (manual CSC resolution)				68	3.0	1.4s

Table 4.2: Comparison between OptiMist and Petrify

in three different ways:

- Direct mapping by the OptiMist tools without detection and elimination of redundant places;
- Direct mapping by the OptiMist tools with latency and size optimisation by removing the redundant places from the STG;
- Logic synthesis by the Petrify tool with automatic resolution of CSC conflicts (unless it is impossible) and logic decomposition into gates with at most four literals.

The result of the experiment is summarised in Table 4.2.

The number of transistors is counted for the case of places being implemented as fast DCs, request-acknowledgement logic of DCs and set-reset logic of FFs being implemented at transistor

level. The condition of having at least three DCs in a loop is met.

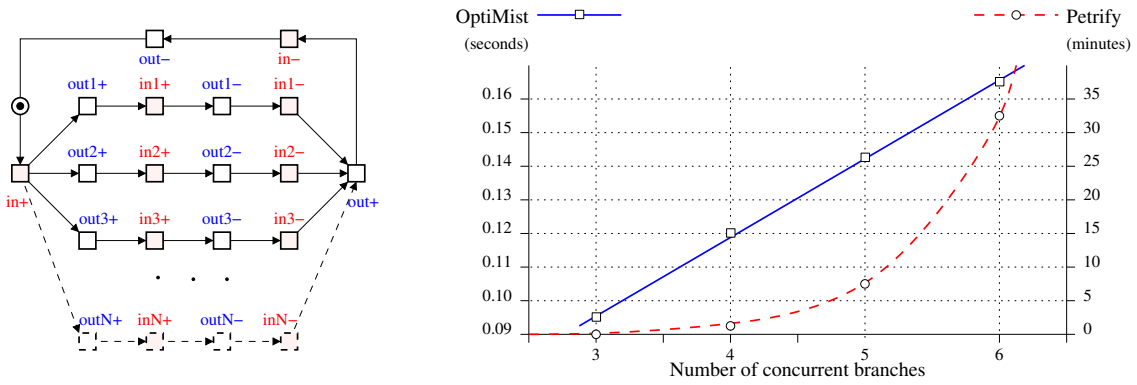
In all experiments, the latency is counted as the accumulative delay of negative gates switched between an input and the next output. The following dependency of a negative gate delay on its complexity is used. The latency of an inverter is associated with a unit delay. Gates which have maximum two transistors in their transistor stacks are associated with 1.5 units; 3 transistors - 2.0 units; 4 transistors - 2.5 units. This approximate dependency is derived from the analysis of the gates in AMS 0.35μm library. The method of latency estimation does not claim to be very accurate. However, it takes into account not only the number of gates switched between an input and the next output, but also the complexity of these gates.

All experiments show the high efficiency of direct mapping optimisation heuristics. About 50% of DCs are redundant in the original STG. Their removal results in up to 35% improvement in the circuit size. The latency of the circuits also benefits from the optimisation. In some cases (*gcd*, *vme-bus*) the worst-case latency cannot be improved because of a potential coding conflict in the conflicting branches. However, this latency is only exhibited by the first output signal after the choice place. The latency of the other outputs is reduced.

The comparison of the circuits obtained by OptiMist (with latency&size optimisation) and Petrify shows that the direct mapping solutions are usually larger than logic synthesis solutions. However the circuits obtained by the direct mapping technique exhibit lower output latency.

For some benchmarks (e.g *count*) Petrify fails to resolve a CSC conflict even if it is reducible. Manual insertion of additional signals is required in such cases. However, OptiMist completes the job automatically for such benchmarks.

The OptiMist tools can also process large specifications, which are not computable by Petrify in acceptable time. This can be illustrated on the scalable benchmark whose STG is shown in Figure 4.21(a). Adding the concurrent branches as shown by dashed lines one can increase the complexity of the benchmark. When the concurrency increases, the Petrify computation time grows exponentially, while the OptiMist computation time grows linearly on the same benchmark, see Figure 4.21(b). Note the different time scale for OptiMist (seconds) and for Petrify (minutes).



(a) Scalable benchmark STG

(b) Computation time

Figure 4.21: Dependency of computation time on STG complexity

4.5 Summary

A method for direct mapping of STGs into circuit netlist has been presented in this chapter. The method exploits the two-level architecture where a circuit consists of two blocks: the tracker and the block of output flip-flops. The tracker computes context signals for outputs concurrently with the environment operation, thus achieving the latency reduction effect. The output flip-flops generate outputs from context and trigger signals. The adopted architecture allows the minimisation of state-holding elements and reduction of latency. The characteristic feature of the method is that the optimisation is achieved at the specification (Petri net) level as opposed to optimisation of logic circuits after the synthesis stage.

The method is implemented in a package of software tools called OptiMist. The package take an STG as the initial specification of a system, converts the STG in a form convenient for mapping, performs optimisation, and produces a Verilog netlist of the circuit. The optimisation of the specification relies on a set of heuristics aimed at circuit latency and size reduction. This package can be employed in combination with Cadence for simulation and technology mapping of circuits.

In the OptiMist tools the optimisation is performed locally and the computation time grows linearly with the size of specification. This allows to process large specifications which are not computable by logic synthesis tools in acceptable time.

The OptiMist tools are fully automated. At the same time a designer can significantly influence the result by choosing one or more optimisation heuristics. In combination with computation speed OptiMist gives the designer an opportunity to synthesise circuits with different optimisation parameters and choose the best solution.

Chapter 5

Synthesis of data path

Synthesis of an asynchronous data path by a direct mapping approach has two levels of granularity. At the higher level, which is reviewed in Chapter 3, the entire data path is obtained by mapping its CPN fragments into hardware components which implement the corresponding mathematical functions. A library of hardware components is produced at the lower level of data path synthesis. Such a library can either be developed by modifying the standard RTL solutions to the asynchronous style manually, or using computationally complex logic synthesis methods [114]. The former approach is restricted by the manual intervention. The latter is still in its infancy and produces solutions that are inefficient in terms of speed and area.

In this chapter a method for automated synthesis of the data path components by direct mapping is presented. It is based on a conventional RTL design flow, similar to the NCL-X approach [62]. Each data path component is first implemented using standard RTL synthesis tools, e.g Synopsys. The obtained RTL circuit is mapped into a hazard-free (having no glitches due to race conditions) logic using a dual-rail encoding with a return-to-spacer signalling. The hazard-free logic facilitates low power consumption and high speed of the circuit. This return-to-spacer signalling is convenient for detecting the completion of computation and producing an acknowledgement signal for the environment.

The dual-rail encoding and the return-to-spacer protocol are also successfully used in security circuits. With this protocol a dual-rail circuit has the same number of switchings per computation cycle independent of processed data. For example, in [89] a secure Amulet core for smartcard

applications is build using dual-rail logic. However, the physical implementation of the rails at the gate level is not symmetric and the use of a standard return-to-spacer switching protocol may still leak secrete data. A new alternating-spacer protocol is proposed in this chapter. It has two spacers interchanging in time, which makes all gates switch per computation cycle. The full potential of this protocol is still to be learnt. One of its applications, the energy balancing for security circuits, is studied in Chapter 6.

The rest of the chapter is organised as follows. Section 5.1 introduces a method for synthesis of data path components using dual-rail encoding and a new alternating-spacer signalling protocol. Section 5.2 presents the converters between single-rail and dual-rail logic domains, converters between single-spacer and alternating-spacer protocols and different implementations of dual-rail flip-flops and latches. A design kit which implements the method is discussed in Section 5.4. Its operation is studied on a 4-bit adder example in Section 5.5.

This chapter is based on results presented in [106, 101]. The whole method presented in this chapter, the VeriMap software tool implementing this method, and individual solutions for converters, flip-flops and latches were tested by several practical designs for our industry partner Atmel Inc.

5.1 Method

Our method for synthesis of the data path components is based on a direct mapping approach and has low algorithmic complexity. Its main idea is to stay as close to the standard industry design flow as possible. The method is applied via an automated tool to netlists obtained by standard RTL synthesis tools from a behavioural specifications ('push-button' approach).

The method can also be applied to existing clocked architectures, dominated by synchronous single-threaded CPU cores and their slow buses, having no pipelining or concurrency. The combinational logic of such data path is transformed into hazard-free dual-rail logic, the registers are replaced by predesigned dual-rail flip-flops, the interface to the single-rail environment is preserved by special converters. The resultant circuit be can desynchronised by replacing the clock signal by a completion detection signal, thus obtaining a self-timed solution.

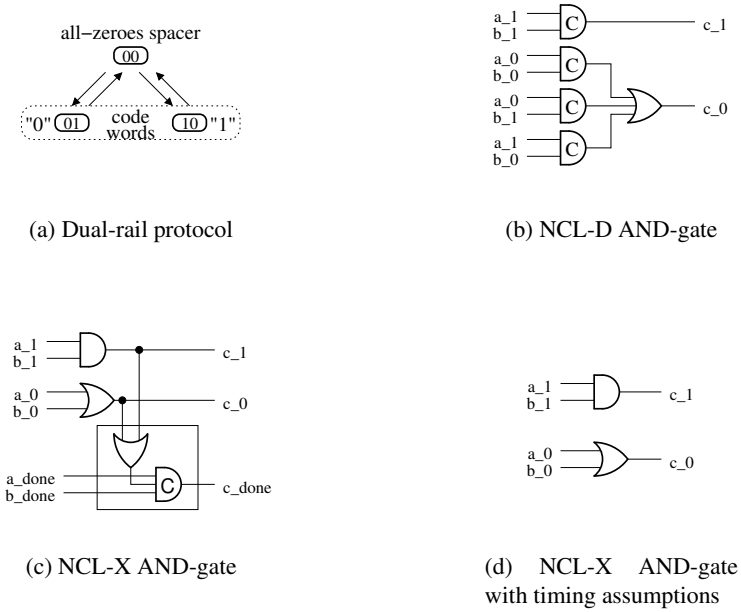
5.1.1 Single-spacer dual-rail

Dual-rail code uses two rails with only two valid signal combinations $\{01, 10\}$, which encode values 0 and 1 respectively. Dual-rail code is widely used to represent data in self-timed circuits [118, 33], where a specific protocol of switching helps to avoid hazards. The protocol allows only transitions from all-zeroes $\{00\}$, which is a non-code word, to a *code word* and back to all-zeroes as shown in Figure 5.1(a); this means the switching is monotonic. The all-zeroes state is used to indicate the absence of data, which separates one code word from another. Such a state is often called a *spacer* and the switching discipline is called a *single-spacer protocol*.

An approach for automatic conversion of single-rail circuits to dual-rail, using the above signalling protocol, that is easy to incorporate in the standard RTL-based design flow has been described in [62]. Within this approach, called Null-Convention Logic [41] one can follow either of two major implementation strategies for logic: one is with full completion detection through the dual-rail signals (NCL-D) and the other with separate completion detection (NCL-X). The former is more conservative with respect to delay dependence while the latter is less delay-insensitive but more area and speed efficient. For example, an AND gate is implemented in NCL-D and NCL-X as shown in Figure 5.1(b,c) respectively. NCL methods of circuit construction exploit the fact that the negation operation in dual-rail corresponds to swapping the rails. Such dual-rail circuits do not have negative gates (internal negative gates, for example in XOR elements, are also converted into positive gates), hence they are race-free under any single transition.

One can abandon the completion detection channels, relying on timing assumptions as in standard synchronous designs; thus saving a considerable amount of area and power. This approach was followed in [17], considering the circuit in a clocked environment, where such timing assumptions were deemed quite reasonable to avoid any hazards in the combinational logic. Hence, in the clocked environment the dual-rail logic for an AND gate is simply a pair of AND-gate and OR-gate as shown in Figure 5.1(d).

The above implementation techniques help to balance switching activity at the level of dual-rail nodes, i.e. the number of switchings is made the same for each cycle of computation. Though, different gates are switching inside the dual-rail nodes. For example, compare the gate switching profiles of the structure in Figure 5.1(d) when computing two different binary sequences of values



(a) Dual-rail protocol

(b) NCL-D AND-gate

(c) NCL-X AND-gate

(d) NCL-X AND-gate with timing assumptions

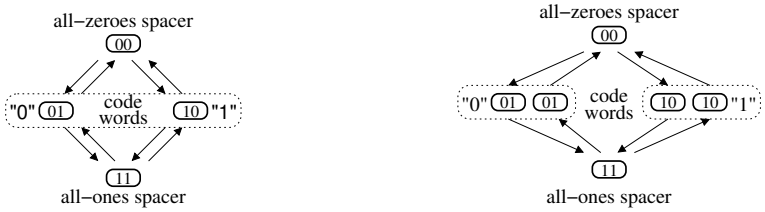
Figure 5.1: Single-spacer dual-rail

c for corresponding input sequences on a and b. The first input sequence is $a = 00$, $b = 00$, and the second one is $a = 11$, $b = 11$. The switching profile of these sequences at the level of gates is different: in the first sequence there are four firings of OR gate and in the second there are four firings of AND (note that we counted both *spacer→ code word* and *code word→ spacer* phases).

Assuming that the power consumed by one rail in a pair is the same as in the other rail, the overall power consumption is invariant to the data bits propagating through the dual-rail circuit. However, the physical realisation of the rails at the gate level is not symmetric, and experiments with these dual-rail implementations show that power source current leaks the data values. While there could be ways of balancing power consumption between individual gates in dual-rail pairs by means of modifications at the transistor level [113], adjusting loads and changing transistor sizes, etc., all such measures are costly. The standard logic library requires finding a more economic solution.

5.1.2 Dual-spacer dual-rail

In this work a new protocol with two spacer states, $\{00\}$ for all-zeroes spacer and $\{11\}$ for all-ones spacer is proposed. This *dual-spacer protocol* defines the switching as follows:



(a) Arbitrary order of spacers

(b) Alternating spacers

Figure 5.2: Dual-spacer dual-rail protocols

spacer→code word→spacer→code word, where the spacer states can be arbitrary and possibly random as shown in Figure 5.2(a).

A possible refinement for the dual-spacer protocol is the *alternating-spacer protocol* shown in Figure 5.2(b). In this protocol the all-ones spacer and all-zeroes spacer alternate in time. The advantage of this is that all bits are switched in each cycle of operation, thus opening a possibility for perfect energy balancing between cycles of operation.

As opposed to the single-spacer protocol, where in each cycle a particular rail is switched up and down (i.e. the same gate switches twice), in the alternating-spacer protocol both rails are switched from all-zeroes spacer to all-ones spacer and back. The intermediate states in this switching are code words. In the scope of the entire logic circuit, this means that for every computation cycle all gates forming the dual-rail pairs switch. This switching behaviour can be utilised for security circuits, refreshing of dynamic logic, online testing, etc. The application of the alternating-spacer protocol to security designs is studied in depth in Chapter 6.

The alternating-spacer discipline can be directly applied to a clocked NCL-X dual-rail circuit (without completion detection logic), see Figure 5.1(c). With some modification of the completion detection logic (OR-gates should be replaced by XOR-gates) the alternating-spacer protocol is also applicable to the traditional NCL-X circuits. However, this protocol cannot be applied to the NCL-D circuits. Their dual-rail gates assume that for each pair of rails the $\{11\}$ combination never occurs. In fact the use of all-ones spacer would upset the speed-independent implementation in Figure 5.1(b), because the outputs of the second layer elements would not be acknowledged during *code word→all-ones spacer* transition. The completion detection for those gates can of course be

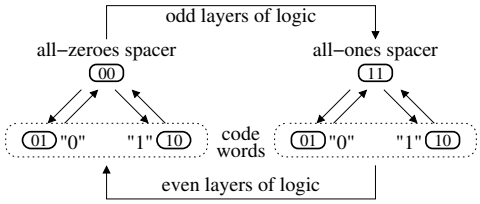


Figure 5.3: Spacer polarity after logic optimisation

ensured by using an additional three-input C-element, but this extra overhead would make this implementation technique much less elegant because of the additional acknowledgement signal channel. In the single-spacer structure, due to the principle of orthogonality (one-hot) between min-terms *a_0·b_0*, *a_0·b_1* and *a_1·b_0*, only one C-element in the rail *c_0* fires each cycle.

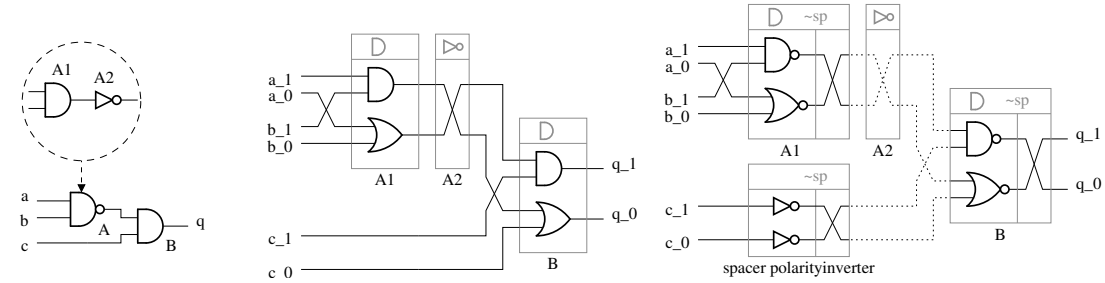
5.1.3 Negative gate optimisation

In CMOS a positive gate is usually constructed out of a negative gate and an inverter. That is why the total area overhead in a dual-rail circuit is more than twofold compared to the corresponding single-rail circuit. Use of positive gates is not only a disadvantage for the size of dual-rail circuit, but also for the length of the critical path. Our method for negative gate optimisation [17] is described in this section.

If in a dual-rail circuit an all-zeroes spacer is applied to a layer of negative gates (NAND, NOR, AND-NOR, OR-NAND), then the output will be an all-ones spacer. The opposite is also true: an all-ones spacer is converted into an all-zeroes spacer. The polarity of signals within code words can be preserved by swapping the output rails.

The spacer alternation between layers of a dual-rail circuit implemented using negative gates can be used for *negative gate optimisation* of the circuits. The optimised circuit uses either all-ones spacer or all-zeroes spacer in different layers. The spacer changes between the layers of logic as captured in Figure 5.3. The inputs of each dual-rail gate should be in the same spacer.

In order to optimise a dual-rail circuit for negative gates the following transformations should be applied. First, all gates of a positive dual-rail circuit are replaced by negative gates. Then, the output rails of those gates are swapped. In order to ensure the same spacer polarity on all inputs of a dual-rail gate *spacer polarity inverters* are used. A spacer polarity inverter is implemented as



(a) Single-rail circuit (b) Positive-logic dual-rail circuit (c) Negative-logic dual-rail circuit

Figure 5.4: Constructing negative-logic dual-rail circuit

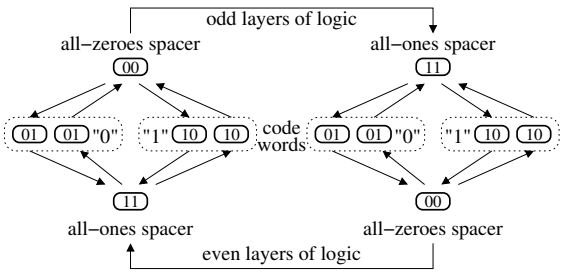


Figure 5.5: Alternating spacer after negative logic optimisation

a pair of inverters having their outputs crossed in order to preserve the polarity of signals in the output code words. They are placed at the wires that connect the layers of logic of the same parity (odd-to-odd or even-to-even).

Consider negative gate optimisation using a simple example shown in Figure 5.4(a). Firstly, a positive-logic dual-rail is built by replacing gates by their dual-rail versions as shown in Figure 5.4(b). Note, that the single-rail NAND-gate *A* is refined into an AND-gate and an inverter. The operation of inversion in dual-rail is implemented by rail swapping and does not require any logic gates. Secondly, the positive gates of the obtained dual-rail circuit are replaced by corresponding negative gates. The output rails of each dual-rail gate are swapped to preserve the polarity of signals in the output code words as shown in Figure 5.4(c). Also the spacer polarity inverter is inserted in <*c_1, c_0*> pair of wires. This is necessary to preserve the same polarity of the spacer on the input of the *B* gate. The solid lines indicate the signals which use the all-zeroes spacer, and the dotted lines indicate the wires with the all-ones spacer.

It is possible to combine negative gate optimisation with the use of the alternating-spacer

protocol, Figure 5.5. It requires a careful analysis of modules interconnect in case of hierarchical circuit design. In such circuits several instances of the same module can be found. If the inputs of the module rely on different polarity of the spacer it might be difficult to automate the optimisation process. The easiest way to avoid this difficulty is to keep the same spacer polarity, e.g. all-zeroes, on all interface signals of all modules. In some cases this rule can be abandoned in favour of speed and size optimisation, as in Section 5.5 full-adder example.

It is also possible to use conventional EDA tools for negative gate optimisation of the combinational logic. However the existing synthesis tools do not consider the dual rail nature of the circuit and can lose the balance between the rails. In contrast, our method for negative gate optimisation keeps each dual-rail gate balanced (to some extent) by either converting both its rails into negative logic or leaving both rails in positive logic.

5.1.4 Completion detection

A dual-rail circuit built out of traditional NCL-X gates requires a *completion detection* logic to compute when the data is ready on the output and the circuit is in a stable state. The result of these computations is a completion signal. This signal is used to latch the data in the circuit flip-flops.

Because of the early propagation effect it is not enough to put the completion detection logic on the flip-flop inputs only. The early propagation happens when a dual-rail gate is triggered by one of its inputs without waiting for the other inputs. If the completion signal relies on the flip-flop inputs only, then a flip-flop might latch a code word and produce a spacer (or vice versa) before all combinational logic gates switch. The propagation speed of a spacer and a code word might be different in the same path of combinational logic. If a spacer propagates faster in a given path, then it eventually might overtake a code word from the previous cycle of computation. The opposite scenario is also possible, that a code word catches up with a spacer from the previous computation cycle. Such clashing of a spacer and a code word leads to hazards in a single-spacer protocol, or, to a circuit malfunction in case of an alternating-spacer protocol.

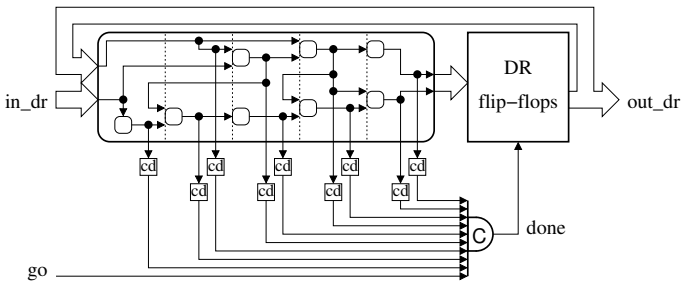
The easiest way to ensure correct computation of the completion signal is to put the completion detection in all combinational logic wires, as shown in Figure 5.6(a). The result of completion detection in individual dual-rail nets are fed into a C-element to produce the *done* signal, which

latches the data in the flip-flops. Note that the inputs of the combinational logic do not require a completion detection because it is already provided by the *go* signal.

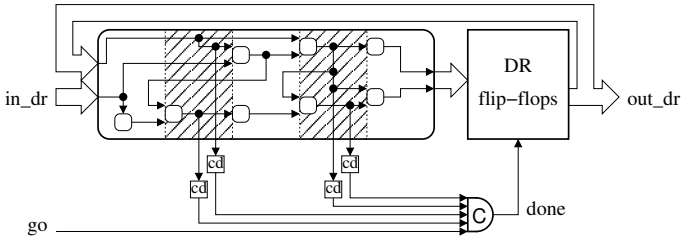
The implementation of a completion detection block (marked with the *cd* label) depends on the signalling discipline. For a traditional return-to-zeros protocol these blocks are implemented as 2-input OR-gates. If the negative gate optimisation is applied to the dual-rail logic, then the completion is detected by NAND gates in the layers with all-ones spacer and by OR-gates in the layers with all-zeros spacer. Dual-rail circuits with alternating-spacer protocol use XOR-gates for completion detection.

One of the possibilities for circuit size optimisation is to skip completion detection in some layers of combinational logic and rely on relative timing, as shown in Figure 5.6(b). This is a *layer-wise optimisation*, which is based on the following observation. A few layers just before the flip-flops can be left without completion detection. It is possible because the multi-input C-element producing the *done* signal is quite slow. If its switching from 0 to 1 (from 1 to 0) takes longer than the propagation of a code word (a spacer) through several combinational logic layers preceding the flip-flops, then completion detection is redundant in these layers. Similarly, after the layer of logic with completion detection, several layers, whose cumulative delay is less then the delay of the C-element, can be left without completion detection. The maximum number of layers between the layers equipped by completion detection depends on the complexity of dual-rail gates forming these layers and on the complexity of the C-element.

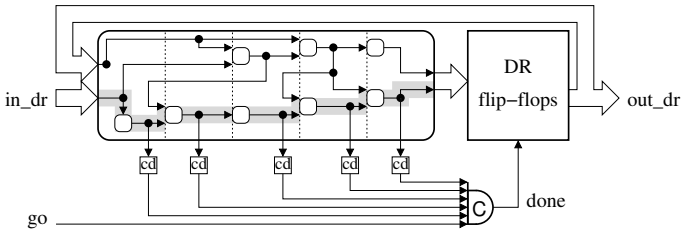
Another opportunity to decrease the size of a completion detection logic is a *path-wise optimisation*. In this optimisation the completion detection is only inserted in the critical paths of the circuit as shown in Figure 5.6(c). As the longest paths are calculated on the circuit netlist, the length of the wires is not taken into account. The actual layout of a circuit may introduce new critical paths, thus causing errors in the completion detection computation. In order to avoid such situation all paths within a safety margin (top 10-20% of the longest paths) should be equipped with the completion detection logic. Note that because of early propagation effect in the combinational logic the set of critical paths depend on the input values. This means that the critical paths must be calculated on all possible sets of input data which can computationally hard and thus requires further investigation.



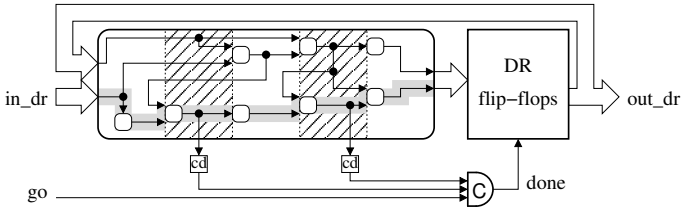
(a) No optimisation



(b) Layer-wise optimisation

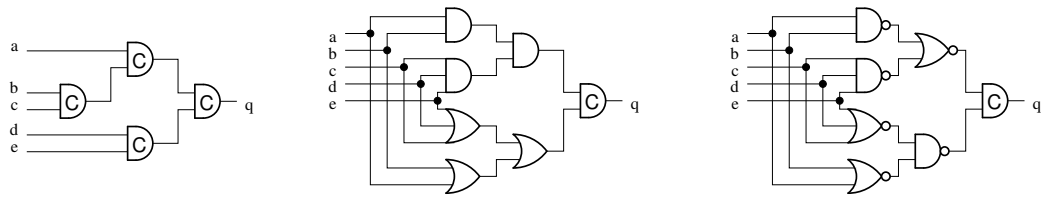


(c) Path-wise optimisation



(d) Mixed layer-wise and path-wise optimisation

Figure 5.6: Optimisation of a completion detection logic



(a) Tree of C-elements (b) Positive gate decomposition (c) Negative gate optimisation

Figure 5.7: Multi-input C-element

A combination of both optimisation techniques has shown the best result, see Figure 5.6(d). The nets which require a completion detection are computed by intersecting a set of nets obtained by the layer-wise optimisation and a set of nets calculated by the path-wise optimisation.

Careful timing analysis of the dual-rail logic is required for the proposed optimisations in order to guarantee the hazard-free switching. It can be performed by a conventional timing analysis tool, e.g. the Cadence Pearl timing analyser.

A multi-input C-element which is used to form the *done* signal can be decomposed into a tree of 2-input C-elements. It would be more convenient to built this tree using both 2-input and 3-input C-elements, especially if the number of inputs is odd. The use of 2-input C-elements only may result in an unbalanced tree having different length of paths from its inputs to the output. However, C-elements are not usually included in a library of gates and need to be built out of complex gates. While a 2-input C-element can be implemented in most technologies, a complex gate required to built a 3-input C-element is usually not available in these libraries.

For example, a 5-input C-element can be designed using four 2-input C-elements as shown in Figure 5.7(a). In such implementation of a multi-input C-element the paths from its inputs to the output may be not balanced. For example, in Figure 5.7(a) the paths from inputs *a*, *d* and *e* are two C-elements, and from inputs *b* and *c* - three C-elements. The difference in paths length can be employed by optimising the connection of the completion detection blocks to the C-element inputs: the closer a completion detection block is to the combinational logic outputs, the shorter C-element path it is connected to. The major drawback of this implementation of a multi-input C-element is its large size when the 2-input C-elements are built out of standard library gates.

The size of a multi-input C-element can be reduced by decomposing it into a 2-input C-element

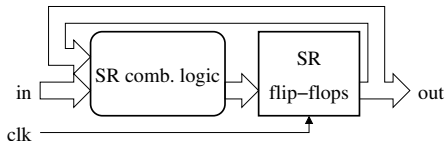
and two balanced trees of AND-gates and OR-gates, as shown in Figure 5.7(b). The trees are built using both 2-input and 3-input gates, which makes it possible to keep the length of paths form all inputs equal. In this implementation an early propagation effect can be observed. When the inputs switch from zeroes to ones, the tree of OR-gates exhibits the effect of early propagation, and vice versa, when the inputs switch from ones to zeroes the tree of AND-gates is subject to early propagation.

Because of the early propagation, the 2-input C-element can trigger the *done* signal before all the gates in the preceding trees switch. A rising edge of *done* initiates a wave of spacers on the combinational logic inputs; its falling edge allows the next wave of code-word. While propagating through the combinational logic these waves trigger the completion detection blocks. A change of the outputs of the completion detection blocks before all internal gates fire in the trees of AND-gates and OR-gates may result in hazards. These hazards are local and do not propagate through the 2-input C-element, thus the hazards do not affect the correct computation of the *done* signal. The hazards can be completely avoided if the following timing assumption is held: all gates in the trees of AND-gates and OR-gates must fire before the inputs of these trees change. This assumption is usually easy to meet.

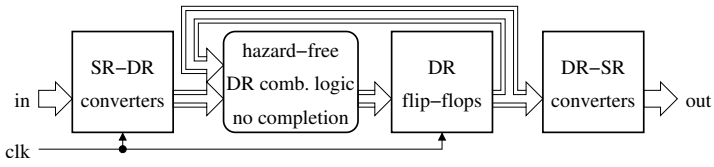
A decomposed multi-input C-element can be further optimised by applying the DeMorgan's law to its trees of AND-gates and OR-gates. The result of such an optimisation for a 5-input C-element is shown in Figure 5.7(c). It is 6 inverters smaller than the non-optimised C-element.

5.1.5 Clocked and self-timed architectures

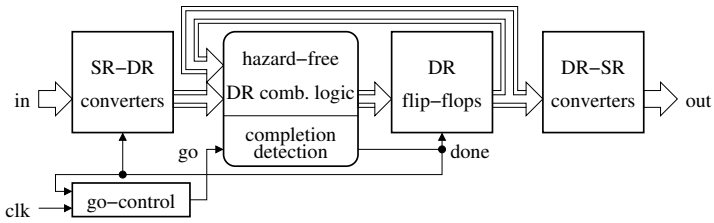
The key feature of our method is its integration with conventional design flow. The method is applied via an automated tool to a clocked single-rail netlist obtained by standard RTL synthesis tools from a behavioural specification. Such circuits have an architecture depicted in Figure 5.8(a). The result is also a netlist which can be simulated and passed to the back-end design tools. Furthermore, all DFT (Design For Testability) features incorporated at the logic synthesis stage are preserved in our approach unchanged. The resultant dual-rail circuit can be built in either of two well-known architectures (cf. [108, 118]): *clocked dual-rail* or *self-timed dual-rail*, Figure 5.8(b, c) respectively.



(a) Single-rail architecture (standard RTL design)



(b) Clocked dual-rail architecture

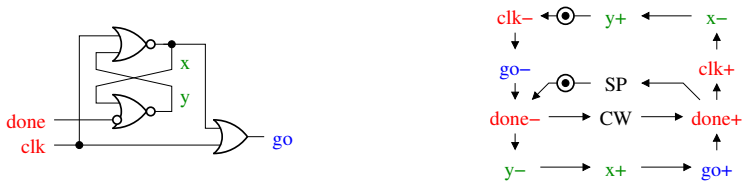


(c) Self-timed dual-rail architecture

Figure 5.8: Design architectures

Clocked dual-rail circuits do not have completion detection logic and rely on the assumption that all gates of dual-rail combinational logic switch from code-words to spacers and back to code-words by the end of the clock cycle. There are two phases in dual-rail logic which need to be complete each clock cycle. In the worst case it might require a twice as long clock period as in the single-rail prototype circuit. Usually there is no need to increase the clock cycle twice because the dual-rail logic is hazard-free and switches faster then the single-rail. The negative logic optimisation also serves the faster switching of dual-rail combinational logic. The speed of dual-rail circuit can be further increased by forcing the individual layers of combinational logic into spacer concurrently.

Self-timed dual-rail circuits do not have a clock and their registers are controlled by a completion signal formed in the completion detection logic. The completion detection logic produces



(a) Circuit (b) STG

Figure 5.9: Go-controller

done- event when the whole combinational logic is in the spacer state and the *go* input is low; similarly *done+* event is produced when the combinational logic is in code-word state and *go* is high. The high level of *done* signal forces the spacers on the outputs of dual-rail flip-flops and single-rail to dual-rail converters. The low level of *done* allows a new set of code-words into the combinational logic.

The *go-controller* helps to synchronise the completion detection of the dual-rail logic with the data input from single-rail environment. Assuming that the single-rail inputs are stable by the negative edge of the clock, the *go* signal goes low just after the *clk-*. This is acknowledged by *done-*, followed by *go+* and switching of the combinational logic into code-word state. The propagation of the code-words through the combinational logic is acknowledged by *done+* which forces the spacers on the inputs of dual-rail combinational logic. The spacers are kept until the next computational cycle. This asymmetrical behaviour of *go-controller* allows the combinational logic to stay in the spacer state most of the time, thus making a power analysis attack more difficult to fulfil.

The self-timed dual-rail circuits should exhibit better throughput compared to the clocked ones. However they suffer from a significant size overhead due to additional logic from completion detection.

The implementation of single-rail to dual-rail converters and dual-rail flip-flops is discussed in the Sections 5.2, 5.3.

(a) Circuit (b) STG

Figure 5.10: Alternating-spacer to single-spacer converter

5.2 Converters

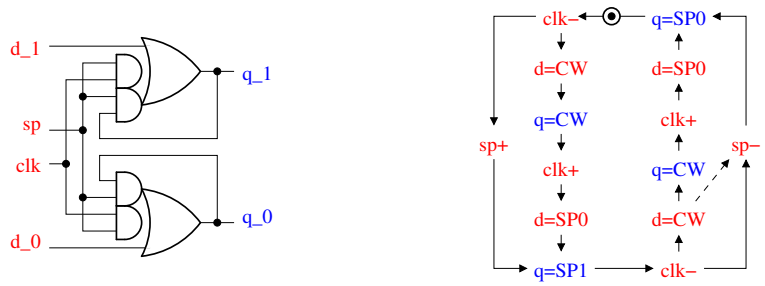
An essential part of the design flow for dual-rail data path is the interface to the single-rail environment. It is supported by means of converters between single-rail and dual-rail logic. The integration of the alternating-spacer protocol into traditional dual-rail logic also requires converters between single-spacer and alternating-spacer protocols. Several solutions for data independent converters are presented in the following subsections. They are designed by means of Petrify tool with subsequent manual optimisation.

5.2.1 Converters between single-spacer and alternating-spacer protocols

If some parts of a dual-rail circuit operate using the single-spacer and other parts the alternating-spacer protocol, then protocol converters should be used on the borders between these parts. The implementation of an *alternating-spacer to single-spacer converter* (AS-SS) and its STG are shown in Figure 5.10. It is transparent to code words and enforces an all-zeroes spacer on the <*q_1, q_0*> output if the input <*d_1, d_0*> is not a code word.

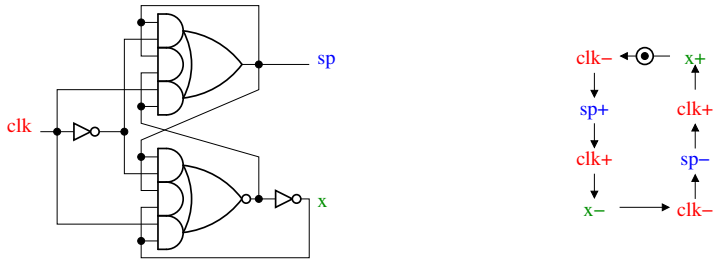
The implementation of a *single-spacer to alternating-spacer converter* (SS-AS) and its STG are depicted in Figure 5.11. The *sp* input of the converter decides which spacer to inject all-zeros or all-ones. It is generated form the *clk* signal by a *spacer-controller*.

The spacer-controller is a toggle constructed out of two latches as shown in Figure 5.12(a). Its operation is captured in Figure 5.12(b). Output *sp* changes on the negative edge of *clk* and the internal signal *x* changes on the positive edge of *clk*, thus the frequency of *sp* is half the frequency of *clk*.



(a) Circuit (b) STG

Figure 5.11: Single-spacer to alternating-spacer converter

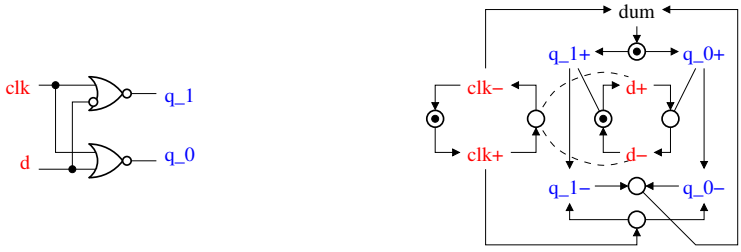


(a) Circuit (b) STG

Figure 5.12: Spacer-controller

The converter operation can be described as follows. When *sp* input is low the converter is transparent and copies the <*d_1, d_0*> input to the <*q_1, q_0*> output. If *clk* is low and *sp* is high the converter is still transparent because the high level of *sp* means that the previous spacer on the <*q_1, q_0*> output was all-zeros (due to spacer alternation). Finally, the high level of both *clk* and *sp* forces an all-ones spacer on the <*q_1, q_0*> output. In order to avoid hazards on the converter output it must hold the all-ones spacer until a code word arrives to its <*d_1, d_0*> input, only after that the converter can be transparent. This is achieved by holding *sp* high long enough for the input to change to code word. This timing assumption for the negative edge of *sp* is depicted by the dashed arc from *d=CW* to *sp-* in Figure 5.11(b). It should be taken into account when using the single-spacer to alternating-spacer converters.

The spacer is generated on the output of a single-spacer to alternating-spacer converter after the positive edge of *clk*. The decision about which spacer to inject (all-zeros or all-ones) happens



(a) Circuit (b) STG

Figure 5.13: Single-rail to single-spacer dual-rail converter

just after the negative edge of *clk*, which gives the *sp* signal about half of the clock period to reach the converter. This generous timing constraint allows the *sp* signal to be treated the same way as an ordinary data wire, not worrying about its delay and balanced distribution (as opposed to the clock tree). However, the buffering of *sp* might be necessary if it drives many converters. Buffering also helps to satisfy the timing assumption for the negative edge of *sp* signal.

5.2.2 Converters between single-rail and dual-rail logic domains

For an integration of a dual-rail circuit into a single-rail environment converters between these logic domains are required. A converter from *single-rail to single-spacer dual-rail* (SR-DR) protocol and its STG are shown in Figure 5.13. The converter forces an all-zeros spacer on its <*q_1, q_0*> output when *clk* is high. Data input *d* can be unstable while *clk* is high, however it is assumed that *d* becomes stable by the time *clk* is low, see dashed arcs is Figure 5.13(b). This timing assumption is easy to meet in both synchronous and asynchronous dual-rail architectures as the *go* signal is high at least the first half of the operation cycle.

A converter from *single-spacer dual-rail to single-rail* (DR-SR) and its STG are shown in Figure 5.14. The converter holds the previous value of the *q* output while the <*d_1, d_0*> input is in all-zeros spacer. When a code word arrives at the <*d_1, d_0*> input the output is changed to the negation of *q_0*. The timing assumption shown by dashed arcs in Figure 5.14(b) is that the converter switches into a new value before the next spacer arrives at its input. This is easy to meet as the converter's delay is only a pair of two-input negative gates.

The converters between single-rail and dual-rail domains are adapted to the alternating-spacer

(a) Circuit (b) STG

Figure 5.14: Single-spacer dual-rail to single-rail converter

(a) Single-rail to alternating-spacer dual-rail (b) Alternating-spacer dual-rail to single-rail

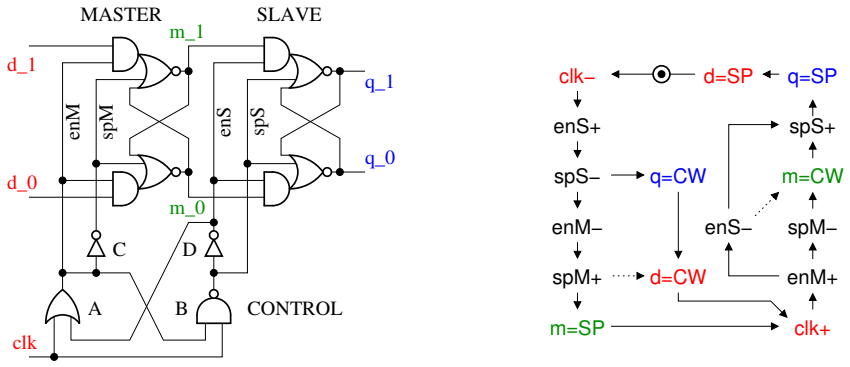
Figure 5.15: Converters between single-rail and alternating-spacer dual-rail

protocol by combining them with converters between single-spacer and alternating-spacer protocols as shown in Figure 5.15.

The timing assumption for the negative edge of *sp* input (see Section 5.2.1) is easy to meet in the single-rail to alternating-spacer dual-rail converter. Both, single-rail to dual-rail converter and the spacer-controller produce code word and *sp*- respectively by the negative edge of *clk*. The delay of the code word on the converter output is equal to the delay of one NOR-gate, while the *sp* signal is delayed by a complex AND-OR-gate and a buffering tree for distributing *sp*.

5.3 Dual-rail flip-flops

Synchronous flip-flops are built to be power efficient, so if they switch to the same value (data input remains the same within several clocks) then nothing changes at the output. The absence of the output transition saves power, but at the same time it makes the power consumption data dependent. In order to avoid this, the dual-rail flip-flops operate in the return-to-spacer protocol.



(a) Circuit (b) STG

Figure 5.16: Single-spacer dual-rail flip-flop

5.3.1 Single-spacer dual-rail flip-flop

A single-spacer dual-rail flip-flop and its STG are shown in Figure 5.16. The flip-flop consists of two latches (MASTER and SLAVE) and their CONTROL unit. Both MASTER and SLAVE have their respective enable (*enM* and *enS*) and reset-to-spacer (*spM* and *spS*) inputs. These signals are used by the CONTROL to manipulate the data transfer between MASTER and SLAVE.

The flip-flop operates in two phases. In the first phase, denoted by a negative edge of *clk*, first a code word is copied from MASTER into SLAVE, and then the MASTER is reset to spacer. In the second phase initiated by a positive edge of *clk*, first the code word from the combinational logic is stored into the MASTER and then the SLAVE is reset to spacer.

The first advantage of this implementation is its size. The flip-flop uses of a single cross-coupled latch in each stage for a couple of input data signals.

The other advantage is the data independent power consumption of the flip-flop. This is achieved by the symmetry of the flip-flop rails. The internal wires are assumed to be short and the difference in their electrical characteristics is negligible.

However, there are two timing assumptions in this design, which are depicted as dotted arcs in Figure 5.16(b):

1. The spacer is enforced in MASTER before the code word propagates through the combinational logic (arc $spM+ \rightarrow d = CW$).

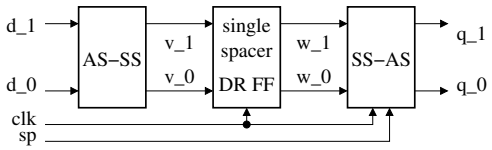


Figure 5.17: Alternating-spacer dual-rail flip-flop

2. The SLAVE is disabled before the code word propagates through MASTER (arc $enS- \rightarrow m = CW$).

The first timing assumption results in a requirement that the cumulative delay of combinational logic and SLAVE latch is greater then the delay of OR-gate A and inverter C. The second timing assumption is satisfied if the delay of MASTER latch and inverter C is greater than the delay of NAND-gate B. Both timing assumptions are easy to meet.

5.3.2 Alternating-spacer dual-rail flip-flop

In order to support the alternating-spacer protocol the single-spacer dual-rail flip-flop is surrounded by converters between single-spacer and alternating-spacer protocols as shown in Figure 5.17.

This implementation uses *sp* signal to decide which spacer to inject in the positive phase of *clk*. Both the dual-rail flip-flop and the spacer-controller produce code word and *sp*- respectively by the negative edge of *clk*. The delay of the code word on the flip-flop output is equal to the delay of the NAND-gate B and the SLAVE latch (see Figure 5.16), while the *sp* signal is delayed by a complex AND-OR-gate and a buffering tree for *sp* distribution (see Figure 5.12). For large circuits the buffering tree provides the necessary delay of *sp*, however if the buffering of *sp* is not required due to the small size of the circuit, then additional delay elements may be needed for the *sp* signal.

The power consumption of the resultant alternating-spacer dual-rail flip-flop is data independent due to the symmetry of logic in its rails. The internal wires must be short to maintain balance in the electric characteristics of the rails.

Another implementation of an alternating-spacer dual-rail flip-flop is shown in Figure 5.18. This is a speed-independent circuit. It consists of three latches LATCH_1, LATCH_2 and LATCH_3 connected in series. Its operation (at a high-level) is captured above each latch, denoting the events

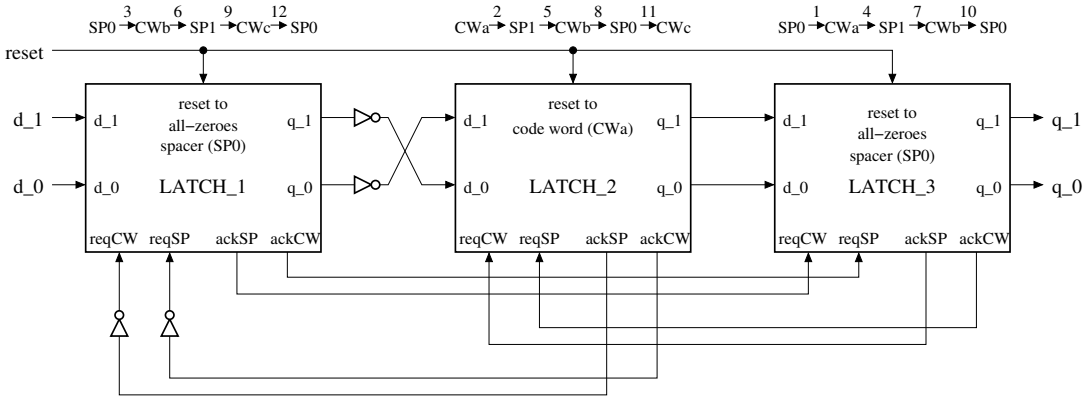


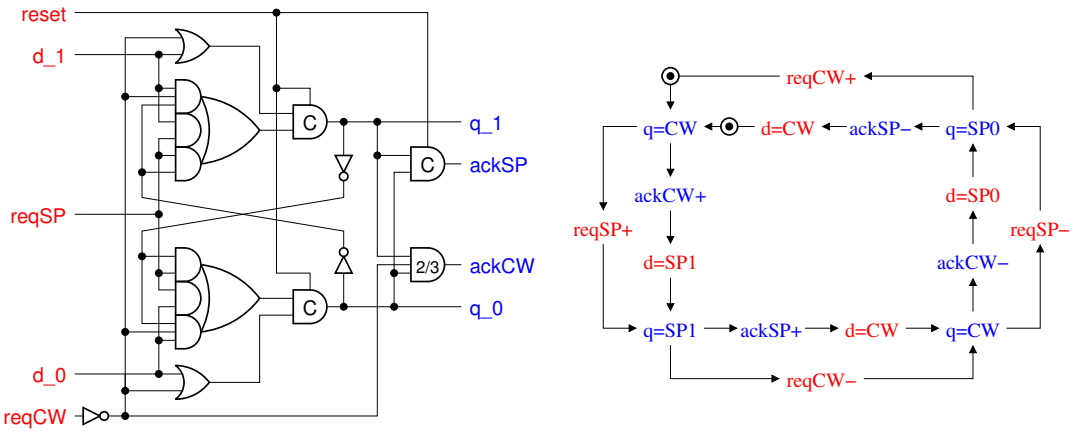
Figure 5.18: Three-stage alternating-spacer dual-rail flip-flop

on its output. The order of the events is shown by numbers above the arrows. Initially LATCH_2 is reset to a code word *CWa*, LATCH_1 and LATCH_3 are reset to all-zeroes spacer. When the reset is released, first, the code word *CWa* is copied from LATCH_2 to LATCH_3. It is followed by coping the spacer from LATCH_1 to LATCH_2. Then the code word *CWb* from the combinational logic output is stored in LATCH_1, which allows all-ones spacer on the output of LATCH_3, and so on. A pair of inverters with their outputs crossed between LATCH_1 and LATCH_2 provides the alternation of the spacer, code words are unchanged. The inverters between the acknowledgements of LATCH_2 and requests of LATCH_1 serve the same purpose of spacer alternation.

An individual dual-rail latch which is used in the above alternating-spacer dual-rail flip-flop is shown in Figure 5.19(a). Its operation is captured by the STG in Figure 5.19(b). It has two request inputs *reqCW* and *reqSP*. A transition on *reqCW* means that the code word on the input <*d_1, d_0*> is ready to be latched. The latching is acknowledged by a transition on *ackCW* output (the same polarity as *reqCW*). A positive transition of *reqSP* input requests the storage of all-ones spacer and its negative transition denotes the storage of all-zeroes spacer. The latching of a spacer is acknowledged by a transition on *ackSP* output (the same polarity as *reqSP*).

5.3.3 Complex flip-flops and transparent latches

Dual-rail flip-flops (either single-spacer or alternating-spacer) and multiplexers compose complex memory elements, e.g. transparent latches, flip-flops with enable input, flip-flops with SCAN inputs, etc. Building a complex dual-rail block out of existing elements is advantageous for design



(a) Circuit (b) STG

Figure 5.19: One stage of an alternating-spacer dual-rail flip-flop

productivity especially on the stage of prototyping, however the resultant circuits are not optimal in terms of size and speed. Subsequently, the size and latency reduction can be achieved by joint optimisation of the comprised elements.

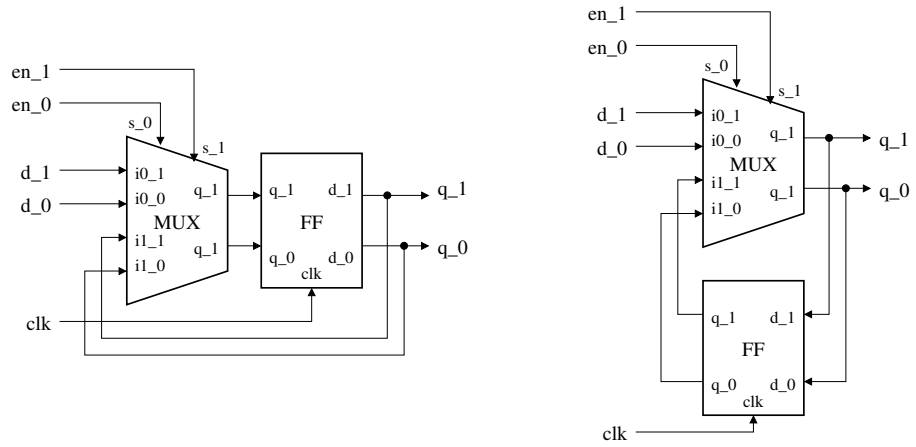
Figure 5.20(a) shows a dual-rail implementation of a D-enabled flip-flop with active low enable. The logical 0 on input <*en_1, en_0*> opens the flip-flop for the new data, while the logical 1 forces the flip-flop to keep its value by means of the feedback from the output <*q_1, q_0*> to the input <*i1_1, i1_0*> of the multiplexer.

Similar, the transparent latch whose dual-rail implementation is shown in Figure 5.20(b) uses its input <*en_1, en_0*> to decide which value to output. If <*en_1, en_0*> is set to logical 1 then the latch outputs the value previously stored in the flip-flop, otherwise the latch is transparent.

A situation when some inputs of a combinational logic gate are in all-zeros spacer and the other inputs of the same gate are in all-ones spacer must be avoided. Different spacers on the inputs of the same gate may produce a code word on the gate output, which violates the switching protocol. The spacer polarity is synchronised on the circuit inputs and flip-flop outputs in the reset phase.

5.3.4 Reset of dual-rail flip-flops

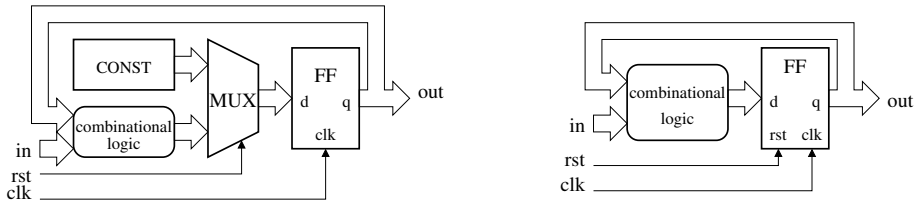
There are two types of reset in traditional single-rail circuits: *synchronous* and *asynchronous*.



(a) D-enabled flip-flop

(b) Transparent latch

Figure 5.20: Complex flip-flop and transparent latch



(a) Synchronous reset

(b) Asynchronous reset

Figure 5.21: Reset types

The *synchronous reset* is similar to an ordinary data input. It controls a multiplexer which selects a value to store in a flip-flop: a predefined constant or an output of combinational logic. An active low synchronous reset schematic is depicted in Figure 5.21(a). When the reset is low the flip-flop is set to a constant value CONST, otherwise the output of combinational logic is supplied to the flip-flop input.

The *asynchronous reset* uses a dedicated flip-flop input to set its value to 1 or 0 (depending on the construction of the flip-flop) as shown in Figure 5.21(b).

When an RTL circuit is converted into dual-rail its synchronous reset, being an ordinary data input, is converted into a pair of rails. This dual-rail reset goes through a return-to-spacer protocol similar to all other data wires. An asynchronous reset is preserved unchanged in a dual-rail circuit.

It behaves exactly the same way as in the original single-rail circuit.

If a single-rail circuit has the same input *rst* used both as synchronous and asynchronous reset, then such a circuit must be modified before the conversion. For this, firstly, a new input signal *rst_sync* is added to the circuit. This signal replaces *rst* in all combinational logic gates, i.e. in all the places where synchronous reset is required. The original *rst* signal is still used for the explicit initialisation of flip-flops via dedicated set/reset inputs, i.e. as asynchronous reset. After this transformation the circuit is converted into dual-rail in a usual manner. The purely synchronous reset *rst_sync* is converted into a pair of rails <*rst_sync_1, rst_sync_0*>. The original *rst* input, which became a purely asynchronous reset, is preserved single-rail. Finally, the dual-rail synchronous reset <*rst_sync_1, rst_sync_0*> is derived from *rst* using a single-rail to dual-rail converter.

5.4 VeriMap design kit

The conversion from single-rail into dual-rail circuit has been implemented in a VeriMap design kit. It successfully interfaces to the Cadence CAD tools. It takes as input a single-rail circuit netlist, created by Cadence Ambit or another logic synthesis tool, and converts it into a dual-rail netlist. The language for input and output netlists is a conventional structural Verilog whose syntax is described in Appendix B. The resulting netlist can be processed by Cadence or other EDA tools supporting the Verilog language.

The structure of the VeriMap design kit is displayed in Figure 5.22. It consists of a VeriMap software tool, a library of *gate prototypes*, a library of *transformation rules* and a library of *gate attributes.*

The main function of the tool is conversion of a single-rail RTL netlist into a dual-rail netlist for either of the two architectures: self-timed or clocked, Figure 5.8(b, c) respectively. It is done in four stages. First, a single-rail circuit is converted into positive-logic dual-rail. Second, the positive dual-rail gates are replaced by negative dual-rail gates and the spacer polarity inverters are inserted. Then, the completion signal is generated (asynchronous design only). Finally, a wrapper module connecting the dual-rail circuit to the single-rail environment is added (optional).

Apart from generating netlists, Verimap tool reports statistics for the original and resultant

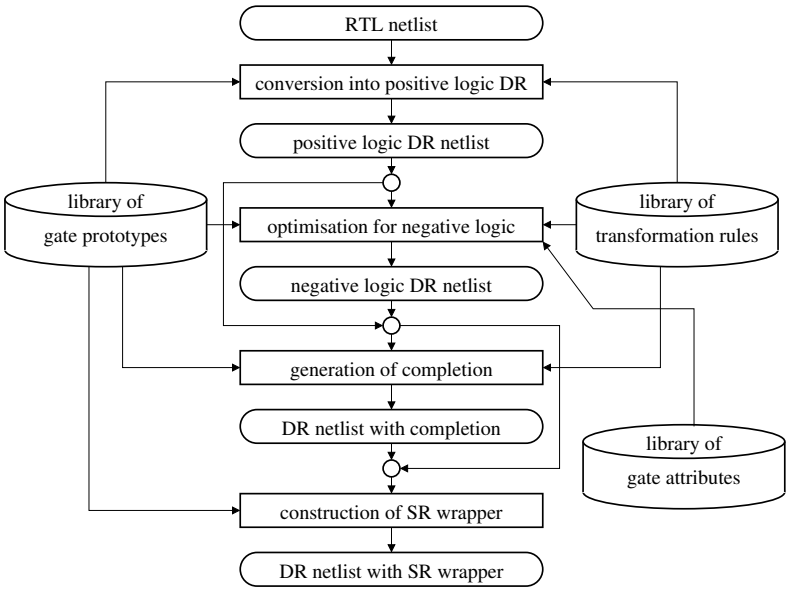


Figure 5.22: Verimap design kit

circuits: the estimated area of combinational logic and flip-flops, the number of negative gates and transistors, and the number of wires. The tool also generates a behavioural Verilog functions assisting the power analysis of the original and resultant circuits. Being included into simulation testbench these functions count the number of switching events in each wire of the circuits.

The libraries define the technology parameters for which the circuit is synthesised. The VeriMap toolkit includes these libraries for the AMS 0.35μ CMOS technology. After some modification they can be also used for other design technologies. Both libraries are described in the following sections.

5.4.1 Gate prototypes

The library of gate prototypes is a set of Verilog files whose format is described in Appendix B. This library contains three types of gates:

- single-rail gates;
- manually designed converters and controllers;
- manually designed dual-rail gates.

The definitions of all single-rail gates which are instantiated in the circuit under conversion have to be in the library. These definitions are used by VeriMap to determine which connections of the gate are inputs and which are outputs. The implementation of these gates is not important and can be skipped.

The converters between single-rail and dual-rail domains, go-controller and spacer-controller are also defined in this library. These converters and controllers cannot be produced automatically by the tool and must be created manually for each design technology.

The library can also contain implementations of some dual-rail gates. The predefined dual-rail gates are used when the automatic conversion of a single-rail gate into dual-rail is impossible or gives a poor result. Typical examples for such gates are multiplexers, flip-flops and latches.

5.4.2 Rules for gate transformation

The library of rules for gate transformation defines how to convert a single-rail gate into a dual-rail one. If a predefined dual-rail implementation of a gate is found in the library of gate prototypes, then the tool uses it. Otherwise a dual-rail implementation is built automatically using these rules. Each line in this library defines one rule. A rule consist of the following space-separated fields:

- *single_rail* - the name of a single-rail gate to which the rule applies;
- *dual_rail* - the name of a corresponding dual-rail gate (usually it is the same as *single_rail*);
- *direct_positive* - the name of a positive gate which implements the function of the single-rail gate;
- *complementary_positive* - the name of a positive gate which implements the function complementary to the single-rail gate;
- *direct_negative* - the name of a negative gate which implements the function of the single-rail gate;
- *complementary_negative* - the name of a negative gate which implements the function complementary to the single-rail gate;

· *interface_polarity* - a sequence of characters defining in a position code which ports of the single-rail gate are inputs (encoded by small characters) and which are outputs (encoded by capital letters). The characters also encode how the gate inputs and outputs are converted:

p - a direct (positive) input;

n - an inverted (negative) input, its rails should be crossed;

P - a direct (positive) output, its rails should be crossed when the gate is converted into a negative dual-rail logic;

N - an inverted (negative) output, its rails should be crossed when the gate is converted into a positive dual-rail logic;

d - a data input, which should be converted into dual-rail (by default);

s - a control input, which is not converted and stays single-rail (e.g., clock and reset inputs);

D - a data output, which should be converted into dual-rail (by default);

S - a control output, which is not converted and stays single-rail (e.g., output of a controller generating local clock or reset signals);

r - a dual-rail implementation of the gate requires an additional reset signal;

c - a dual-rail implementation of the gate requires an additional clock signal.

· *flags* - is a string of four characters answering the following sequence of questions (*y* for yes, *n* for no):

1. Is this rule for a flip-flop or a latch?
2. Does a dual-rail gate obtained by this rule have early propagation?
3. Should the positive gates listed in *direct_positive* and *complementary_positive* be taken into account when optimising for size and speed?
4. Should the negative gates listed in *direct_negative* and *complementary_negative* be taken into account when optimising for size and speed?

In order to simplify the library of transformation rules, one rule can describe several gates. A rule is applied to a gate if the *single_rail* field matches the beginning of the gate name, called *prefix*. The rest of the gate name, called *suffix*, is added to the corresponding *direct_positive*, *complementary_positive*, *direct_negative* and *complementary_positive* fields. If there are more than one rule matching the beginning of a gate name, then the rule with the longest *single_rail* field is chosen. The association of a single rule to a set of gates helps to deal with different gate drives, because the drive is usually encoded by the last characters of the gate name.

If the automatic generation of a positive-logic dual-rail gate for a given single-rail gate is not possible (or a manual implementation is preferred for some reason), then underscore symbols '_' should be put in place of the *direct_positive* and *complementary_positive* fields. Similarly, the automatic generation of a negative-logic dual-rail gate can be banned by placing '_' in the *direct_negative* and *complementary_negative* fields.

Consider the gate transformation rules on the following simple example:

```
NA2     NA2     AND2  OR2  NA2  NO2  ppN   nyyy  // 2-input NAND
LOGIC0  ground  _     _    _    _    pPcr  ynyy  // Tie-down to logic low-level
```

The first rule describes a 2-input NAND-gate `NA2`. The name of its dual-rail implementation is the same as the original gate name. Its positive-logic dual-rail implementation consists of a 2-input AND-gate and a 2-input OR-gate. Its negative-logic dual-rail implementation combines a 2-input NAND-gate and a 2-input NOR-gate. The list of the gate connections starts with two inputs and finishes with an output, which requires a spacer inverter in a positive-logic dual-rail implementation. The NA2 is a combinational logic gate (not a flip-flop), its dual-rail implementation is prone to early propagation, both the positive and negative dual-rail implementations can be used for optimisation algorithm.

The second rule describes a 'tie-down to 0' gate `LOGIC0`. According to the rule its dual-rail implementation should be renamed to `ground`. There is no rule for for automatic conversion of this gate into a dual-rail (manual implementation is required). The manually designed dual-rail gates require additional inputs: a clock and a reset. The `LOGIC0` gate is similar to a flip-flop holding a constant 0. It does not exhibit the early propagation. Both, positive-logic and negative-logic dual-rail implementations are available for this gate (manually designed and placed in the library of gate prototypes).

5.4.3 Gate attributes

The negative gate optimisation usually improves the size and speed of a dual-rail circuit. However, for a negative gate with high output drive it is not true, because such a gate consists of a positive gate and a strong inverter attached to the output. A *library of gate attributes* is useful in such a case. For each single-rail gate it helps to choose the optimal dual-rail gate between its negative-logic and positive-logic implementations. If this library is missing, then the negative-logic dual-rail implementation is preferred.

The library consists of two sections. The first section lists the names of attributes, which are associated with each library gate. The second section assigns the values of these attributes to the gates. The file format is explained on the following example:

```
[
  delay,  // gate delay in "simple negative gate" number
  area    // area of the module in square microns
]
AND2      2,73  // 2-input AND
OR2       2,73  // 2-input OR
NA2       1,55  // 2-input NAND
NO2       1,55  // 2-input NOR
```

In this example, the comma-separated list in the square brackets consists of two attribute names: `delay` and `area`. The former attribute defines the maximum delay of a gate in some abstract units (in our case a unit equals to a simple negative gate delay). The latter attribute defines a gate area in square microns. The second section of the example lists 4 gates with their `delay` and `area` attributes. These are used by VeriMap to optimise the latency and size of the resultant dual-rail circuit. If there are two possible dual-rail implementations for a given single-rail gate, a positive-logic and a negative-logic, the one with smaller `delay` and/or cumulative `area` attributes is chosen. Taking into account the transformation rules for the `NA2` gate, its negative-logic dual-rail implementation is beneficial in terms of speed and size. Indeed, the negative-logic implementation, which consists of a `NA2` and `NO2`, has a 1 unit delay and $110\mu m^2$area. The positive-logic implementation, which consists of a `AND2` and `OR2`, has a 2 unit delay and $146\mu m^2$area.

A user is free to extend this library by adding new gate attributes. For example, a power consumption attribute can be used by VeriMap to optimise the dual-rail circuit for low power.

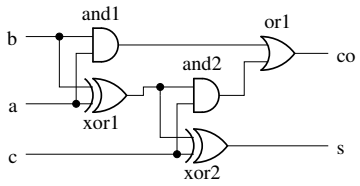


Figure 5.23: Single-rail full-adder

Note that the optimisation preferences are defined by the order of the attributes in the list.

5.5 Full-adder example

The operation of the VeriMap toolkit is illustrated on a full-adder example shown in Figure 5.23. Its AMS-0.35μm^2technology netlist is in `full_adder.v` file. The libraries of gate prototypes and transformation rules are predefined in VeriMap design kit for AMS-0.35μm^2 technology. The libraries are in the `ams.v` and `ams.rls` files respectively. The names of the library files should be passed as the `--rules` and `--include` command-line options to the VeriMap tool (see Appendix B for the full list of options).

The first step is building a positive-logic dual-rail circuit. It is obtained by the following command:

```
$ verimap --rules ams.rls --include ams.v \
--optimisation-level0 --transformation-level3 \
--output full_adder_dr.v full_adder.v
```

Its `--optimisation-level` and `--transformation-level` options mean that no optimisation should be applied and no completion detection logic is required. The tool just duplicates all the wires and replaces all gates by corresponding positive-logic dual-rail elements as shown in Figure 5.24. The number in parentheses, next to the gate name, is the layer count starting from the circuit inputs.

The next step is the negative-logic optimisation. The optimised circuit is built by the following command:

```
$ verimap --rules ams.rls --include ams.v,ams_dr-cl.v \
--optimisation-level1 --transformation-level3 --spacer-1 c,co \
--output full_adder_dr.v full_adder.v
```

The `--optimisation-level` option is modified to request an optimisation for negative gates.

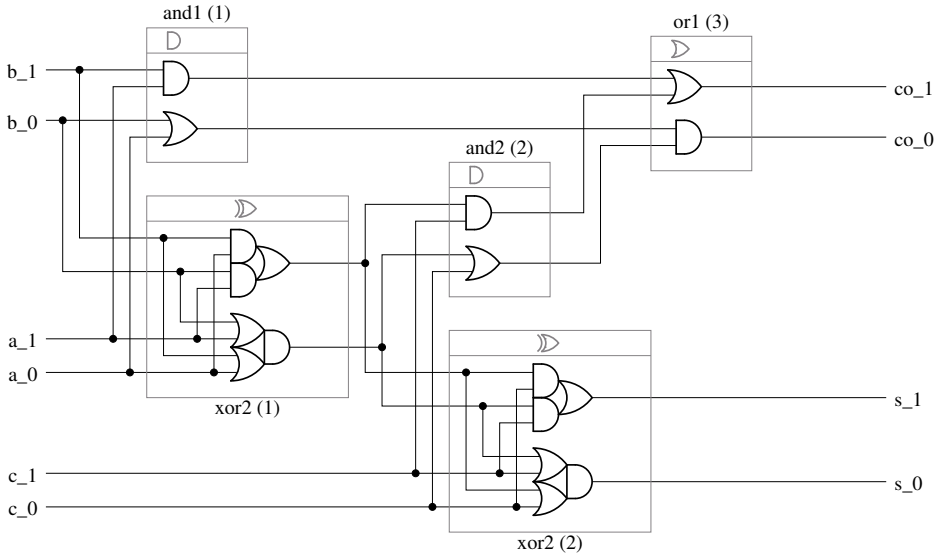


Figure 5.24: Positive-logic dual-rail implementation of a full-adder

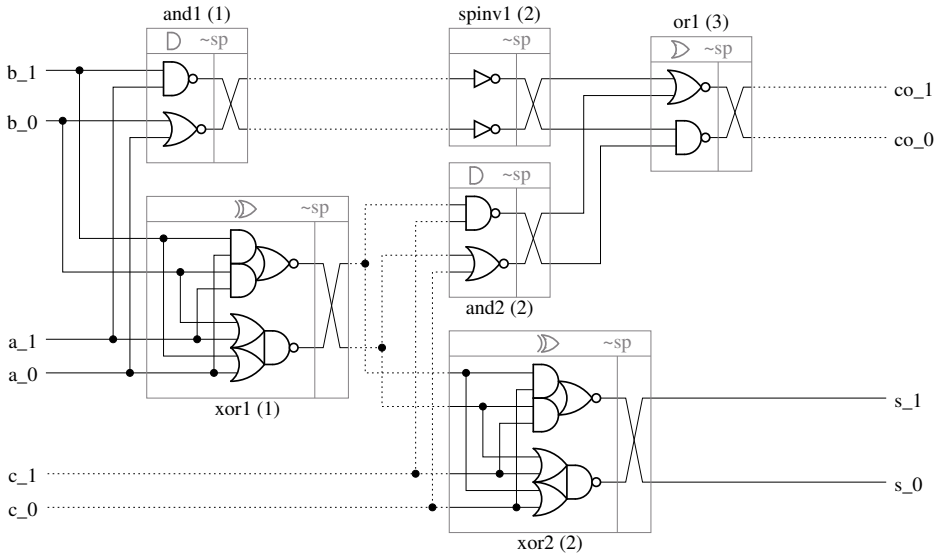


Figure 5.25: Negative-logic dual-rail implementation of a full-adder

Note the `--spacer-1` option which forces the *c* input and *co* output to use the all-ones spacer (by default the interface signals use all-zeros spacer). Figure 5.25 shows the resultant dual-rail circuit. There is one spacer inverter inserted in the wires between *and1* and *or1* which connect the layers of the same parity. Without this spacer polarity converter the *or1* gate would have different spacers on its inputs, which would introduce hazards on the <*co_1, co_0*> output.

The use of the all-ones spacer on the *c* and *co* interfaces allows to shorten the carry calculation

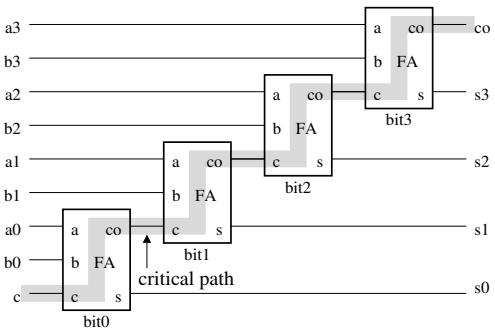


Figure 5.26: 4-bit ripple-carry adder

path by two spacer inverters. This results in a significant size and speed improvement in an N-bit ripple-carry adder where the carry calculation is on the critical path. Such an adder is composed of N 2-bit full-adders with the output *co* of the previous bit connected to the input *c* of the next bit full-adder. For example, a 4-bit ripple-carry adder with the critical path highlighted is shown in Figure 5.26.

The completion detection logic can be added to the dual-rail circuit by following command:

```
$ verimap --rules ams.rls --include ams.v,ams_c.v \
--optimisation-level1 --transformation-level4 --spacer-1 c,co \
--output full_adder_dr.v full_adder.v
```

There are two changes in this command: the `ams_c.v` library containing a C-element definition is included and the `--transformation-level` option is modified. The circuit obtained by this command is shown in Figure 5.27. Note there is no completion detection on the output of *xor1* and *spinv1*. These gates do not exhibit early propagation, which is reflected in the library of transformation rules.

The completion detection shown in Figure 5.27 is suitable for a single-spacer protocol only. OR-gates check the completion on the rails with all-zeros spacer and NAND-gates on the rails with all-ones spacer. In order to support an alternating-spacer protocol the NAND-gates and OR-gates should be replaced by XNOR-gates.

Here is some statistics for different full-adder implementations without completion detection logic. The dual-rail full-adder optimised for negative logic uses 8 inverters less than the non-optimised one (10 inverters are removed and 2 inverters are inserted for spacer polarity inversion). The critical path (carry flag calculation) has been shorten by 3 inverters compared to the original

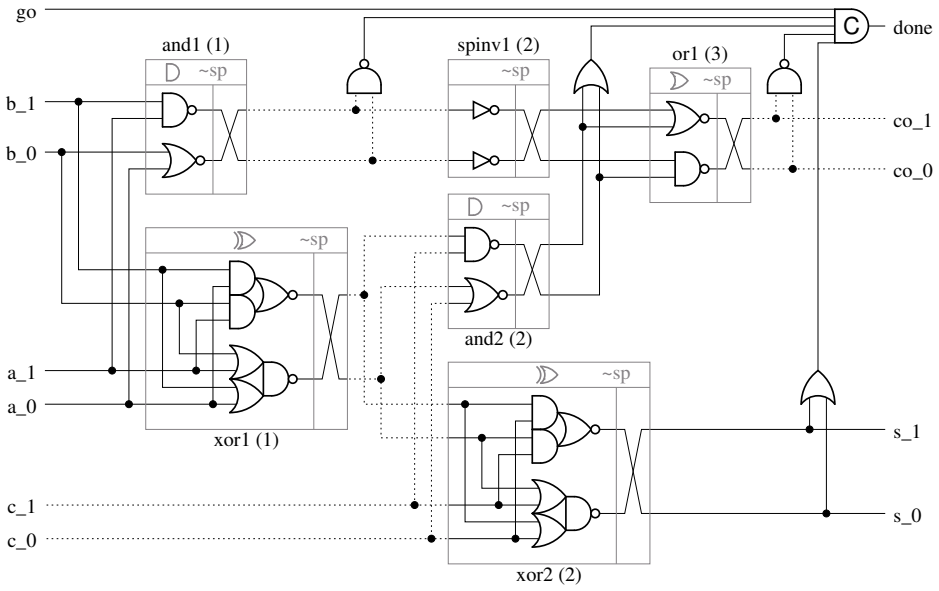


Figure 5.27: Dual-rail full-adder with completion detection logic

single-rail circuit. CMOS implementation of the single-rail full-adder uses 38 transistors, non-optimised dual-rail - 76 transistors. The optimised dual-rail circuit (without completion detection logic) consists of 60 transistors, which is only 1.58 times more than in the single-rail circuit.

5.6 Summary

In this chapter a method for automated synthesis of the data path components by direct mapping is presented. Each data path component is first implemented using standard RTL synthesis tools, e.g Synopsys. The obtained RTL circuit is mapped into a hazard-free logic using a dual-rail encoding with a return-to-spacer signalling and a completion detection logic is added in order to indicate when the computation is finished, similar to the NCL-X approach [62]. Methods for negative logic and completion detection logic optimisations are also proposed.

A new alternating-spacer protocol is proposed which has two spacers interchanging in time within the dual-rail logic framework. It is very cheap yet effective way to make all gates in a dual-rail circuit switch per computation cycle. The potential applications of the alternating-spacer protocol are energy balancing in security circuits, online testing, refreshing of dynamic logic.

The idea of using two spacers is deemed particularly efficient for dual-rail logic, where the

Hamming distance between each spacer and a valid combination is the same. While it can still be used without too much overhead in optimally balanced k-of-n codes (e.g. 3-of-6) it would be much less efficient in other popular codes such as 1-of-4 [2].

A set of converters for integrating a dual-rail circuit into a single-rail environment are designed using Petrify with subsequent manual optimisation. Two implementations of a secure dual-rail flip-flop are presented. The first solution is small (built of two latches and a simple controller) and relies on local timing assumptions. The second solution is speed-independent and large (built of three latches with dedicated controllers).

The VeriMap design kit presented in this chapter automatically converts a single-rail circuit into a dual-rail circuit. It supports two design architectures: clocked and self-timed. The design kit successfully interface to standard RTL design flow used by most ASIC designers. The tool takes as input a single-rail circuit netlist, created by Cadence Ambit, and converts it into a dual-rail Verilog netlist. The operation of the tool is illustrated on a 4-bit adder benchmark.

Chapter 6

Synthesis of security circuits

Secure applications such as smart cards require measures to resist timing and power attacks. Dual-rail encoding provides a method to enhance the security properties of a system making power analysis more difficult. As an example, in the design described in [94] the processor can execute special secure instructions. These instructions are implemented as dual-rail circuits, whose switching activity is meant to be independent from data. Whilst alternatives exist at the software level to balance power, the need for hardware solutions is also mandatory. Special types of CMOS logic elements have been proposed in [112], but this low-level approach requires changing gate libraries and hence is costly for a standard cell or FPGA user.

In recent work [113] a methodology for designing secure circuits was proposed. The main advantage of the method is that it is integrated in a standard design flow. However, this approach suffers from the following drawbacks. First, it is difficult to build a dual-rail gate which consumes the same power regardless of data processed. Even if such a secure gate is built for one set of fabrication parameters (output load, supply voltage, environment temperature) it still can expose unbalanced power consumption in other conditions. Second, the use of positive logic and separation of complementary rails imply recalculation of inverted inputs of each gate to the input of the circuit. This may cause a significant (up to four times) increase in the circuit size, for instance a tree of XOR gates. Use of positive logic may also increase the length of the critical path because additional inverters are inserted. Finally, the method is only applicable to netlists built of a limited subset of the library gates.

The clock signal is typically used as a reference in power analysis techniques. System "desynchronisation" as in [76, 125] can help hide the clock signal. To mask the operation of a block of logic is a much more complex task which could demand very expensive changes to the entire design flow. A cheaper desynchronisation method to rebuild individual blocks within the same synchronous infrastructure so, that their power signatures become independent of the mode of operation and of the data processed. This method is used in [62], where synchronous pipelines are transformed into asynchronous circuits using dual-rail coding. Dual-rail encoding was also successfully used in [89] to built a secure Amulet core for smartcard applications.

These desynchronisation methods represent a combination of two aspects of security: hiding the reference signal and hiding the data being processed. The major leakage of information about the processed data is due to the data-dependent power signature. The correlation between data and power signature can be minimised by *balancing* and *randomising* the data encoding w.r.t. power signature. The balancing techniques aim at keeping the power signature invariant to processed data, thus the power consumption should be at least the same as that in the worst (in terms of power) case of computation. The randomisation techniques mask the data-dependent power signature by inserting random noise. The noise can be introduced in many different ways, ranging from randomised masking of the secret key, to clock noise insertion, to repetition of the algorithm on randomly generated secret keys, to embedding the secret key in a much larger random key [23, 47]. The level of noise should be big enough to mask the level of power signature which can be correlated to the processed data. Both balancing and randomisation techniques burn extra power to increase the security, however the power consumption is a very secondary concern in security circuits.

This chapter focuses on the balancing of data encoding only, as randomisation techniques can be applied independently and possibly in conjunction with our method. The rest of the chapter is organised as follows. Metrics for circuit analysis in terms of security are introduced in Section 6.1. This section also shows the influence of single-spacer and alternating-spacer protocols on the circuit security. The AES benchmark results and potential improvements follow and finally the conclusions are presented.

This chapter is based on results presented in [105].

6.1 Timing and power attacks

A side channel attack is based on information leakage from the physical implementation of a cryptosystem, rather than theoretical weaknesses in the algorithms. For example, timing information, power consumption, electromagnetic emanations or even sound can provide information which can be exploited to break a system. Leakage of information through timing and power is typically used in the attacks on circuits.

A *timing attack* is a form of side channel attack where the attacker tries to break a cryptosystem by analysing the time taken to execute a cryptographic algorithm on different input data. Timing attacks on synchronous circuits typically use the clock signal as a reference [60]. In asynchronous world timing attacks can be resisted, for example, by inserting random delays in request and acknowledgement lines of the circuit and by restricting the early propagation inside its combinational logic.

A *power attacks* is based on analysis of variations in a circuit power consumption when processing different data [71]. Two main approaches to the power analysis are *Simple Power Analysis* (SPA)[70] and *Differential Power Analysis* (DPA) [61]. In SPA attacks, an attacker directly observes a system's power consumption and tries find a correspondence between the amount of power consumed and the secret data processed. DPA attacks use statistical analysis and error correction techniques to extract information correlated to secret data. Implementation of a DPA attack involves two phases: data collection and data analysis. Data collection for DPA may be performed by sampling a device's power consumption during cryptographic operations as a function of time. The more traces of circuit power consumption are collected the better chances that statistical data analysis reveals the secret data.

The collection of power traces for a successful attack, however, is not easy. A device processing secret data can limit the number of computations, or allow a new set of data to be computed only after some delay. Such measures make the collection of sufficient amount of data difficult. Also, there is usually no direct access to the device, which makes the direct sampling of power signature impossible. The secret information can still leak through electromagnetic emission or temperature of the circuit. If the information leakage is reduced to the noise level, then filtration techniques should be applied. Filtering usually implies integration over time. In the following

sections energy consumption is analysed as such an integral measure.

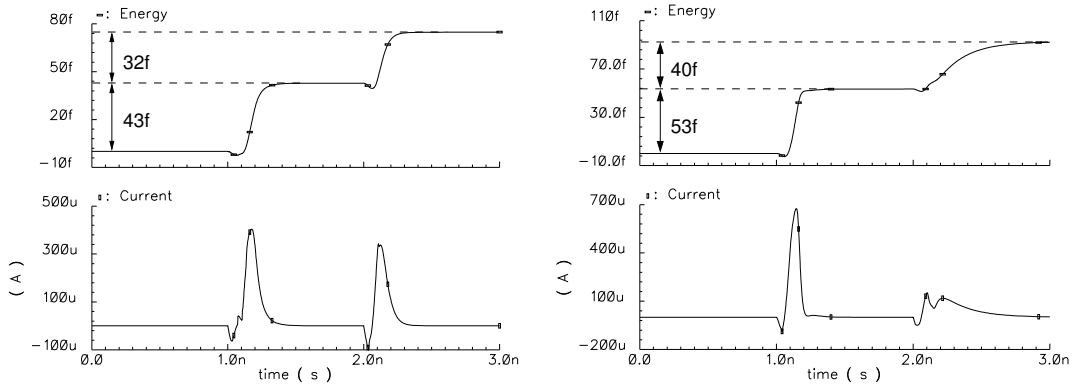
6.2 Energy imbalance

Energy imbalance (further referred to as *imbalance*) can be measured as the variation in energy consumed by a circuit processing different data. If e_1 and e_2 are the energy consumptions of two input patterns, then the numerical value of imbalance is calculated as:

$$d = \frac{|e_1 - e_2|}{e_1 + e_2} \cdot 100\% \tag{6.1}$$

The imbalance in a single-rail circuit is mainly caused by data-dependent quantity of switching events. In a dual-rail circuit with return-to-spacer protocol the number of switching events is constant for every clock cycle. This reduces the data-dependency of power consumption and this is verified in simulation results below. However, the imbalance is not eliminated completely. It still takes place due to the different power consumption of complementary gates which form a dual-rail gate. For example, the power signature of a 2-input dual-rail AND gate (as in Figure 5.1(d)) switching from *all-zeroes spacer* to *code zero* (the OR component is switching) is different from the same gate switching from *all-zeroes spacer* to *code one* (the AND component is switching).

An experiment has been conducted in order to determine the worst case imbalance in a dual-rail gate. The SPICE analog simulator and the AMS-0.35μ design kit were used obtain all the waveforms. For the experiment a 3-input dual-rail NAND gate was chosen. Such a gate consists of one standard 3-input NAND gate and one standard 3-input NOR gate. These gates have the maximum difference between the number of transistor levels in their pull-up and pull-down stacks. Four-input gates are not considered as they may be not implementable in future low-voltage technologies. The current consumption of a gate consists of three components: the input generator current, the gate current and the output load current. In the experiment we determined the gate current, which is the source of the imbalance, by subtracting the input generator current from the overall current and removing the load of the gate. For this the same benchmark was simulated twice, under $V_{CC} = 0V$ and $V_{CC} = 3.3V$. The waveform of the power supply current under $V_{CC} = 0V$ was subtracted from the waveform under $V_{CC} = 3.3V$. A single positive 1ns pulse



(a) 3-input NAND gate

(b) 3-input NOR gate

Figure 6.1: Power signature of non-loaded gates

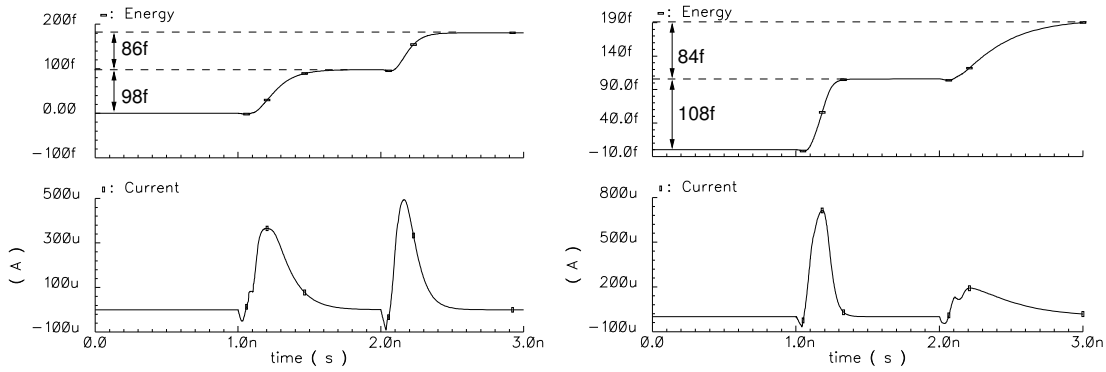
with rise and fall times of 150ps was applied to all the inputs of each gate.

The gate currents are shown in Figure 6.1. The imbalance is obtained by comparison of the energy consumed during switching. The energy waveforms in Figure 6.1 are the integrated current starting from time 0. The full cycle energy imbalance calculated by formula 6.1 is 10.7%. The energy imbalance during the falling transitions of the gates is 10.4% and during the rising transitions it is 11.1%.

The imbalance (being a relative value) becomes smaller if an identical output load is connected to both NAND and NOR gates. Figure 6.2 shows the gate currents when each gate output load is simulated as a pair of capacitors connected to the ground and V_{CC}, each capacitor is 0.016pF (equivalent to 4 inverter inputs). The full cycle energy imbalance value in this experiment is 2.1%. The energy imbalance during the falling transitions of the gates is 4.8% and during the rising transitions it is 1.2%.

In order to show that the 3-input NAND and NOR gates exhibit the worst case imbalance among simple complementary gates the same set of experiments was also conducted for 2-input NAND and NOR gates. The full cycle energy imbalance in this experiment was 8.4% for non-loaded gates and 1.3% for the gates loaded with 0.0032pF capacitors.

The experiments have shown that the worst case imbalance in a dual-rail circuit is 10.7%. This imbalance only occurs if the gates are not loaded. The worst case imbalance under a realistic load



(a) 3-input NAND gate

(b) 3-input NOR gate

Figure 6.2: Power signature of loaded gates

is 2.1%. Further imbalance reduction is possible by either modifying the circuit so that it uses more symmetrical gates or by modifying the gates at the transistor level.

6.3 Exposure time

Exposure time is a period of time during which the energy imbalance is visible. The longer the imbalance is visible the easier it is to measure. This is why the exposure time of the imbalance should be minimised along with the energy imbalance reduction. In a dual-rail circuit the exposure time depends on the spacer protocol. We have evaluated the lower and upper bounds for the exposure time on the single-spacer and the alternating-spacer protocols. The clock cycle is used as a measure of the exposure time.

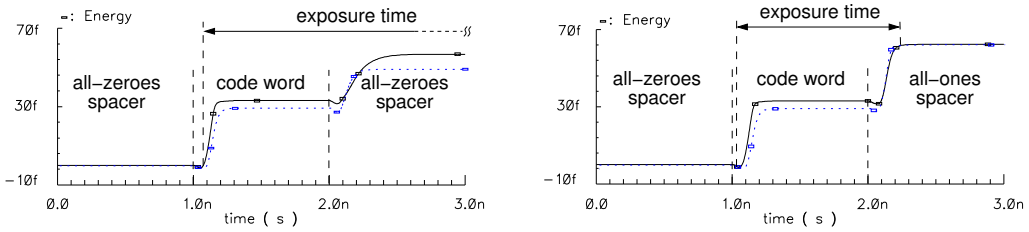
In a dual-rail circuit using the single-spacer protocol the lower bound of exposure time is one clock cycle and the upper bound is the whole time the circuit operates. These bounds can be derived from the analysis of a dual-rail gate operation. The imbalance in a dual-rail gate is caused by switching one of the components of a dual-rail gate. It is visible until the other complementary single-rail gate switches up and down. The lower bound is hit if the switching of the first single-rail gate is delayed as long as possible (until the end of the first half of the clock cycle) and the other single-rail gate switches as early as possible (in the beginning of the second half of the next clock cycle). In this case the exposure time is equal to 0.5 clock cycle. If the combinational logic

delay is small comparing to the clock period, then the lower bound becomes 1 clock cycle.

The upper bound depends upon data. If the gate output switches between alternative code words in each cycle (going through the spacer each time), then the upper bound is 1.5 clock cycles (or 1 clock cycle under the assumption of combinational logic delay being small). In this case, however, the entropy of information at such an output is zero. In order to make the output more informative one can implement somehow (this is not supported by any industrial or to the best of our knowledge by any academic tools) the Manchester serial code at that output. Then the upper bound will be 1.5-2 clock cycle (the second value is for the small combinational logic delay). Finally, if no ad-hoc provisions are made in order to control the sequence of switching, the upper bound becomes undefined. Our benchmarks have shown that the average case of exposure time can be about a dozen of clock cycles.

The exposure time can be reduced by applying the alternating-spacer protocol. For this protocol the exposure time lower boundary is 0 (actually, one gate delay) and upper boundary is one clock cycle. Consider a dual-rail gate operating in alternating-spacer protocol. In the first half of the clock cycle one component of the dual-rail gate fires introducing a data-dependent imbalance. This imbalance is exposed until the second half of the clock cycle when the other complementary component fires leading the energy consumption to a data-independent constant value. If the first single-rail gate fires just before the positive edge of the clock and the complementary gate fires just after the positive edge of the clock then the lower boundary of the exposure time is achieved. The upper boundary is reached if one single-rail gate fires in the very beginning of the clock cycle and the other gate fires in the very end of the same clock cycle. Under a relatively slow clock the exposure time is about half the clock cycle, and gets shorter under a faster clock.

In order to show the influence of the spacer protocol on the exposure time the following experiment was performed using the SPICE analog simulator and the AMS-0.35μ design kit. Single-spacer and alternating-spacer protocols were applied to a 2-input dual-rail AND gate for one clock cycle. In each protocol two different code words were applied to the inputs of the gate: both logical zeroes and both logical ones. The obtained energy waveforms are shown in Figure 6.3, the solid line for both logical zeroes and the dotted line for both logical ones code words. The experiment shows that in the single-spacer protocol the energy imbalance is not compensated in



(a) Single space protocol

(b) Alternating spacer protocol

Figure 6.3: Exposure time for dual-rail 2-input AND gate

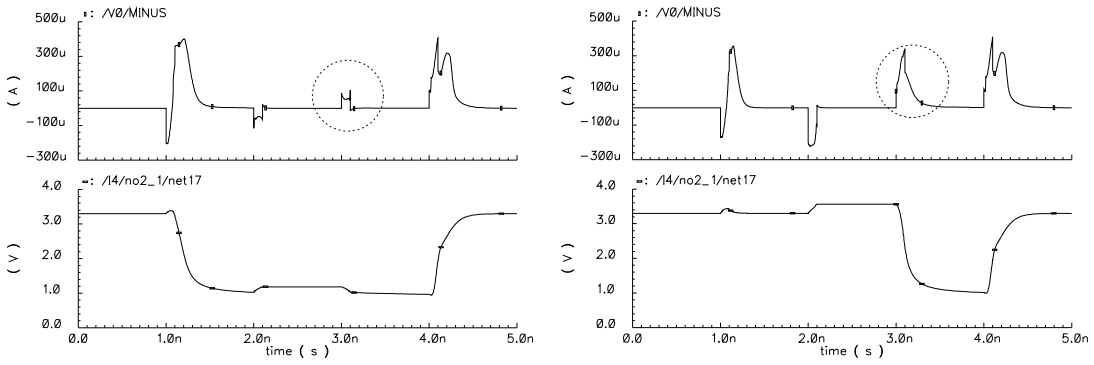
the current clock cycle, but can be potentially compensated in one of the following clock cycles. In the alternating-spacer protocol the imbalance is exposed only between the adjacent spacers, which reduces the exposure time to less than one clock cycle.

6.4 Early propagation and memory effect

Other security-related characteristics of a circuit are *early propagation [46]* and *memory effect* [112, 46]. These characteristics have much less impact on the security features of a circuit than imbalance and exposure time. So, it is essential to minimise the circuit imbalance and exposure time before optimising the circuit for the early propagation and memory effect metrics.

The early propagation is the ability of a gate to fire without waiting for all its inputs. Early propagation causes the data-dependent distribution of circuit switching events in time. The effect of early propagation is bounded by half of the clock cycle. One way to avoid the early propagation is to balance all paths by inserting buffers in such a way that all inputs of each gate arrive simultaneously. In a dual-rail circuit NCL-D gates can be used in order to restrict the early propagation effect to limited areas only.

The memory effect is the ability of a CMOS gate to remember its previous state. It is shown by an example of a 2-input NOR gate simulated under two input sequences: $a = 00100$, $b = 01110$ and $a = 01100$, $b = 00110$, see Figure 6.4(a,b) respectively. The sequences vary in the second bit only. However, the power signature shows a noticeable difference in the fourth bit (marked with dotted circles). This can be explained by the parasitic capacitor between p-transistors which



(a) a,b: 0,0->0,1->1,1-0,1->0,0

(b) a,b: 0,0->1,0->1,1->0,1->0,0

Figure 6.4: Memory effect in OR gate

charges differently when processing the second bit of the input sequences. The capacitor voltage is shown at the bottom of the diagrams. A possible solution is to modify the gates in such a way, that the gates parasitic capacitors were charged independently from input data. For example, a 2-input NOR gate can be implemented with two stacks of p-transistors controlled by the input signals in different orders (i.e. <ab> and <ba>).

6.5 AES benchmark

The method presented in Chapter 5 and the VeriMap tool were tested by design of several cryptographic circuits for our industry partner, a semiconductor company Atmel Inc. These circuits were fabricated and evaluated in Atmel Inc laboratories. However, the results of the experiments are subject to a non-disclosure agreement. Therefore additional experiments were conducted. In this connection we would like to thank Julian Murphy, who tested the tool by producing several security-enhanced AES designs.

Two AES [109, 32] designs are used: *Open core AES [117]* and *AES with computable Sboxes* [66, 122], see Appendix C for details. For each design a single-rail AES circuit was synthesised from an RTL specification by using Cadence Ambit v4.0 tool and AMS-0.35μ library. The VeriMap tool was applied to the netlist generated by Ambit and the dual-rail netlist was produced. The dual-rail circuits were optimised for negative gates and used alternating-spacer

dual-rail protocol. Both single-rail and dual-rail designs were analysed for static delays (SDF delay annotation) and simulated in Verilog-XL v3.10. By adhering to the RTL design flow, the netlists can be directly used in the back-end design tools of Cadence.

A chip implementing the AES algorithm with computable Sboxes has been produced using the fabrication facilities of Europractice. The chip description, its floor-plan and package photos can be found in Appendix C. Building a test-board and setting up power analysis attacks on this chip is one of our future work topics, for now simulation results are used for evaluation of our method.

The rest of this section summarises the experiments performed to characterise the proposed method in terms of security, size and power consumption. The statistics for some parts of AES, namely *ciphers* and *Sboxes,* are shown in Table 6.1, Table 6.2 and Table 6.3.

The purpose of the first experiment is to evaluate the correlation between data and switching activity of the circuits. Switching activity is the number of switching events in the circuit within one clock cycle. Table 6.1 presents the minimum, average and maximum switching activity for the Sboxes and ciphers. These values are obtained by simulating the circuits with a number of input vectors. In single-rail Sbox a transition is determined by a pair of input vectors. The Cartesian product of previous and next 8-bit input vectors includes 256x256=65,536 possible combinations. For simulation of Sboxes 10,000 random pairs of these vectors were chosen. The ciphers were simulated with the 284 vectors supplied with the Open core AES testbench. Note that for dual-rail circuits the switchings of single-rail wires (e.g. reset, clock and the signal which determines the injecting spacer) are also taken into account.

The experiment shows a significant difference between the min/average/max switching activity values for the single-rail Sbox benchmarks. The minimum value is zero, and the maximum values are up to 48% higher than the average values. At the same time, switching activity for the dual-rail circuits is constant. In the single-rail switching activity varies significantly depending on data and clearly there is no switching activity if the input data does not change. In addition many switching events in single-rail circuits are caused by hazards and the single-rail Sbox benchmarks are no exception. Here the hazards caused up to 80% of data-dependent switching events. The number of switching events in dual-rail combinational logic is constant for any input data and is equal to the number of wires (as every second wire switches twice).

benchmark name		switching activity (hazards)		
		min	avg	max
Sbox (open core)	single-rail	0 (0)	162 (33)	277 (124)
	dual-rail	1,180	1,180	1,180
	overhead	∞	628%	326%
Sbox (comp.)	single-rail	0 (0)	525 (345)	936 (746)
	dual-rail	868	868	868
	overhead	∞	65%	-17%
cipher (open core)	single-rail	0	9,147	13,236
	dual-rail	41,285	41,285	41,285
	overhead	∞	351%	211%
cipher (comp.)	single-rail	0 (0)	3,810 (2,013)	6,140 (3,682)
	dual-rail	13,055	13,055	13,055
	overhead	∞	242%	112%

Table 6.1: Switching activity of single-rail and dual-rail circuits

Switching activity in the Open core dual-rail cipher is 351% higher than in the single-rail cipher and 255% higher for the AES design with computable Sboxes. These values are greater than the results for their corresponding combinational logic Sboxes. The bigger difference can be explained by the nature of computations in complex circuits. They execute in bursts, which are defined by the algorithm. Under a burst the switching is similar to our experiments with combinational circuits. However, between the bursts the situation is significantly different: a single-rail circuit is inactive and a dual-rail circuit continues to 'burn power' by switching between code words and spacers.

A possible way to address this issue is to implement *clock gating*. This, however, should be different from the conventional clock gating technique. It is important to make it data-independent. At this stage we do not see a feasible way of implementing this at the netlist level. Most likely it will require analysis of behavioural specifications. We view this idea as a subject of future work.

In order to compare the security features of single-spacer and alternating-spacer circuits, the AES design with computable Sboxes was also converted into single-spacer dual-rail. Both single-spacer and alternating-spacer dual-rail implementations were simulated with 284 input vectors from the standard AES testbench in the encryption and decryption modes. The switching activities of "1" and "0" rails were recorded separately. Table 6.2 shows the worst case difference in switching activity between "1" and "0" rails. The imbalance between the number of switching

benchmark name		switching activity	
		single-spacer	alternating-spacer
cipher (encryption)	rail_1	8,388	6,505
	rail_0	4,622	6,505
	imbalance	29%	0%
cipher (decryption)	rail_1	8,572	6,505
	rail_0	4,438	6,505
	imbalance	32%	0%

Table 6.2: Switching activity in dual-rail wires

events in the rail_1 and rail_0 is calculated as $imbalance = \frac{|rail\_1 - rail\_0|}{rail\_1 + rail\_0} \cdot 100\%$. While the total switching activity is the same in both implementations, the single-spacer implementation exhibits significant differences in the number of switching events on the complementary rails. As the complementary gates within a dual-rail gate have different power consumptions, the power signature of the single-spacer dual-rail circuit becomes dependent on the processed data. Alternating-spacer dual-rail circuits do not suffer from this leakage because all gates are switching in every clock cycle. The cost of improved security features is the increase in the number of gates, wires and area.

Table 6.3 summarises the size of circuits in terms gates, transistors, wires and estimated area. The benchmarks indicate only 84-88% overhead in gate numbers (a positive gate is counted as a pair of a negative gate and an inverter) for AES design with computable Sboxes. This is less than 100% due to the negative gate optimisation. For Open core design the overhead is more than 100% due to the structure of its Sbox module. During the negative logic optimisation of Open core Sbox more inverters were inserted into a non-critical path (as components of spacer inverters) than removed from the critical path.

The number of wires is increased by 117-145%. Wires are duplicated in a dual-rail circuit and then spacer converters are added, further increasing the number of wires.

The estimated area of the benchmarks combinational logic indicates a 102%-127% overhead. A significant area increase for flip-flops (228%-289%) can be explained by using dual-rail flip-flops constructed out of standard logic gates. This can be improved by transistor level optimisation of the flip-flops.

Figure 6.5 visualises the security improvement for the AES block. These diagrams have been

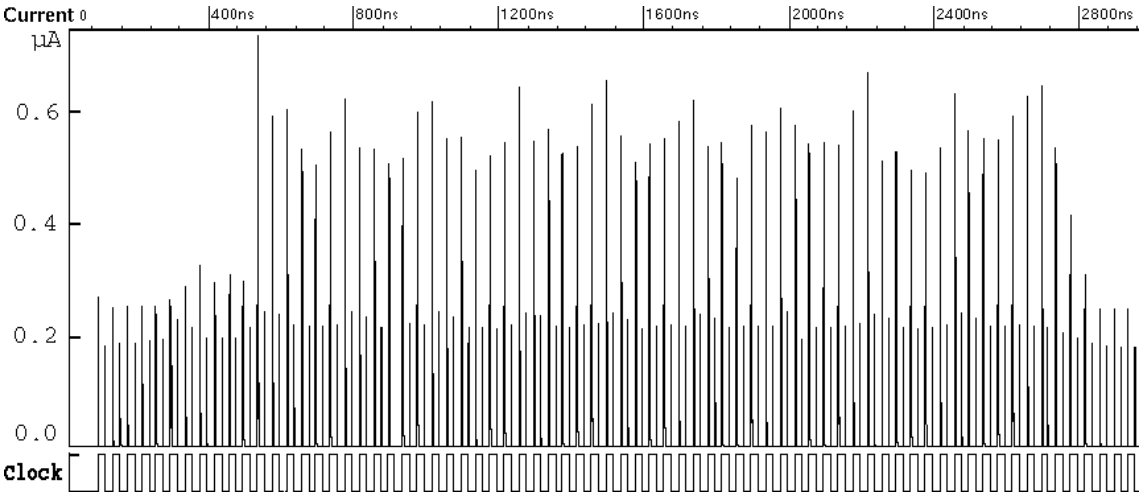
benchmark name		neg.gate count	transistor count	wire count	estimated area	
					CL	FF
Sbox (open core)	single-rail	655	3,180	482	44,593	0
	dual-rail	1,523	6,672	1,180	101,364	0
	overhead	133%	110%	145%	127%	0
Sbox (comp.)	single-rail	634	2,362	400	32,975	0
	dual-rail	1,164	4,628	868	68,603	0
	overhead	84%	96%	117%	108%	0
cipher (open core)	single-rail	12,752	68,184	9,980	873,175	142,370
	dual-rail	26,396	139,828	24,367	1,925,190	466,870
	overhead	107%	105%	144%	120%	228%
cipher (comp.)	single-rail	10,372	50,344	5,936	580,046	118,678
	dual-rail	19,510	95,066	13,055	1,237,260	462,021
	overhead	88%	89%	120%	113%	289%

Table 6.3: Size of single-rail and dual-rail circuits

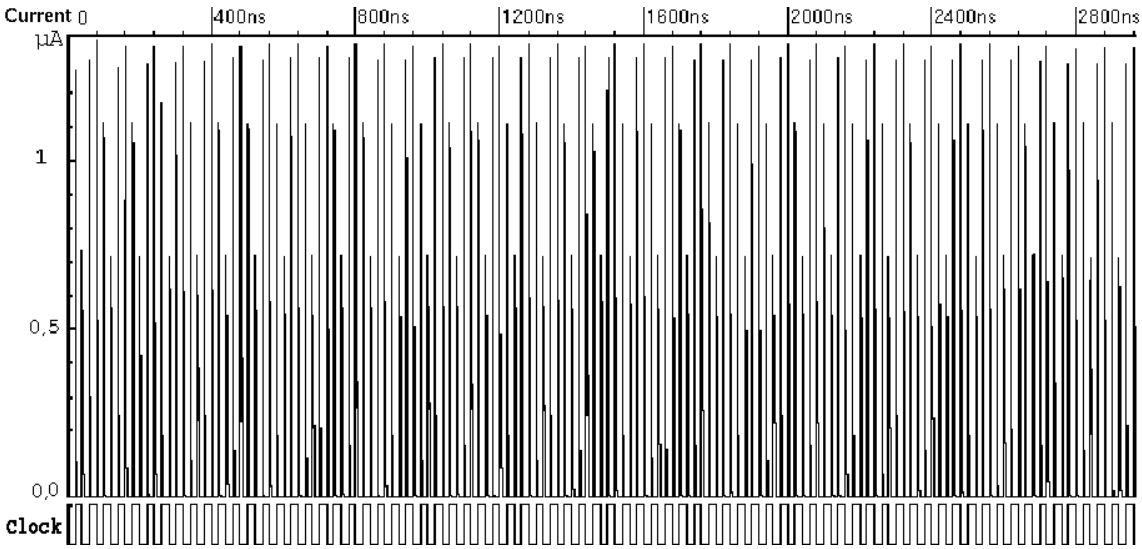
generated from the AES design versions with computable Sboxes: in single-rail and in dual-rail using the alternating-spacer protocol. As the Open core AES design yielded similar plots they are not shown. Figure 6.5(a) clearly shows the initial operation of the circuit and the AES computation phases. The first peaks reveal the data being clocked into the circuit, the middle peaks show the iterative rounds being performed and the last peaks show the data being clocked out. On the other hand looking at Figure 6.5(b), the operation is masked, now the 'clocking in and out' and AES computation rounds are indistinguishable from one another. The repetitive peaks correspond to the spacer and data alternation.

The diagrams were generated using Synopsys Nanosim mixed-signal simulation software, which permits fast mix-signal simulation up to 100 times faster than a pure SPICE simulation. The single-rail and dual-rail AES implementations were simulated using the same, randomly chosen, key and input data. Simulations were performed with different keys and input data to ensure fairness; similar plots for the dual-rail implementation were also spawned from each simulation, which confirms the improvement in security.

The security improvement in combinational logic blocks is illustrated in the example given in Figure 6.6. The Open core Sbox was simulated under 16 random data values. The diagram in Figure 6.6(a) shows the power signature in the single-rail implementation and the four diagrams in Figure 6.6(b) show the power consumption of the dual-rail implementation with alternating-spacer



(a) Single-rail implementation



(b) Dual-rail implementation, alternating spacer protocol

Figure 6.5: Power signature for AES design with computable Sboxes

protocol. From these diagrams one can see a significant variation in power consumption of the single-rail circuit computing different input data. The variation of the current is much less visible in the dual-rail circuit. The difference of the curves is due to the effects of early propagation only, i.e. the integrated area under the current curve in dual-rail circuit is a constant value invariant of input data. This is a significant security improvement comparing to the single-rail implementation.

The restriction of the early propagation effects will be addressed in our future work.

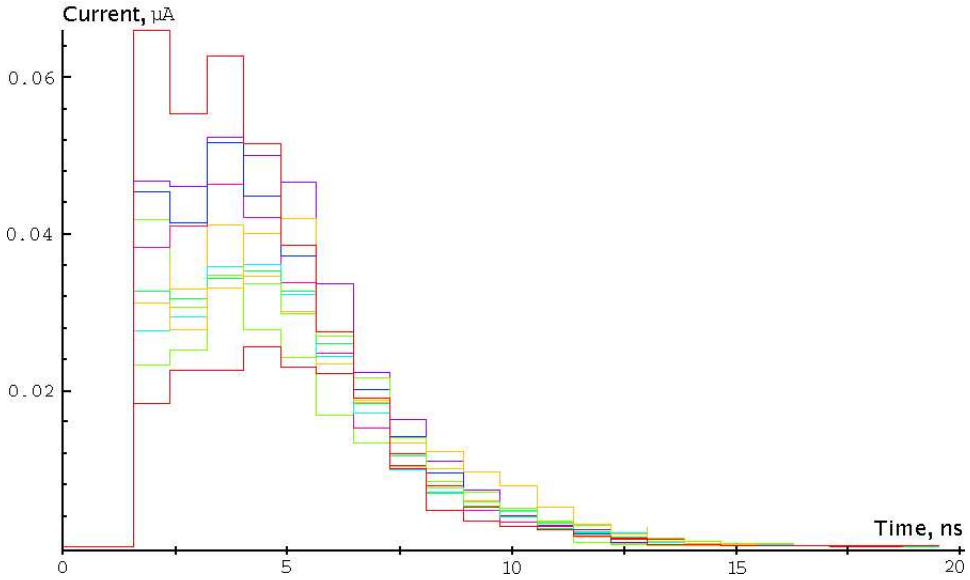
It is clear that in the AES designs there are opportunities to minimise power consumption as not all logic is necessarily being used all the time. Industry synthesis tools can identify sleep mode logic and use this information to annotate places in the netlist which could be committed to sleep mode logic later in the design flow. This low power optimisation could be utilised in our dual rail circuitry, one approach would be to put a spacer on the input to the identified sleep mode logic and holding this there for the clock cycles whilst it is not used. By doing so the switching is now zero, thus saving power. This technique would not reveal data as the sleep mode logic is in a "meaningless" spacer state. By using the synthesis tool to identify the sleep mode logic we are adhering to the RTL design flow and our conversion tool could use the annotated netlist to apply the optimisation to dual rail circuits; note the committal stage of the sleep mode logic would need to be different to what the synthesis tools would do (simple AND gates using a control signal). Presently this has not been implemented in the tool but investigated using schematic entry with simple examples which gave promising results. This needs to be investigated further together with the clock gating idea.

6.6 Summary

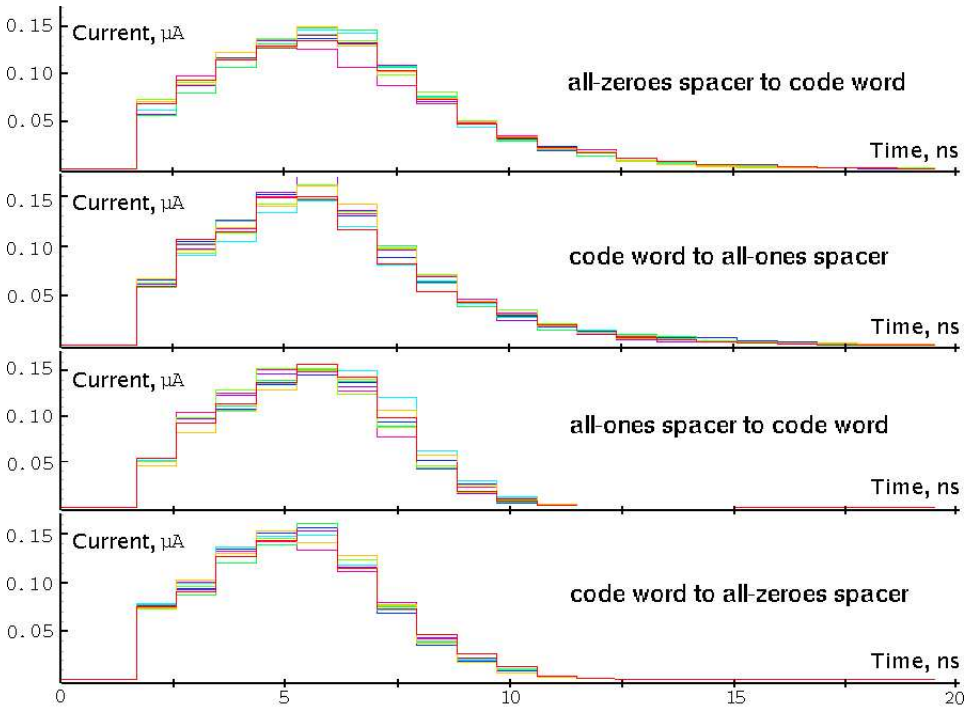
Two security metrics introduced in this chapter are energy imbalance and exposure time. They have been used for a comparative analysis of single-rail, traditional dual-rail and alternating-spacer dual-rial circuits at level of gates. In these experiments the dual-rail circuits with alternating-spacer protocol have shown the least dependency between the processed data and power consumption.

Two AES designs were used for benchmarks: AES with computable Sboxes and Open core AES. The single-rail AES circuits ware synthesised using the standard RTL design flow. For each single-rail AES implementation two dual-rail circuits were obtained by the VeriMap tool: one with a single-spacer protocol and the other with an alternating-spacer protocol. These circuits were compared in terms of area, security and power consumption (switching activity and hazards).

The AES benchmarks indicate that the dependency between data and switching activity, which exists in traditional dual-rail circuits, is fully eliminated in our method with alternating-spacer protocol. The price to pay for the improved security features is the increased average switching



(a) Single-rail implementation



(b) Dual-rail implementation, alternating spacer protocol

Figure 6.6: Power signature for Open Core Sbox

activity and area overheads. The ways of reducing the size and switching activity of dual-rail circuits are outlined as topics for future work.

A chip implementing the AES algorithm with computable Sboxes has been produced using the fabrication facilities of Europractice. Building a test-board and setting up power analysis attacks on this chip is one of our future work topics.

Chapter 7

Conclusions

This thesis presents a framework for automated synthesis of asynchronous circuits from high-level specifications using a direct mapping approach. The use of high-level HDLs allows the designers with limited asynchronous design experience to create circuits at a much lower cost. The direct mapping approach facilitates low algorithmic complexity and transparent correspondence between the initial specification and resultant circuit. The proposed methods were implemented in a set of software tools which interface the conventional EDA tools and form a coherent design flow. The tools and the underlying methods were evaluated on a number of benchmarks. Several chips designed with these tools were fabricated and tested by a semiconductor company Atmel Inc.

7.1 Summary of contribution

The intermediate model in the proposed synthesis framework is Petri nets, used with various interpretations. Expressing advanced concurrency and timing paradigms Petri nets are ideal for the modelling of self-timed systems. Chapter 2 gave a necessary background in the asynchronous systems and in the Petri nets modelling language.

A coherent subset of synthesis methods for self-timed circuits from high-level HDLs was reviewed in Chapter 3 . Two main synthesis concepts were studied syntax-driven translation and logic synthesis.

In syntax-driven translation the language statements are mapped into circuit components and the interconnect between the components is derived from the syntax of the system specification.

This approach is attractive from the productivity point of view, as it avoids computationally hard global optimisation of the logic. However, the direct translation of the parsing tree into a circuit structure may produce very slow control circuits and the lack of global optimisation may not meet the requirements for high-speed circuits.

In logic synthesis the initial system specification is transformed into an intermediate behavioural format convenient for subsequent verification and synthesis. Between syntax driven-translation and logic synthesis the latter produces smaller circuits with faster control logic. It also offers more space for optimisation of the control path. The existing logic synthesis design flow, however, has several drawbacks:

- The design flow lacks an automatic synthesis of hazard-free data path components.
- The data path synthesis is unacceptable for security applications due to dependency between processed data and power consumption.
- Synthesis of the control path described at STG level exhibits high algorithmic complexity.
- The control path obtained from an STG exhibits high and unpredictable input-output latency.

These problems were addressed in the framework proposed in this thesis. The framework is based on the existing logic synthesis design flow reviewed in Chapter 3. This design flow is enriched by direct mapping methods for synthesis of low-latency control logic and hazard-free data path components. The resultant design flow called BESST [101] reuses existing methods for high-level partitioning and scheduling, splitting the system into data path and control paths, and deriving their intermediate Petri net representations [16, 96].

The main idea of the direct mapping approach is that a graph specification of a system is translated into a circuit netlist in such a way, that the graph nodes correspond to the circuit elements and graph arcs correspond to the interconnects. The key features of the direct mapping approach are low algorithmic complexity and transparent correspondence between the elements of the initial specification and the components of the resultant circuit. The low computational complexity allows processing large specifications and applying different combinations of peephole optimisations to find the best solution. The transparency of the approach is advantageous for checking the functional correctness of the implementation.

In Chapter 4 a method for the direct mapping of control circuits from STGs was presented. In this method an STG is, firstly, split into a device and an environment, which synchronise via a communication net modelling wires. The device is then represented as a tracker and a bouncer. The tracker follows the state of the environment and is used as a reference point by the device outputs. The bouncer interfaces the environment and generates output events in response to the input events according to the state of the tracker. This two-level device architecture provides an efficient interface to the environment and is convenient for subsequent mapping into a circuit netlist. The optimisation and mapping techniques have low algorithmic complexity, and the obtained circuits exhibit low latency.

Chapter 5 presented a method for converting a conventional RTL data path into a hazard-free circuit. In this method the hazard-free logic is obtained by use of a dual-rail encoding with a return-to-spacer signalling. A new alternating-spacer protocol with two spacers interchanging in time is proposed. This switching discipline makes all gates switch per computation cycle, which can be used for security circuits, online testing and refreshing of dynamic logic. In scope of this thesis a set of converters for integrating a dual-rail circuit into a single-rail environment were designed; several implementations of a secure dual-rail flip-flop and latch were developed.

The security application of the alternating-spacer protocol is studied in depth in Chapter 6. Two spacers alternating in time within the dual-rail logic help to balance switching activity and energy consumption per clock cycle, thus making power analysis more difficult. In order to estimate the ability of alternating-spacer protocol to resist power analysis attacks, two security metrics, energy imbalance and exposure time, are introduced and used on a set of cryptographic benchmarks.

As a result of this work, several software tools were developed, namely the OptiMist tool for the direct mapping of low-latency asynchronous controllers from STGs, and the VeriMap tool for synthesis of hazard-free data path components. The tools are available for download from http://www.async.org.uk/.

OptiMist takes an STG as the initial specification of a controller, converts it to a form convenient for mapping, performs optimisation, and produces a Verilog netlist of the circuit. The STG optimisation relies on a set of heuristics aimed at circuit latency and size reduction. As the optimisation is performed locally, the computation time grows linearly with the size of specification.

This allows to process large specifications which are not computable by logic synthesis tools in acceptable time. OptiMist is fully automated, yet a designer can significantly influence the result by choosing one or more optimisation heuristics. In combination with computation speed OptiMist gives the designer an opportunity to synthesise circuits with different optimisation parameters and choose the best solution. The tool can be employed in combination with Cadence for simulation and technology mapping of circuits.

VeriMap automatically converts a single-rail circuit into a clocked or a self-timed dual-rail circuit. The tool successfully interfaces to the Cadence CAD tools. It takes as input a single-rail circuit netlist, created by Cadence Ambit, and converts it into a dual-rail Verilog netlist. The conversion procedure is defined by a set of libraries specific for the technology parameters. VeriMap is supplied with the libraries for the AMS 0.35μ CMOS technology which can be adjusted to the other design technologies.

Several cryptographic circuits were designed in collaboration with our industry partner Atmel Inc using the VeriMap tool kit. These circuits were fabricated and evaluated in the company's laboratories. However, the results of the experiments are subject to a non-disclosure agreement. Therefore additional experiments were conducted in order to evaluate the resistance of our circuits to the power analysis attacks. The AES algorithm was chosen for this experiment. A circuit implementing AES was designed using the VeriMap tool and has been produced using the open fabrication facilities of Europractice. Building a test-board and setting up power analysis attacks on this chip is one of our future work topics.

7.2 Future work

There are two main drawbacks in using the dual-rail encoding with the return-to spacer protocol: increased power consumption and size of the resultant circuit. The former can be improved by clock gating and data guarding techniques. The latter can be addressed by selective conversion into dual-rail logic and by a more sophisticated strategy for inserting completion detection logic.

Our experiments have shown a higher switching activity in dual-rail circuits compared to the corresponding single-rail logic. It can be explained by the nature of computations in complex circuits. They execute in bursts, which are defined by the algorithm. Under a burst the switching

is similar in both single-rail and dual-rail circuits. However, between the bursts the situation is significantly different: a single-rail circuit is inactive and a dual-rail circuit continues to 'burn power' by switching between code words and spacers. A possible way to address this issue is to implement clock gating. This, however, should be different from the conventional clock gating technique. It is important to make it data-independent. At this stage we do not see a feasible way of implementing this at the netlist level. Most likely it will require analysis of behavioural specifications.

There is another opportunity to minimise power consumption as not all logic is necessarily being used all the time. Industry synthesis tools can identify this parts of logic and annotate places in the netlist which could be committed to sleep mode logic later in the design flow. This low power optimisation can also be utilised in dual-rail circuitry. One approach would be to put a spacer on the input to the identified sleep mode logic and hold this there for the clock cycles whilst it is not used. By doing so the switching is brought to zero, thus saving power. This technique would not reveal data as the sleep mode logic is in a 'meaningless' spacer state. By using the synthesis tool to identify the sleep mode logic we are adhering to the RTL design flow and our conversion tool could use the annotated netlist to apply the optimisation to dual-rail circuits; note the committal stage of the sleep mode logic would need to be different to what the synthesis tools would do. Presently this has not been implemented in the tool but investigated using schematic entry with simple examples which gave promising results.

In a real circuit some parts of its data path are not secret, e.g. selection of an operating mode. This gives a space for circuit size optimisation by converting into dual-rail only that parts of the data path which are critical for the circuit security. Currently this is possible by semi-automatic partitioning of a circuit into sets of secret and a non-secret modules. Only secret modules are subsequently converted into a secure dual-rail. However, this approach is effective only for the circuits with high granularity of the modules. Indeed, it is not practical to convert a big module just because of a couple of secret wires. Moreover, the approach is not applicable to flat netlists, where the whole circuit is described as one module.

From a designer perspective it would be convenient just to indicate the buses which carry the secret data and allow the tool to calculate the logic which should be converted in order to keep the

power consumption independent of that data. The interfaces between the dual-rail parts of the data path and the rest of the circuit are supported by converters (single-rail to dual-rail and dual-rail to single-rail). From the benchmarks we learnt, that sometimes a small portion of non-secure data path, being converted into dual-rail, decreases the number of interface signals significantly, thus decreasing the number of converters. An optimisation algorithm is required, which trades off the number of interfaces and the number of non-secure gate to be converted.

The size of a self-timed dual-rail circuit can be decreased by further optimising its completion detection logic. In particular, the layer-wise optimisation can be improved by increasing the number of layers without completion detection. The original layer-wise optimisation is too pessimistic. In the phase of circuit switching from spacers into code words, it requires that the completion signal is issued only after all gates in the combinational logic have switched to code words. However, it is possible to have some gates still switching while generating the completion signal, providing that the flip-flop inputs hold code words and those gates finish switching by the time a spacer arrives to their inputs. Thus, a wave of spacers can start propagating through the combinational logic, while the gates in front of this wave finish their switching to code words. Careful timing analysis is required to guarantee that the wave of spacers does not overtake the wave of code words. Similar reasoning is applicable to the phase of circuit switching from code words to spacers. This approach has shown promising results on simple benchmarks, yet it requires further research.

Appendix A

OptiMist user manual

This appendix explains how to install and use the OptiMist toolkit for direst mapping of asynchronous circuits from STGs. The toolkit contains the following tools:

- `optimist` - a wrapper script which is a front-end to the OptiMist tools;
- `om_detect` - a tool for detection of redundant places;
- `om_expose` - a tool for exposure of the outputs;
- `om_transform` - a tool for elimination of redundant places;
- `om_verilog` - a tool for mapping of the optimised specification into a circuit;
- `om_lib` -generation of a library of required DCs and FFs either at transistor- or gate-level;
- `om_graph` - a tool for visualisation of an STG with read-arcs extension and tracker-bouncer structure.

Section A.1 explains how to install the OptiMist tools. These tools work with files in ASTG format, which is presented in Section A.2. The usage of the tools is described in Section A.3.

A.1 Installation

The compilation of OptiMist tools from source is done by issuing the following command:

```
$ make
```

This will compile six tools which form the OptiMist package: `om_detect`, `om_expose`, `om_transform`, `om_verilog`, `om_library` and `om_graph`. These tools can ether be used individually or by means of `optimist` script. The script is present in the source files directory. It connects all OptiMist tools in one command-line interface. In order to start using the OptiMist tools put them into a directory which is referenced from the `PATH` environment variable (e.g. `/usr/local/bin/`). This can be done by running the following command as root:

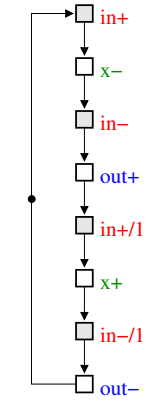
```
$ make install
```

A.2 ASTG format

The OptiMist tools read and write system description in ASTG format. ASTG language captures STG in a human-readable ASCII format without layout information. This format is used by such synthesis tools as Petrify and SIS.

The ASTG format can be described using the STG of a toggle circuit shown in Figure A.1. All the words in bold font are keywords of ASTG language. The file stars with an optional description of the model after the `.model` keyword. It is followed by space-separated lists of input, output and internal signals preceded by the `.input`, `.output` and `.internal` keywords respectively. If there were dummies in the STG they would be listed after the `.dummy` keyword.

```
.model Toggle
.input in
.output out
.internal x
.initial state !in x !out
.graph    # List of arcs
in+ x-
x- in-
in- out+
out+ in+/1
in+/1 x+
x+ in-/1
in-/1 out-
out- in+
.marking { < out-, in+ > }
.end
```



model: Toggle
input: in
output: out
internal: x

Figure A.1: Toggle STG

The initial states of the signals are defined in a space-separated list after the `.initial state` keyword. Each signal state is the signal name with or without preceding exclamation mark '!'. The signals whose names are preceded by the exclamation mark are initially low and all the other signals are initially high. The explicit definition of the initial state is optional as state of each signal can be calculated from the initial marking.

The main part of the ASTG file is a list of arcs between the STG nodes ((places and transitions) started with the `.graph` keyword. Each line in this list defines one or more arc. The line starts with a place or a transition from which the arcs originates. It is followed by a space-separated list of places and transitions where the arcs connect to. It is easy to distinguish transitions from places because their names contain '+' and '-' signs. A signal transitions of the same polarity can be differentiate by their index. The index follows the transition name after a back slash '/'. If there is an arc from one transition to another transition it means that there is an implicit place on this arc.

The initial marking is defined as a space-separated list of places and/or arcs (in case of implicit places) containing a token. An ASTG file is closed by `.end` keyword. The comments in an ASTG file start with hash symbol '`#`' and finish by the end of the line.

OptiMist uses only a subset of ASTG language which is related to the Petri net description. It does not handle the extensions for channels, delays and timing assumptions. The grammar of this subset of ASTG language follows.

```
stg
    : model signals initial_state graph marking end
model
    : // Empty
    | ".model" NAME NL
signals
    : // Empty
    | signals signals_declaration
signals_declaration
    : ".inputs" signal_list NL
    | ".outputs" signal_list NL
    | ".internal" signal_list NL
    | ".dummy" signal_list
signal_list
    : // Empty
```

```
    | signal_list NAME
initial_state
    : // Empty
    | ".initial state" state_list NL
state_list
    : // Empty
    | state_list state
state
    : NAME
    | '!'  NAME
graph
    : ".graph" line_list
line_list
    : // Empty
    | line_list NL
    | line_list line NL
line
    : node arc_list
arc_list
    : // Empty
    | arc_list node
node
    :  trans instance_opt
    | place_or_dummy
trans
    :  NAME '+'
    | NAME '-'
place_or_dummy
    :  NAME instance_opt
instance_opt
    : // Empty
    | '/' NUMBER
marking
    : ".marking" '{' marking_list '}' NL
marking_list
    : // Empty
```

```
    | marking_list marking
marking
    : node
    | edge
edge
    : '<' node ',' node '>'
end
    : ".end"
```

In this grammar the words in bold font are the keywords of the ASTG language. The parts of the lines after the double back slash '//' are comments. The capitalised words have special meaning: NL is one ore more 'new line' characters; NUMBER is an integer positive number; NAME is a string which consists of alphabetic characters, digits and underscore symbols and starts with either a character or an underscore symbol.

A.3 Usage of OptiMist tools

There are two ways of using the OptiMist tools: basic and expert. In the basic mode a the user interacts with the `optimist` wrapper script, which runs necessary OptiMist tools and passes parameters to them. In the expert mode the user runs the OptiMist tools explicitly, which gives more control on optimisation parameters.

A.3.1 OptiMist wrapper

The easiest way to process an STG file is to give it as a parameter to the `optimist` script:

```
$ optimist file.g
```

This script calls OptiMist tools with necessary command-line parameters and creates the following files:

`file_3.g` - STG with redundant places detected using *optimise_choice*, *optimise_latency* and *optimise_size* heuristics

`file_3e.g` - STG with redundant places detected and outputs exposed

`file_3et.g` - STG from which redundant places are removed

By default the input STG is optimised using all heuristics. The minimum number of DCs in a loop is set to 3. The maximum join transition fanin is limited by 2. This options can be overridden by using command-line parameters whose full list of is shown below:

```
USAGE:
  optimist [OPTIONS] INPUT_FILE_NAME
OPTIONS:
  -l, --level[N]         - level of redundant places detection (N=0,1,2,[3])
      0 = all places are tagged as mandatory
      1 = optimise_choice heuristic is applied
      2 = optimise_choice and optimise_latency heuristics are applied
      3 = use optimise_choice, optimise_latency and optimise_size heuristics
  -r, --redundant PLACE_LIST - list of predefined redundant places []
  -s, --separator PLACE_LIST - list of predefined mandatory places []
  -n, --num-loop[N]      - minimum number of DCs in a loop (N=1,2,[3])
  -j, --join-fanin[N]    - maximum fanin of join transitions (N=[0],1,2,3)
      0 = no control on the join transitions fanin
    1-3 = restrict join transitions fanin to the given number,
          higher value for this parameter is not practical
  -v, --verilog          - map STG into Verilog netlist
  -i, --info             - include circuit statistics into netlist
  -t, --test             - generate circuit ready for off-line testing
  -g, --graph            - draw STG in PostScript format
  -d, --debug            - print debug information
  -h, --help             - print this help
```

For example, the same file can be processed with the following command-line parameters:

```
$ optimist -l2 -v -g file.g
```

or

```
$ optimist --level2 --verilog --graph file.g
```

This commands will produce the following set of files:

`file_2.g` - STG with redundant places detected using *optimise_choice* and *optimise_latency* heuristics

`file_2e.g` - STG with redundant places detected and outputs exposed

`file_2et.g` - STG from which redundant places are removed

`file_2et.v` - Verilog netlist obtained by mapping the STG into DCs and FFs

`file_2.ps` - PostScript file for the `file_2.g` STG

`file_2e.ps` - PostScript file for the `file_2e.g` STG

`file_2et.ps` - PostScript file for the `file_2et.g` STG

Following is the description of individual OptiMist tools and their command line parameters.

A.3.2 OptiMist tool for detection of redundant places

USAGE:

```
om_detect [FLAGS] [IN_FILE_NAME]
```

OPTIONS:

```
-o, --output OUT_FILE_NAME - the name of file to output the result [STDOUT]
-l, --level[N]        - level of redundant places detection (N=0,1,2,[3])
    0 = all places are tagged as mandatory
    1 = optimise_choice heuristic is applied
    2 = optimise_choice and optimise_latency heuristics are applied
    3 = use optimise_choice, optimise_latency and optimise_size heuristics
-n, --num-loop[N]     -  minimum number of DC in a loop (N=1,2,[3])
-f, --fork            - consider places after fork as mandatory
-j, --join            -  consider places after join as mandatory
-c, --choice          -  consider choice places as mandatory
-m, --merge           -  consider merge places as mandatory
-t, --token           -  consider places marked with a token as mandatory
-r, --redundant       -  PLACE_LIST list of predefined redundant places []
-s, --separator       -  PLACE_LIST list of predefined mandatory places []
-d, --debug[N]        -  level of debug information (N=[0],1,2,3,4)
-v, --version         -  print version and copyright
-h, --help            -  print this help
```

A.3.3 OptiMist tool for exposure of outputs

USAGE:

```
om_expose [FLAGS] [IN_FILE_NAME]
```

OPTIONS:

```
  -o, --output OUT_FILE_NAME - output file [STDOUT]
  -d, --debug[N]        - level of debug information (N=[0],1,2,3,4)
  -v, --version         - print version and copyright
  -h, --help            - print this help
```

A.3.4 OptiMist tool for elimination of redundant places

USAGE:

```
  om_transform [FLAGS] [IN_FILE_NAME]
```

OPTIONS:

```
  -o, --output OUT_FILE_NAME - output file [STDOUT]
  -l, --level[N]        - level of transformation (N=0,1,2,3,4,[5])
      0 = remove transient arcs only
      1 = 0 + move initial marking to mandatory places
      2 = 1 + recalculate context signals
      3 = 2 + eliminate redundant places
      4 = 3 + remove transient arcs, not connected places and transitions
      5 = 4 + simplify join transitions
  -j, --join-fanin[N]   - max join transitions fanin (N=[0],1,2,3)
      0 = no control on the join transitions fanin
    1-3 = restrict join transitions fanin to the given number,
          higher value for this parameter is not practical
  -d, --debug[N]        - level of debug information (N=[0],1,2,3,4)
  -v, --version         - print version and copyright
  -h, --help            - print this help
```

A.3.5 Optimist tool for mapping STG into Verilog netlist

USAGE:

```
  om_verilog [FLAGS] [IN_FILE_NAME]
```

OPTIONS:

```
  -o, --output OUT_FILE_NAME - output file [STDOUT]
  -t, --test            - generate circuit ready for on-line testing
  -s, --statistics      - include statistic comments into Verilog netlist
  -dc,--david_cell      - provide detail DC statistics
  -ff,--flip_flop       - provide detail FF statistics
  -i, --input           - provide input signals statistics
```

```
  -d, --debug[N]        - level of debug information (N=[0],1,2,3,4)
  -v, --version         - print version and copyright
  -h, --help            - print this help
```

A.3.6 OptiMist tool for generation of Verilog netlist for DCs and FFs

USAGE:

```
  om_lib [FLAGS] [ELEMENT_NAME]
```

OPTIONS:

```
  -o, --output OUT_FILE_NAME - output file [STDOUT]
  -f, --format[N]       - format of the output data (N=0,1,2,[3])
    0 = check the validity of the element name
    1 = pins description
    2 = gate-level Verilog netlist
    3 = transistor-level Verilog netlist
  -d, --debug[N]        - level of debug information (N=[0],1,2,3,4)
  -v, --version         - print version and copyright
  -h, --help            - print this help
```

A.3.7 OptiMist tool for drawing STG using GraphVis DOT format

USAGE:

```
  om_graph [FLAGS] [IN_FILE_NAME]
```

OPTIONS:

```
  -o, --output OUT_FILE_NAME - output file [STDOUT]
  -l, --legend          - show legend
  -c, --connect         - connect read-arcs to places
  -i, --inputs          - show elementary cycles for inputs
  -d, --debug[N]        - level of debug information (N=[0],1,2,3,4)
  -v, --version         - print version and copyright
  -h, --help            - print this help
```

Appendix B

VeriMap user manual

This appendix explains how to install and use the VeriMap toolkit. VeriMap improves the security of RTL circuits by balancing the power consumption of the circuit. The balancing is achieved by means the dual-rail signalling with return-to-spacer protocol.

There are two alternatives for the return-to-spacer protocol: *single-spacer* or *alternating-spacer*. The former is the traditional protocol for dual-rail circuits. It relies on the assumption that the rails of the circuit are symmetrical in terms of power consumption. The latter protocol makes each gate in a dual-rail circuit switch twice within a clock cycle brining the power consumption to a constant value for each cycle. The imbalance still exists, but it is visible only for less than a clock cycle.

There are two possibilities for the implementation of completion signal: *clocked* or *self-timed*. In the former approach the phases of data and spacer propagation through the logic are controlled by the clock signal. It relies on the assumption that the clock period is long enough to complete both phases. In the latter approach the completion signal is computed by a completion detection logic. This allows to start the spacer phase of the protocol as soon the data propagates through the combinational logic, thus reducing the time when the imbalance is visible.

The obtained dual-rail circuit can be automatically put in a single-rail wrapper to preserve the interface between the circuit and the environment.

The installation of VeriMap is explained in Section B.1. Both the input and output of VeriMap are circuit netlists in Verilog language, which is described in Section B.2. The usage of the tool is

described in Section B.3.

B.1 Installation

The compilation of VeriMap tool from source is performed by the following command:

```
$ make
```

This will compile a binary called `verimap`. In order to start using the tool put it into a directory which is referenced from the `PATH` environment variable (e.g. `/usr/local/bin/`). This can be done by running the following command as root:

```
$ make install
```

B.2 Structural Verilog

The Verilog language provides the means of describing a digital system at a wide range of levels of abstraction. It uses the behavioural constructs at the high level and structural constructs at the low level of abstraction. The behavioural and structural constructs can be mixed. The VeriMap tool uses only the structural subset of Verilog language, also known as Structural Verilog.

In Verilog the system is described as a set of modules. Each of the modules has an interface to other modules and the description of its contents. Usually a circuit is divided into modules based on their logic function, similar to writing a program in a structural programming language. The modules are then interconnected by means of nets connected to their interfaces.

The Structural Verilog syntax is explained on the full-adder example presented in Figure B.1. There are two modules defined:: *half_adder* and *full_adder*. The *half_adder* module has two inputs (*a* and *b*), two outputs (*s* and *c*) and consists of two library gates (instance *inst_s* of 2-input XOR-gate and instance *inst_c* of 2-input AND-gate). The gates use the module inputs to form the outputs. The *full_adder* module contains the instance library gate *OR* and the instances of the previously defined *half_adder* module.

```
module half_adder (a, b, s, c);
  input a, b;
  output s, c;
  XOR inst_s (.A(a), .B(b), .Q(s));
  AND inst_c (.A(a), .B(b), .Q(c));
endmodule
module full_adder (a, b, c, s, co);
  input a, b, c;
  output s, co;
  wire p, g, r;
  half_adder inst_1 (a, b, p, g);
  half_adder inst_2 (c, p, s, r);
  OR inst_co (.A(r), .B(g), .Q(co));
endmodule
```

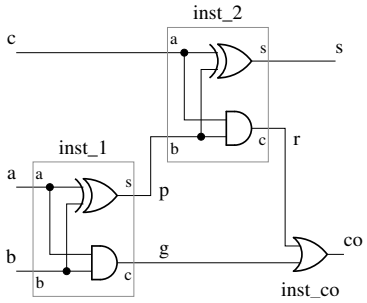


Figure B.1: Full adder in Verilog language

Note that there are two ways of connecting wires to the module interface: *by order* and *by name*. If the former way is used then the instance connections are defined by a comma-separated list of wire names, e.g. *inst_1*. Each wire in this list is connected to that port from the module definition, which has the same position in the list of module interfaces. In the latter way each wire is associated with a module port name explicitly, e.g. *inst_co*.

Note that C-style and C++-style comments are accepted in Verilog. A C-style comment is a text between '/*' and '*/'; a C++-style comment is a text between the '//' sequence and the end of the line.

The formal grammar definition for the Structural Verilog language is the following.

```
circuit
    : // Empty
    | circuit module
module
    : "module" NAME port_list_opt ';' module_item_clr "endmodule"
module_item_clr
    : // Empty
    | module_item_clr port_declaration
    | module_item_clr instance_declaration
port_declaration
    : input_declaration
    | output_declaration
```

```
    | inout_declaration
    | reg_declaration
    | wire_declaration
input_declaration
    : "input" range_opt variable_list ';'
output_declaration
    : "output" range_opt variable_list ';'
inout_declaration
    : "inout" range_opt variable_list ';'
reg_declaration
    : "reg" range_opt variable_list ';'
wire_declaration
    : "wire" range_opt variable_list ';'
instance_declaration
    : NAME instance_list ';'
instance_list
    : instance
    | instance_list ',' instance
instance
    : instance_name_opt instance_index_opt '(' connection_list ')'
instance_name_opt
    : // Empty
    | NAME
instance_index_opt
    : // Empty
    | '[' NUMBER ']'
connection_list
    : ordered_connection_list
    | named_connection_list
ordered_connection_list
    : connection
    | ordered_connection_list ',' connection
named_connection_list
    : named_connection
    | named_connection_list ',' named_connection
named_connection
```

```
    : '.'NAME '(' connection ')'
connection
    : port_expression_opt
    | '{' port_list '}'
port_list_opt
    : // Empty
    | '(' port_list ')'
port_list
    : port
    | port_list ',' port
port
    : port_expression_opt
port_expression_opt
    : // Empty
    | port_expression
port_expression
    : port_reference
port_reference
    : NAME port_reference_arg
    | NUMBER
port_reference_arg
    : // Empty
    | '[' NUMBER ']'
    | '[' NUMBER ':' NUMBER ']'
variable_list
    : NAME
    | variable_list ',' NAME
range_opt
    : // Empty
    | range
range
    : '[' NUMBER ':' NUMBER ']'
```

In this grammar the words in bold font are the keywords of the Verilog language. The parts of the lines after the double back slash '`//`' are comments. The capitalised words have special meaning: `NUMBER` is an integer positive number; `NAME` is a string which consists of alphabetic characters,

digits and underscore symbols and starts with either a character or an underscore symbol.

B.3 Usage of the VeriMap tool

```
USAGE:
  verimap [OPTIONS] [INPUT_FILE_NAMES]
FILE OPTIONS:
  -o,  --output OUTPUT_FILE_NAME   - Verilog netlist output file [STDOUT]
  -i,  --include INC_FILE_NAMES    - Verilog include files [STDIN]
  -l,  --library LIB_FILE_NAME     - Verilog library output []
  -g,  --generate GEN_FILE_NAME    - Verilog generated modules [STDOUT]
  -r,  --rules RULES_FILE_NAMES    - rules for gate transformation files []
  -p,  --params PARAMS_FILE_NAMES  - parameters of modules input files []
  -a,  --assignments ASSIGN_FILE_NAMES  - assignments input files []
  -cn, --completion-nets NET_FILE_NAMES - completion nets input files []
    *_FILE_NAMES is a coma separated list of files for reading
TRANSFORMATION OPTIONS:
  -tl, --transformation-level[TL]  - transformation level (TL=[0],1,2,3)
       0 = no circuit transformation, the netlist is re-formatted only
       1 = calculate the optimisation possibilities only
       2 = convert the circuit into dual-rail
       3 = build completion detection logic, add go input and done output
  -cd, --completion-delay[CD]      - delay of completion logic (CD=[0],1...)
  -ct, --clock-toggle              - build a toggle for alternating spacer
  -srw,--single-rail-wrapper       - build a single-rail wrapper
OPTIMISATION OPTIONS:
  -ol, --optimisation-level[OL]    - optimisation level (OL=[0],1,2)
       0 = no optimisation (positive gates)
       1 = optimisation for negative gates or for a parameters
           given by --optimisation-params option
       2 = optimisation of spacer converters distribution
  -op, --optimisation-params PARAM_NAMES - optimise for parameters []
       PARAM_NAMES is a coma separated list of parameter names which are
       specified in the files provided by --params option
  -gd, --gate-delay-param PARAM_NAME - name of the cell delay parameter []
       PARAM_NAME is parameter name which is specified in the files
```

```
      provided by --params option
STRUCTURE OPTIONS:
  -tm, --topmost-module MODULE_NAME - name of the topmost module
  -buf,--buffer                     - buffer SPACER and DONE signals
  -ra, --reset-active[RA]           - reset active level (RA=[0],1)
       0 = active-0 reset, 1 = active-1 reset
  -rst,--reset PORT_NAMES           - names of reset ports [*.rst]
  -clk,--clock PORT_NAMES           - names of clock ports [*.clk]
  -sp0,--spacer-0 PORT_NAMES        - spacer-0 ports [*.*]
  -sp1,--spacer-1 PORT_NAMES        - spacer-1 ports []
       PORT_NAMES is a comma-separated list of MODULE_NAME.PORT_NAME
           MODULE_NAME.* = all ports of the MODULE_NAME module
           *.PORT_NAME = port PORT_NAME of all modules
           *.* = all ports of all modules
           PORT_NAME = port PORT_NAME of the topmost module
STATISTICS OPTIONS:
  -s,  --statistics STAT_FILE_NAME - statistics output file [STDOUT]
  -sl, --statistics-level[SL]      - statistics output level (SL=[0],1,2,3)
       0 = no statistics is collected
       1 = statistics is collected for single-rail circuit only
       2 = statistics is collected for dual-rail circuit only
       3 = statistics is collected for both single- and dual-rail circuits
CONES OPTIONS:
  -c,  --cones CONES_FILE_NAME     - cones output file name [-]
  -cl, --cones-level[CL]           - cones output level (CL=[0],1,2,3)
       0 = table of cones intersection only
       1 = output nets in each cone
       2 = output nets and instances in each cone
       3 = additional comments for cones layers
  -cv, --cones-vertex ITEM_NAMES   - list of vertex items to build the cones
       ITEM_NAMES is a comma-separated list of MODULE_NAME.ITEM_NAME
            ITEM_NAME is either net or instance name
            MODULE_NAME.* = all nets and instances of the MODULE_NAME module
            *.ITEM_NAME = item ITEM_NAME of all modules
            *.* = all nets and instances of all modules
            ITEM_NAME = item ITEM_NAME of the topmost module
```

```
SWITCHING ACTIVITY OPTIONS:
  -w, --wires WIRES_FILE_NAME      - wires switching output file [STDOUT]
  -wl, --wires-level[WL]           - level of details for wires (WL=[0],1,2,3)
       0 = no wires switching analysis
       1 = wires switching is analysed for single-rail circuit only
       2 = wires switching is analysed for dual-rail circuit only
       3 = wires switching is analysed for both single- and dual-rail circuits
  -wp, --wires-pattern PATTERN     - wires output format [always @(?)  c=c+1;]
       Question mark ?  in the PATTERN is replaced by the wires full names
TIMING ANALYSIS OPTIONS:
  -cmd CMD_FILE_NAME               - Pearl timing analysis command file
  -gcf GCF_FILE_NAME               - GCF file for timing analysis
  -pp, --path-possibility[N]       - consider N worst paths (N=[1],2...)
INFORMATION OPTIONS:
  -h, --help                       - print this help only
  -v, --version                    - print version only
```

Appendix C

AES designs

The symmetric block cipher Rijndael [32] was standardised by NIST as the Advanced Encryption Standard (AES) [109] in November 2001 as the successor to DES. The algorithm is a block cipher that encrypts/decrypts blocks of 128, 192, or 256 bits, and uses symmetric keys of 128, 192 or 256 bits. It consists of a sequence of four primitive functions, SubBytes, ShiftRows, MixColumns and AddRoundKey called a round. A round is executed 10, 12 or 14 times depending on the key and plain text lengths. Before the rounds are executed the AddRoundKey function is applied for initialisation plus the last round omits the MixColumns operation. A new key is derived for each round from the previous key.

For decryption the procedure is reversed and inverse versions of the aforementioned functions are applied, excluding AddRoundKey, this has no inverse.

A detailed explanation of each function can be found in [32, 66]. For clarity the SubBytes function performs a non linear transformation using byte substitution tables (Sboxes), each Sbox is a multiplicative inversion in GF(256) followed by an affine transformation.

A brief description of the two architectures follows.

C.1 Open core AES architecture

This design operates on 128 bits and has two separate 'sub-cores' one for encryption and the other for decryption; they share the same type of key generation module [117] and initial permutation module, however separate instances exist inside each sub-core. The core is shown in Figure C.1;

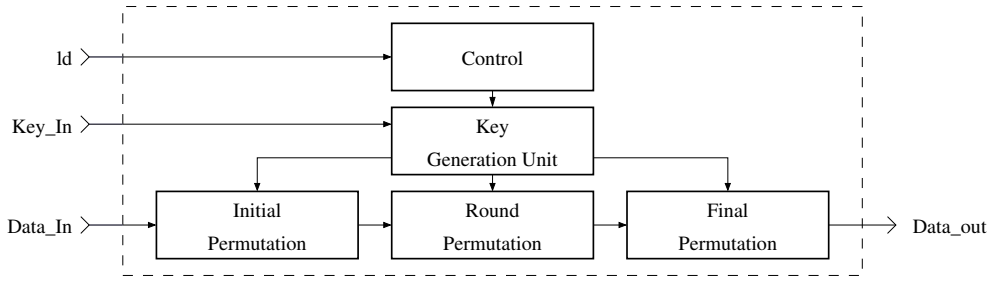


Figure C.1: Open core AES

each sub core has 16 inverse/Sboxes inside the round permutation module. The initial permutation modules simply perform the AddRoundKey function and the round permutation modules loops internally to perform the 10 rounds and the final permutation module performs the last round. For this yields a complete encryption in 12 clock cycles. The decryption core consists of 16 inverse Sboxes these differ from the Sboxes used for encryption. The key reversal buffer stores keys for all the rounds and these are presented to the round permutation module each round in reverse order. Using this principle a complete decryption can be performed in 12 clock cycles. It must be highlighted that since the keys are used in reverse order - the initial key must be first expanded 10 times to get the last key, taking 10 extra clock cycles. In this design the Inv/SubBytes transformations (Sboxes) are hardwired instead of being computed on the fly or stored in a ROM. This can be seen as simply a large decoder. The sub-cores both have 128 pins for plain/cipher text and 128 pins for the key and miscellaneous control pins and logic.

C.2 AES with computable Sboxes architecture

This architecture combines encryption and decryption into one core working on 128 bits. The designs' basis is taken from [66], it was chosen due to its structure namely: it is highly regular (this keeps the layout small), it has short balanced combinational paths, hardware reuse for encryption and decryption which yields a small area and finally it has a 32 pin interface for the data (128 pin for the key) and shared computable Sboxes.

The design consists of a key generation unit, control logic and a data unit incorporating 16 data cells, 4 Sboxes (these perform the Sbox and inverse Sbox unlike the open core design) and a barrel

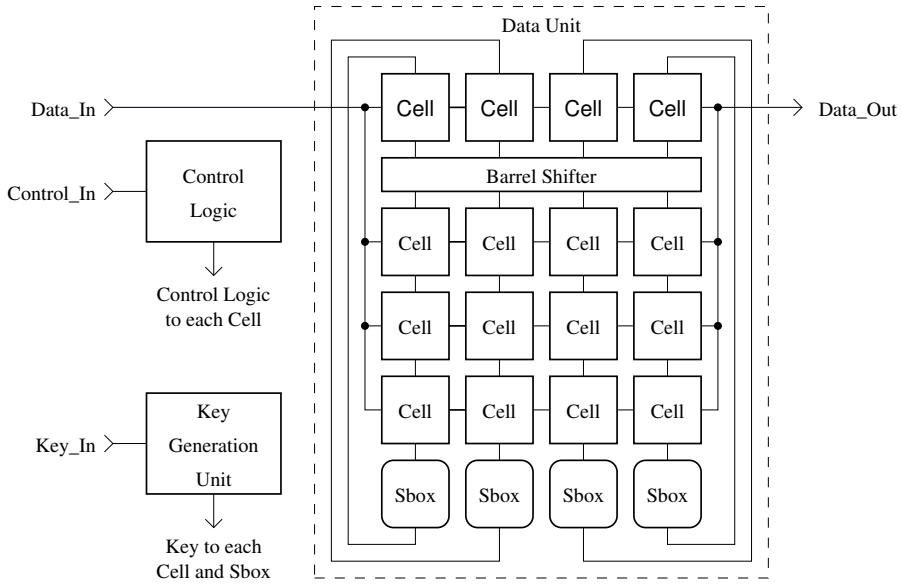


Figure C.2: AES with computable Sboxes

shifter. The core is displayed in Figure C.2, since the core does both encryption and decryption the diagram summarises both.

The data unit can perform any of the AES round functions and uses the key provided by the key unit for the AddRoundKey function. A single data cell comprises of a register, a GF(256) multiplier [66], a bank of XOR gates, and an input selection multiplexer. Additional multiplexers are included to enable the required function to be selected.

The Sboxes are able to perform either the Sbox transformation or the inverse Sbox transformation taking two clock cycles to compute a result due to a two-stage pipeline. Whilst the Sboxes are not used by the data unit (the MixColumns operation) the key generation unit takes advantage of this to generate the next key. The Sbox is computed by reducing the computation to GF(16) and GF(16) arithmetic and then applying the affine transformation as illustrated in [122].

Since the design has a 32 pin interface for the data, four clock cycles are required to clock the plain text, or cipher text into the data unit, and the same number to retrieve the data. After loading, the round functions are selected by the control logic. In total 60 clock cycles are needed for a complete encryption or decryption. As with the other design the input key needs to be expanded to the last key value before any rounds can take place; this takes an extra 20 clock cycles due to the pipelined Sbox. The total number of cycles for encryption or decryption could be reduced to

30 by using 16 Sboxes at the expense of more area.

C.3 AES chip

A circuit presented in this section is designed in collaboration with Julian Murphy, who did layouts of all blocks and floorplaning of the chip.

Two implementations of AES with computed Sboxes are fabricated on this chip in order to compare their resistance to power analysis attacks. The first one is a standard RTL implementation synthesised from a behavioural AES specification, and the other one is a dual-rail implementation obtained from the RTL netlist by using the VeriMap tool kit.

The floorplan of the chip is depicted in Figure C.3. The blocks labelled on the floorplan are the following:

- Block 1 is a single-rail AES core with computable Sboxes capable of 128 bit encryption and decryption. This block was synthesised using Cadence Ambit v4.0 from a behavioural Verilog specification.

- Block 2 is a dual-rail AES core with computable Sboxes capable of secure encryption and decryption, using an alternating-spacer signalling protocol. This dual-rail AES block was obtained by direct mapping from an RTL netlist of the single-rail AES design using the VeriMap tool kit. No additional synthesis was required for this block. This design is about 110% larger than the single-rail AES block.

- Block 3 is an independent design for on-chip sub-picosecond phase alignment.

- Block 4 is an interface logic which multiplexes the input and output pins and selects which block to activate.

The chip was laid out in an AMS 0.35μ four layer process, which works on a 3.3 voltage. The designs use standard cells, no custom transistor sizing or transistor-level implementations of dual-rail flip-flops are used. The design time was four months from initial layout to tape out. After initial layout, the design was simulated digitally and then simulated at the transistor level. Both simulations produced correct results.

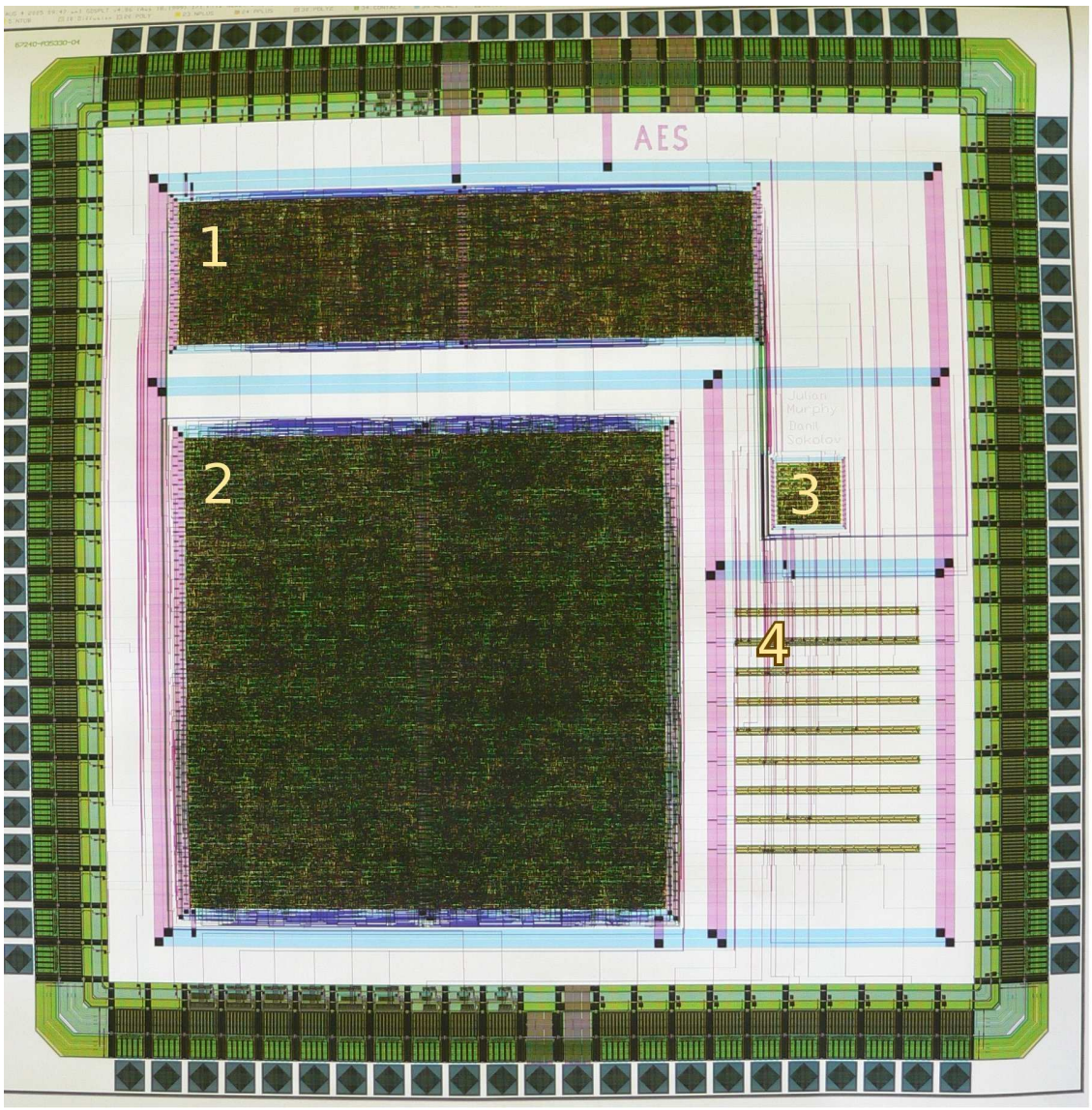


Figure C.3: AES chip floorplan

The chip is produced using the Europractice fabrication facilities. The area of the complete chip is 10.89mm^2. The chip and its package are shown in Figure C.4.The blocks labelled on the circuit floorplan are clearly visible on the chip surface.

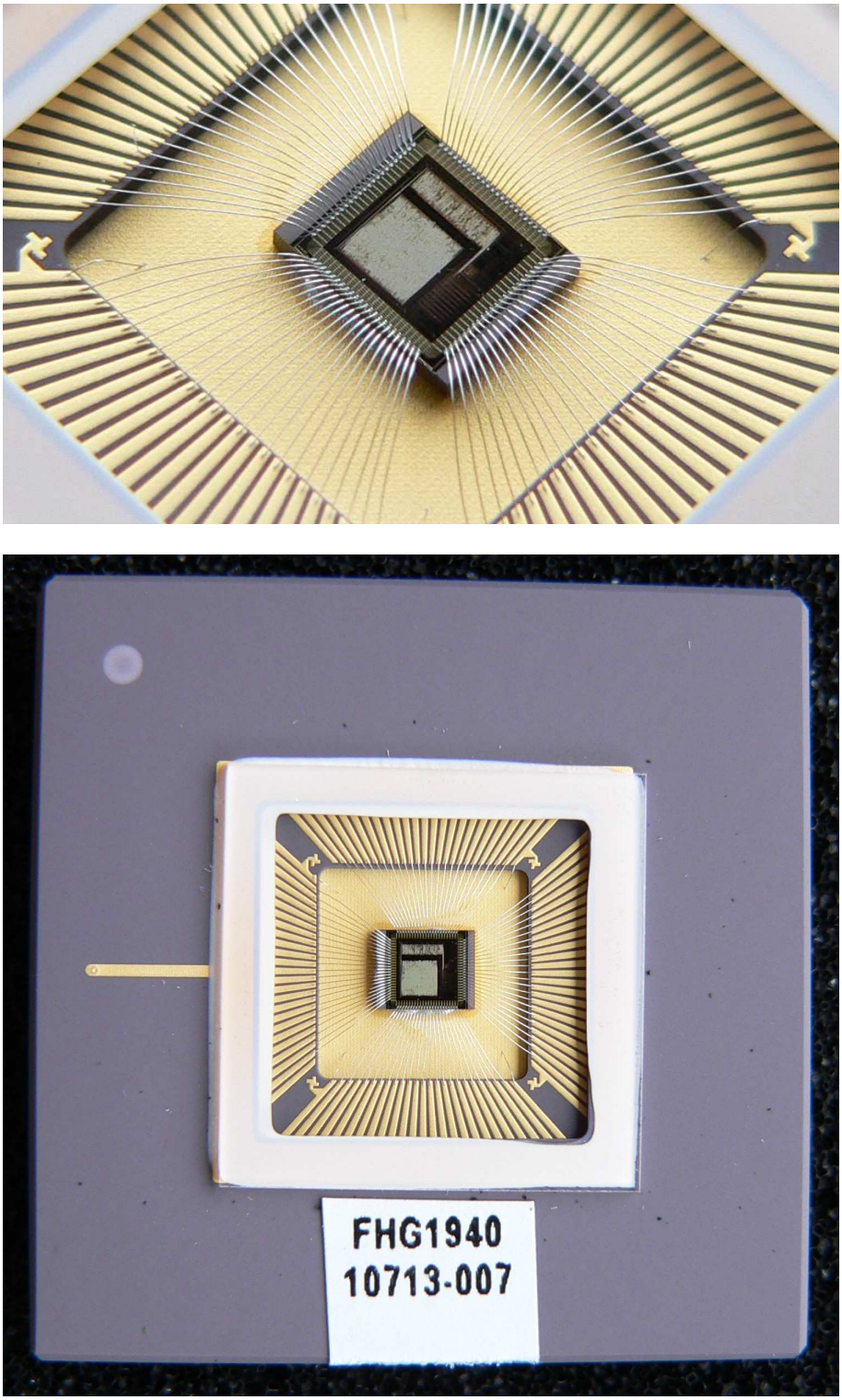


Figure C.4: AES chip package

Bibliography

[1] International technology roadmap for semiconductors: 2003 edition. http://public.itrs.net/Files/2003ITRS/Home.htm, 2003.

[2] John Bainbridge and Steve Furber. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 118–126. IEEE Computer Society Press, March 2001.

[3] John Bainbridge, W.B. Toms, Doug Edwards, and Steve Furber. Delay-insensitive, point-to-point interconnect using M-of-N codes. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 132–140. IEEE Computer Society Press, May 2003.

[4] Andrew Bardsley. *Implementing Balsa handshake circuits*. PhD thesis, Dept. of Computer Science, University of Manchester, 2000.

[5] Andrew Bardsley and Doug Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, April 1997.

[6] Andrew Bardsley and Doug Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000.

[7] Peter A. Beerel. Asynchronous circuits: an increasingly practical design solution. In *Proc. of the International Symposium on Quality Electronic Design (ISQED)*, 2002.

[8] Peter A. Beerel and Teresa H. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conference Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, November 1992.

[9] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[10] Kees van Berkel. *Handshake circuits: an asynchronous architecture for VLSI programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[11] Kees van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.

[12] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[13] Ivan Blunno and Luciano Lavagno. Towards a language-based design flow for asynchronous circuits. In *Proc. International Workshop on Logic Synthesis*, June 1999.

[14] Ivan Blunno and Luciano Lavagno. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 84–92. IEEE Computer Society Press, April 2000.

[15] Frank Burns, Albert Koelmans, and Alex Yakovlev. Synthesis of secure circuits from SystemC. In *Post Graduate Conference, School of EECE, University of Newcastle upon Tyne*, January 2005.

[16] Frank Burns, Delong Shang, Albert Koelmans, and Alex Yakovlev. An asynchronous synthesis toolset using Verilog. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 724–725, February 2004.

[17] Alex Bystrov, Danil Sokolov, and Alex Yakovlev. Balancing power signature in secure systems. In *Proc. UK Asynchronous Forum*, Newcastle upon Tyne, UK, June 2003.

[18] Alex Bystrov and Alex Yakovlev. Asynchronous circuit synthesis by direct mapping: Interfacing to environment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 127–136, Manchester, UK, April 2002. IEEE Computer Society Press.

[19] Luca P. Carloni, Kenneth L. McMillan, Alexandra Saldanha, and Alberto Sagiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *Proc. International Conference Computer-Aided Design (ICCAD)*, pages 309–315, November 1999.

[20] Josep Carmona. *Structural methods for the synthesis of well-formed concurrent specifications*. PhD thesis, Software Department, Universitat Politècnica de Catalunya, March 2004.

[21] Josep Carmona, Jordi Cortadella, and Enric Pastor. A structural encoding technique for the synthesis of asynchronous circuits. In *International Conference Application of Concurrency to System Design*, pages 157–166, June 2001.

[22] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous systems*. PhD thesis, Stanford University, October 1984.

[23] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Proc. Advances in Cryptology (CRYPTO)*, page 398, 1999.

[24] Tiberiu Chelcea, Andrew Bardsley, Doug Edwards, and Steven M. Nowick. A burst-mode oriented back-end for the Balsa synthesis system. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 330–337, March 2002.

[25] Wei-Chun Chou, Peter A. Beerel, and Kenneth Y. Yun. Average-case technology mapping of asynchronous burst-mode circuits. *IEEE Transactions on Computer-Aided Design*, 18(10):1418–1434, October 1999.

[26] Tam-Anh Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, June 1987.

[27] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336. Academic Press, 1967.

[28] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, Spain, November 1996.

[29] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. A region-based theory for state assignment in speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 16(8):793–812, August 1997.

[30] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. *Hardware and Petri nets: application to asynchronous circuit design*, volume 1825 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, June 2000.

[31] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. *Logic synthesis for asynchronous controllers and interfaces*. Springer-Verlag, Berlin, Germany, March 2002.

[32] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES - the Advanced Encryption Standard*. Information security and cryptography. Springer-Verlag, 2002.

[33] Ilana David, Ran Ginosar, and Michael Yoeli. An efficient implementation of boolean functions as self-timed circuits. *IEEE Transactions on Computers*, 41(1):2–11, 1992.

[34] René David. Modular design of asynchronous circuits defined by graphs. *IEEE Transactions on Computers*, 26(8):727–737, August 1977.

[35] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, University of Utah, September 1997.

[36] A. Dindhuc, J.-B. Rigaud, A. Rezzag, A. Sirianni, João Loenardo Fragoso, L. Fesquet, and Marc Renaudin. Tima asynchronous synthesis tools. In *Communication to ACID Worshop*, January 2002.

[37] R. Dobkin, Ran Ginosar, and C. P. Sotiriou. Data synchronization issues in GALS SoCs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 170–179. IEEE Computer Society Press, April 2004.

[38] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[39] Don Edenfeld, Andrew B. Kahng, Mike Rodgers, and Yervant Zorian. 2003 technology roadmap for semiconductors. *Computer*, 37(1):47–56, January 2004.

[40] Doug Edwards and Andrew Bardsley. Balsa: an asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[41] Karl Fant and Scott Brandt. NULL conventional logic: a complete and consistent logic for asynchronous digital circuit synthesis. In *International Conference on Application-specific Systems, Architectures, and Processors*, pages 261–273, 1996.

[42] Marcos Ferretti and Peter A. Beerel. Single-track asynchronous pipeline templates using 1-of-N encoding. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1008–1015, March 2002.

[43] João Loenardo Fragoso, Gilles Sicard, and Marc Renaudin. Power/area tradeoffs in 1-of-M parallel-prefix asynchronous adders. In Jorge Juan Chico and Enrico Macii, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, volume 2799 of *Lecture Notes in Computer Science*, pages 171–180, September 2003.

[44] R. M. Fuhrer, Steven M. Nowick, Michael Theobald, N. K. Jha, B. Lin, and Luis Plana. Minimalist: an environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, July 1999.

[45] R.J. van Glabbeek and Peter W. Weijland. Branching time and abstraction in bisimulation semantics. 43(3):555–600, 1996.

[46] S. Guilley, P. Hoogvorst, Y. Mathieu, R. Pacalet, and J. Provost. CMOS structures suitable for secured hardware. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1414–1415, February 2004.

[47] M. A. Hasan. Power analysis attacks and algorithmic approaches to their countermeasures for Koblitz curve cryptosystems. 50(10):1071, 2001.

[48] Scott Hauck. Asynchronous design methodologies: an overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[49] Matthew C. B. Hennessy and Robin Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309, Noordweijkerhout, Netherland, July 1980. Springer-Verlag.

[50] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[51] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.

[52] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964. Reprinted from J.Franklin Institute, vol. 257, no. 3, pp. 161–190, Mar. 1954, and no. 4, pp. 275–303, Apr. 1954.

[53] Hans M. Jacobson, Prabhakar N. Kudva, Pradip Bose, Peter W. Cook, Stanley E. Schuster, Eric G. Mercer, and Chris J. Myers. Synchronous interlocked pipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 3–12. IEEE Computer Society Press, April 2002.

[54] Axel Jantsch. *Modelling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2004.

[55] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer-Verlag, 1997.

[56] Joep Kessels, Torsten Kramer, Ad Peeters, and Volker Timm. *DESCALE: a Design Experiment for a Smart Card Application consuming Low Energy*. Kluwer Academic Publishers, 2001.

[57] Joep Kessels and Ad Peeters. The Tangram framework: asynchronous circuits for low power. In *Proc. Asia and South Pacific Design Automation Conference*, pages 255–260, February 2001.

[58] Victor Khomenko. *Model checking based on Petri net unfolding prefixes*. PhD thesis, School of Computer Science, University of Newcastle upon Tyne, 2002.

[59] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent hardware: the theory and practice of self-timed design*. Series in Parallel Computing. Wiley-Interscience, John Wiley& Sons, Inc., 1994.

[60] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. Advances in Cryptology (CRYPTO)*, pages 104–113. Springer-Verlag, 1996.

[61] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis: leaking secrets. In *Proc. Advances in Cryptology (CRYPTO)*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[62] Alex Kondratyev and Kelvin Lwin. Design of asynchronous circuits using synchronous CAD tools. *IEEE Design & Test of Computers*, 19(4):107–117, 2002.

[63] Luciano Lavagno and Alberto Sagiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits*. Kluwer Academic Publishers, 1993.

[64] Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, and Alex Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Proc. International Symposium*

on Advanced Research in Asynchronous Circuits and Systems (ASYNC), pages 114–125. IEEE Computer Society Press, April 2000.

[65] Agnes Madalinski, Alex Bystrov, Victor Khomenko, and Alex Yakovlev. Visualization and resolution of coding conflicts in asynchronous circuit design. In *Proc. Design, Automation and Test in Europe (DATE)*. IEEE Computer Society Press, March 2003.

[66] S. Mangard, M. Aigner, and S. Dominikus. A highly regular and scalable AES hardware architecture. *IEEE Transactions on Computers*, 52(4):483–491, 2003.

[67] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[68] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[69] Alain J. Martin. Remarks on low-power advantages of asynchronous circuits. In *Proc. European Solid-State Circuits Conference (ESSCIRC)*, 1998.

[70] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES)*, volume 1965 of *Lecture Notes in Computer Science*, pages 78–92, 2000.

[71] T. Messerges, E. Dabbish, and R. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51(5):541–552, 2002.

[72] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[73] U. Montanari and F. Rossi. Contextual nets. *Acta Informacia*, 32(6):545–596, 1995.

[74] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):68–70, April 1965.

[75] Gordon Moore. No exponential is forever: but "Forever" can be delayed! In *International Solid State Circuits Conference*, volume 1, pages 20–23, 2003.

[76] Simon Moore, P. Anderson, P. Cunningham, R. Mullins, and G. Taylor. Improving smart card security using self-timed circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 211–218. IEEE Computer Society Press, April 2002.

[77] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[78] Tadao Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[79] Jens Muttersbach. *Globally-asynchronous locally-synchronous architectures for VLSI systems*. PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, 2001.

[80] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 52–59. IEEE Computer Society Press, April 2000.

[81] Chris J. Myers. *Asynchronous circuit design*. Wiley-Interscience, John Wiley& Sons, Inc., July 2001.

[82] Chris J. Myers and Teresa H. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[83] John O'Leary and Geoffrey Brown. Synchronous emulation of asynchronous circuits. *IEEE Transactions on Computer-Aided Design*, 16(2):205–209, February 1997.

[84] Enric Pastor. *Structural methods for the synthesis of asynchronous circuits from signal transition graphs*. PhD thesis, Universitat Politècnica de Catalunya, February 1996.

[85] Suhas S. Patil and J. B. Dennis. The description and realization of digital systems. In *Proc. IEEE COMPCON*, pages 223–226, 1972.

[86] Ad Peeters. Support for interface design in Tangram. In Alex Yakovlev and Reinder Nouta, editors, *Asynchronous Interfaces: Tools, Techniques, and Implementations*, pages 57–64, July 2000.

[87] Ad Peeters and Kees van Berkel. Synchronous handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 86–95. IEEE Computer Society Press, March 2001.

[88] Carl Adam Petri. *Kommunikation mit automaten (Communicating with automata)*. PhD thesis, University of Bonn, 1962.

[89] Luis Plana, Peter Riocreux, Andrew Bardsley, Jim Garside, and Steve Temple. SPA - a synthesisable Amulet core for smartcard applications. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 201–210, Manchester, UK, April 2002. IEEE Computer Society Press.

[90] Wolfgang Reisig and Grzegorz Rozenberg. Informal introduction to Petri nets. In *Petri nets*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[91] Leonid Rosenblum and Alex Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.

[92] M. Sacker, A. Brown, P. Wilson, and A. Rushton. A general purpose behavioural asynchronous synthesis system. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 125–134. IEEE Computer Society Press, April 2004.

[93] Hiroshi Saito, Alex Kondratyev, Jordi Cortadella, Luciano Lavagno, and Alex Yakovlev. What is the cost of delay insensitivity? In *Proc. International Conference Computer-Aided Design (ICCAD)*, pages 316–323, November 1999.

[94] H. Saputra, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, R. Brooks, S. Kim, and W. Zhang. Masking the energy behaviour of DES encryption. In *Proc. Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2003. IEEE Computer Society Press.

[95] Charles L. Seitz. System timing. In Carver A. and Mead, editor, *Introduction to VLSI systems*, chapter 7. Addison-Wesley, 1980.

[96] Delong Shang, Frank Burns, Albert Koelmans, Alex Yakovlev, and F. Xia. Asynchronous system synthesis based on direct mapping using VHDL and Petri nets. *IEE Proceedings, Computers and Digital Techniques*, 151(3):209–220, May 2004.

[97] Montek Singh and Michael Theobald. Generalized latency-insensitive systems for GALS architectures. In *Proc. Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Architecture (FMGALS) (within the International Formal Methods Europe Symposium)*, September 2003.

[98] Alexander Smirnov, Alexander Taubin, Mark Karpovsky, and Leonid Rosenblum. Gate transfer level synthesis as an automated approach to fine-grain pipelining. In *Proc. Workshop on Token Based Computing (ToBaCo) and , Satellite Event of the 25-th International conference on application and theory of Petri nets*, June 2004.

[99] Alexander Smirnov, Alexander Taubin, Ming Su, and Mark Karpovsky. An automated fine-grain pipelining using domino style asynchronous library. In *International Conference Application of Concurrency to System Design*, pages 68–76, June 2005.

[100] Douglas Smith. VHDL and Verilog compared and contrasted plus modeled example written in VHDL, Verilog and C. In *Proc. ACM/IEEE Design Automation Conference*, pages 771–776, June 1996.

[101] Danil Sokolov, Frank Burns, Delong Shang, Agnes Madalinski, Alex Bystrov, and Alex Yakovlev. Asynchronous system design flow based on Petri nets. In *University Booth Exhibition in Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2005.

[102] Danil Sokolov, Alex Bystrov, and Alex Yakovlev. Automated design of low-latency asynchronous circuits by direct mapping. In *Postgraduate Research Conference in Electronics, Photonics, Communications and Software*, Nottingham, UK, April 2002.

[103] Danil Sokolov, Alex Bystrov, and Alex Yakovlev. Tools for STG optimisation in the direct mapping of asynchronous circuits. In *Special Interest Group on Design Automation*, Bournemouth, UK, September 2002.

[104] Danil Sokolov, Alex Bystrov, and Alex Yakovlev. STG optimisation in the direct mapping of asynchronous circuits. In *Proc. Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2003. IEEE Computer Society Press.

[105] Danil Sokolov, Julian P. Murphy, Alex Bystrov, and Alex Yakovlev. Improving the security of dual-rail circuits. In *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES)*, August 2004.

[106] Danil Sokolov, Julian P. Murphy, Alex Bystrov, and Alex Yakovlev. Design and analysis of dual-rail circuits for security applications. *IEEE Transactions on Computers*, 54(4):449–460, April 2005.

[107] Danil Sokolov and Alex Yakovlev. Clock-less circuits and system synthesis. *IEE Proceedings, Computers and Digital Techniques*, 152(3):298–316, May 2005.

[108] Jens Sparsø and Steve Furber. *Principles of asynchronous circuit design: a system perspective*. Kluwer Academic Publishers, 2001.

[109] National Institute Of Standards and Technology. The advanced encryption standard (AES), federal information processing standard 197. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[110] Ivan Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. The 1988 Turing award lecture.

[111] Ivan Sutherland and Scott Fairbanks. GasP: a minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 46–53. IEEE Computer Society Press, March 2001.

[112] K. Tiri, M. Akmal, and I. Verbauwhede. A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proc. European Solid-State Circuits Conference (ESSCIRC)*, 2002.

[113] K. Tiri and I. Verbauwhede. A logical level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 246–251, February 2004.

[114] W.B. Toms. *Synthesis of quasi-delay-insensitive datapath circuits*. PhD thesis, Dept. of Computer Science, University of Manchester, 2004.

[115] Jan Tijmen Udding. *Classification and composition of delay-insensitive circuits*. PhD thesis, Eindhoven University of Technology, 1984.

[116] S. H. Unger. *Asynchronous sequential switching circuits*. Wiley-Interscience, John Wiley& Sons, Inc., New York, USA, 1969.

[117] Rudolf Usselmann. AES Rijndael IP core. Open cores (http://www.opencores.org/), 2004.

[118] Victor Varshavsky. *Self-timed control of concurrent processes: the design of aperiodic logical circuits in computers and discrete systems*. Kluwer Academic Publishers, 1990.

[119] Victor Varshavsky and V. Marakhovsky. Asynchronous control device design by net model behavior simulation. In J. Billington and Wolfgang Reisig, editors, *Application and Theory of Petri Nets*, volume 1091 of *Lecture Notes in Computer Science*, pages 497–515. Springer-Verlag, June 1996.

[120] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. C to asynchronous dataflow circuits: an end-to-end toolflow. In *International Workshop on Logic Synthesis*, June 2004.

[121] Walter Vogler and Ralf Wollowski. Decomposition in asynchronous circuit design. In Jordi Cortadella, Alex Yakovlev, and Grzegorz Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *Lecture Notes in Computer Science*, pages 152–190. Springer-Verlag, 2002.

[122] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. An ASIC implementation of AES SBoxes. *Proc. RSA*, 2271:67–78, 2002.

[123] Alex Yakovlev, Albert Koelmans, and Luciano Lavagno. High-level modeling and design of asynchronous interface logic. *IEEE Design & Test of Computers*, 12(1):32–40, 1995.

[124] T. Yoneda, H. Onda, and Chris J. Myers. Synthesis of speed independent circuits based on decomposition. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 135–145. IEEE Computer Society Press, April 2004.

[125] Zhong Chuan Yu, Steve Furber, and Luis Plana. An investigation into the security of self-timed circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 206–215. IEEE Computer Society Press, May 2003.

[126] Kenneth Y. Yun and Ryan P. Donohue. Pausible clocking: a first step toward heterogeneous systems. In *Proc. International Conference Computer Design (ICCD)*, October 1996.

Index