Efficient Algorithms for the Design of Asynchronous Control Circuits

Michael Theobald

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2002

Efficient Algorithms for the Design of Asynchronous Control Circuits

Michael Theobald

Advisor: Steven M. Nowick

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2002

ABSTRACT

Efficient Algorithms for the Design of Asynchronous Control Circuits

Michael Theobald

Asynchronous (or "clock-less") digital circuit design has received much attention over the past few years, including its introduction into consumer products. One major bottleneck to the further advancement of clock-less design is the lack of optimizing CAD (computer-aided design) algorithms and tools. In synchronous design, CAD packages have been crucial to the advancement of the microelectronics industry. In fact, automated methods seem to be even more crucial for asynchronous design, which is widely considered as being much more error-prone.

This thesis proposes several new efficient CAD techniques for the design of asynchronous control circuits. The contributions include: *(i)* two new and very efficient algorithms for hazard-free two-level logic minimization, including a heuristic algorithm, ESPRESSO-HF, and an exact algorithm based on implicit data structures, IMPYMIN; and *(ii)* a new synthesis and optimization method for large-scale asynchronous systems, which starts from a Control-Dataflow Graph (CDFG), and produces highly-optimized distributed control.

As a case study, this latter method is applied to a *differential equation solver*; the resulting synthesized circuit is comparable in quality to a highly-optimized manual design.

Contents

List of Figures

Acknowledgments

Thank you to the many people who have made my stay at Columbia an educational and memorable journey.

I would like to thank my advisor Steven Nowick for inviting me to do research at Columbia and guiding me throughout my stay. Steve's expert knowledge of asynchronous circuits has been a quality source for my research. I am particularly thankful to Steve for always being open to and supportive of exploring new research ideas. I have also benefited from his insight and many suggestions on presenting my work. Thank you, Steve.

The other members of my dissertation committee — Stephen Unger, Stephen Edwards, Prabhakar Kudva, and Charles Zukowski — have all given me insightful comments on my research and this thesis. I would also like to thank them for accommodating me on such a tight schedule.

Stephen Unger has been a constant source of invaluable insight during my entire stay at Columbia. I have always enjoyed working and interacting with Steve during our research seminars, as his teaching assistant, and when we co-authored papers together with his former student John Cheng. Steve's positive attitude and smiles have often made my day.

I am thankful to the members of my research group at Columbia: Montek Singh, Tibi Chelcea, Luis Plana, Fu-Chiung (John) Cheng and Bob Fuhrer. Stimulating discussions with all of them have had a positive impact on my research. Tibi, thank you for your great friendship. Montek and I were office-mates and our friendship was by no means confined to the office. Montek, thank you for your invaluable friendship.

Without Dean Zvi Galil my studies at Columbia would never have happened because he put me in touch with my advisor Steve Nowick, for which I'm very grateful. I have also truly enjoyed Zvi's lectures on algorithms as well as all the research and social events he has organized to make the Computer Science department a great place.

I have enjoyed two incredible internships with Philips Research in Eindhoven, The Netherlands, and Sun Microsystems Labs in Mountain View, California.

I would like to thank Ad Peeters, Kees van Berkel, and Joep Kessels for making my stay at Philips a pleasant and fruitful one. The internship gave me precious insight into real-world and commercial asynchronous designs. Ad was a superb coach, and I enjoyed the many interesting discussions about research and non-research related issues. I will never forget Ad's always-positive attitude which has inspired me tremendously.

I would like to thank Ian Jones and Ivan Sutherland for inviting me to spend three equally terrific months in the Asynchronous Design Group at Sun Microsystems Labs. Ian, as my direct supervisor, created a great working atmosphere; collaborating with him was a pleasure. Ivan, as the group leader, always made sure I felt comfortable and also taught me many interesting lessons from his long

circuit design experience. I enjoyed learning from Bill Coates and Jo Ebergen and my friendship with Tarik Ono-Tesfaye.

This thesis would never have happened if my former research advisor, Bernd Becker, at Johann Wolfgang Goethe-Universität in Frankfurt, Germany, had not laid the foundation and introduced me to logic synthesis in his courses. Bernd invited me to participate in his ongoing research efforts on decision diagrams, which lead to my thesis there. Bernd, thank you for your support over the years.

Olivier Coudert of Monterey Design Systems helped me tremendously with the implementation of my algorithms by generously providing some of his tools and even writing interfaces so that I could use them. Thank you very much, Olivier!

Thank you to the researchers who contributed to my understanding of asynchronous circuit design: Jordi Cortadella at the Technical University of Catalonia, Spain; Mike Kishinevsky at Intel Corporation; Michel Berkelaar and Jeroen Rutten at TU Eindhoven and Magma Design Automation; Hans Jacobson at the University of Utah and IBM; and Peter Beerel at the University of Southern California.

I am indebted to a number of friends who have been a big part of my life at Columbia: Simon Baker, Tobias Höllerer, Anton Nikolaev, Kazi Zaman, and all the members of the Columbia Sockets intramural soccer team. Go Sockets!

I would like to especially thank Lee Uehara for her continued support and love, Bernard and Maximilian, Thorsten and my parents.

Several grants have made this research possible: NSF RIA Grant MIP-9308810, NSF CAREER Award MIP-9508810, NSF Grant CCR-97-34803, NSF Award CCR-9988241, NSF ITR Award NSF-CCR-0086036; an Alfred P. Sloan Research Fellowship; a gift from Sun Microsystems.

Chapter 1

Introduction

The vast majority of digital circuit design today is based on a synchronous, or clocked, approach. In this case, the total system is designed as the composition of one or more subsystems where each subsystem can be thought of as a clocked state machine, and the subsystem changes from one state to the next on the edges of a regular clock. Intuitively, the clock forces synchronous systems to behave in a discrete manner, which often greatly simplifies the task of designing a correct circuit.

In contrast, asynchronous circuit design does not follow this methodology. There is no clock that controls the timing of state changes. Subsystems exchange information at mutually-negotiated times with no external timing regulation.

1.1 Motivation

1.1.1 Why Asynchronous Design?

While the synchronous approach to digital circuit design has led to dramatic progress in the advancement of modern computers, asynchronous circuits have some unique characteristics that can be exploited to advantage:

- Asynchronous circuits can be faster, in some applications.

  The clock period of synchronous circuits must be long enough to ensure that all possible computations have completed, yielding worst-case performance. Asynchronous circuits can sense when a computation has actually completed, and therefore exhibit average-case performance [30, 45].

- Asynchronous circuits can consume less power.

  In many synchronous circuits, clock distribution dominates the power consumption. In contrast, asynchronous circuits have no clock, and consume power only when and where active [9, 30, 45].

- Asynchronous circuits can be beneficial in minimizing Electro-Magnetic Interference (EMI).

  Consumed energy of a system is partially radiated into space, possibly affecting other electronic equipment. For this reason, notebooks have to be turned off during plane take-offs. Since clocks in synchronous circuits cause periodic activity, the radiation spectra shows peaks at the clock frequency and at multiples thereof (the so-called harmonics). In contrast, radiation spectra of asynchronous circuits exhibit more spread [9, 84].

- Asynchronous circuits are inherently more suitable for designing mixed-timing interfaces.

  Asynchronous interfaces are very common, e.g. keyboards or memories. A clocked system that has an asynchronous external input may fail as it is subject to metastability. A metastable state is an unstable equilibrium state which a system can remain in for an unbounded amount of time. In addition, the use of synchronous mixed-timing domains is becoming more widespread, and the design of asynchronous interface circuits is an important application area, facilitating component reuse and the development of easily-scalable heterogeneous systems.

1.1.2 Recent Advancement in Asynchronous Design

Asynchronous design has been the focus of much recent research activity [30, 7]. In fact, recent research has demonstrated the feasibility and some of the advantages of asynchronous design.

- Philips has introduced asynchronous 80C51 microcontrollers [44, 7] into consumer electronics, such as pagers and cell phones. The controller shows a power advantage of a factor of four compared to a synchronous implementation. However, the key reason for choosing an asynchronous microcontroller over a synchronous one was its significantly better EMI properties.

- The University of Manchester introduced AMULET2e, an asynchronous embedded controller [43]. AMULET2e's performance and power-efficiency are competitive with the industry-leading clocked ARM design. In addition, it

has some unique advantages, such as instantaneous restart from an inactive state, which is not possible in existing synchronous designs. More recently, AMULET3 [42] has been introduced, and is planned to be used in smartcards.

- A number of major companies are designing experimental asynchronous chips, e.g., Intel's high-speed instruction-length decoder [87], Sun Microsystem's high-speed pipelines [20], and IBM's recent digital FIR filter [94].

- Several new start-ups have recently been founded, such as Fulcrum Microsystems, Theseus Logic and Self-Timed Solutions.

A number of additional large-scale asynchronous circuits have been developed, including microprocessors [72, 95, 61, 71], a standby circuit for a low-power pager [48], a low-power DCC Error Corrector [104] and a Huffman decoder [6].

1.1.3 Present and Future of Asynchronous Design

Although synchronous design is likely to dominate for the foreseeable future, the above-mentioned recent advancements clearly indicate that asynchronous circuits can be expected to play an increasing role in digital circuit design over the next years.

However, one major bottleneck to the success of asynchronous methods is the development of optimizing CAD tools. In synchronous design, CAD packages have been critical to the advancement of modern digital design. The design of a complete chip can now be synthesized from a high-level behavioral description with minimal manual intervention.

CAD tools are even more crucial for the advancement of asynchronous design technology, since asynchronous design is, in a sense, more subtle. The main reason for this perception is that the operational domain of asynchronous circuits is no longer a discrete one, which leads to additional design challenges. For example, in synchronous circuits glitches on wires during a clock cycle are no problem, as long as the signal is stable at the next clock edge. In contrast, in asynchronous circuits any glitch may be misinterpreted as a real change and therefore cause malfunction. Thus, the elimination of hazards [103] (i.e., the potential for glitching) is an additional key challenge in all asynchronous systems.

While some impressive progress has recently been made in developing asynchronous CAD tools, much more needs to be done. In fact, asynchronous CAD tools are still more than dwarfed by their synchronous counterparts, both in quantity and sophistication. While there are major synchronous CAD companies like SYNOPSYS, MENTOR, and CADENCE, most of the asynchronous CAD is developed in a few research universities and laboratories.

A key limitation of current asynchronous CAD tools is the lack of techniques for systematic *design-space exploration* and *optimization* of large-scale asynchronous systems. For large-scale asynchronous systems, two design approaches are now widely-used: (i) *manual design,* and (ii) use of *automated syntax-directed translation.* Manual design allows a number of aggressive optimizations, but is cumbersome, slow and error-prone, and it does not provide systematic and automated exploration of the design space. For example, the Intel asynchronous instruction-length decoder chip [87] took over two years to complete, using a combination of manual techniques and academic synthesis tools for designing individual controllers.

The new contribution of this thesis is a set of approaches for the automated synthesis and optimization of large-scale asynchronous systems. At a lower level of design, new and highly-efficient algorithms for finding optimal hazard-free logic implementations are introduced. At a higher level of design, a new approach is proposed for the synthesis and optimization of distributed control of entire asynchronous systems, from high-level specifications. It is the first such proposed approach that facilitates wide-ranging *design-space exploration*.

1.2 Asynchronous Design

To better explain the contributions of the thesis, this section first gives a quick overview of the state-of-the-art synthesis methods for asynchronous controllers and asynchronous systems.

A number of previous methods have been developed for designing asynchronous controllers and systems. Most of these methods differ in fundamental aspects (see [30, 45] for good overviews).

- Specification style

  Popular specification styles include *state-based models* (see e.g. [79]), *event-based models* such as Petri-nets [24], *programming language models* augmented with constructs to express parallelism [13, 8], *Control-Data Flow Graphs*, and *Hardware Description Language (HDL) models*.

- Synthesis target

  Some methods target monolithic finite-state-machine-like implementations

(see e.g. [79]). Others target distributed components that communicate with each other [8].

- Delay model

  While some synthesized circuits work correctly for any delay assignment to gates and wires, others assume that wire delays are negligible. Many different delay models exist [103].

- Interaction with environment

  Some methods allow the environment to respond to a circuit's outputs as soon as they become available [23]. Other methods impose timing constraints on how fast the environment may respond [79, 103].

1.2.1 Synthesis of Single Controllers

Several existing approaches to synthesizing individual controllers are now reviewed. Much of the recent work on single asynchronous controllers has focused on *Petri Net-based methods* [58, 4, 50, 19, 107, 23, 24], and *burst-mode methods* [79, 29, 76, 111, 54, 90, 38]. In this section, these two classes are briefly reviewed. A more detailed introduction to burst-mode methods, which are the main focus of the thesis, is included in the next chapter.

1.2.1.1 Graph-Based Methods

Graph-based methods view the behavior of asynchronous systems as a partially-ordered sequence of events, and their specifications are often based on Petri Nets [68]. Petri Nets can describe both concurrency and choice of events.

A Petri Net is a directed bipartite graph, where the two kinds of vertices are called *places* and *transitions*. Initially, certain places are assigned so-called *tokens* to represent the state of the system. A place that is marked with a token can be viewed as a condition that is true. A transition may *fire* if all its preceding conditions are true, i.e. if all the preceding places have tokens. Intuitively, firing corresponds to an action (or state change of the system). When a transition fires, it removes one token from each preceding place and places one token in each successor place. Thus, at any time, the assignment of tokens to places describes the current state of the systems, and also describes which events (actions) may take place (or are enabled).

Petri Net- and other graph-based methods [58, 4, 50, 19, 107, 23, 24] typically synthesize circuits to work correctly regardless of gate delays (*speed-independent delay model*) [30], and the environment is allowed to respond to the circuit's outputs without timing constraints (*input/output mode*) [30].

1.2.1.2 FSM-Based Methods

Most of the new algorithms in this thesis are based on so-called *burst-mode methods* [79, 29, 76, 111, 54, 90, 38]. Burst-mode methods start from a state-based specification. State changes occur after a set of inputs has changed (input burst), and are followed by a corresponding change of output signals (output burst). The synthesis target is a finite-state-machine model, whose combinational logic works correctly regardless of gate and wire delays, but the correct sequential operation depends on timing constraints. In this case, the environment must wait for a circuit to stabilize before responding with new inputs. A more detailed introduction to

burst-mode specifications and implementations is given in the next chapter.

Burst-mode synthesis typically consists of three steps. *State minimization* merges states with the aim of reducing the complexity of the implementation. *State assignment* assigns a unique binary code to each of the reduced states. Finally, *logic optimization* derives a logic implementation for each of the primary- and next-state outputs. A recent comprehensive CAD package for burst-mode synthesis, called MINIMALIST [38, 39, 41], includes many sophisticated algorithms for each of the steps.

Burst-mode methods have recently been applied to several large and realistic design examples, including an experimental low-power infrared communications chip at HP-Labs and Stanford [60], a second-level cache-controller [77], a SCSI controller [108], a differential equation solver [109], and an experimental instruction-length decoder at Intel [18].

1.2.2 Synthesis of Large-Scale Systems

Asynchronous synthesis methods for large-scale systems can be grouped into different categories depending on how the system to be synthesized is specified. There are four categories: *Translation-based methods*, *Petri net-based methods*, *HDL-based methods*, and *Control-Data Flow Graph methods*.

1.2.2.1 Translation-Based Methods

Translation-based methods [62, 13, 8, 9, 85] view an asynchronous circuit as a collection of communicating processes, and specify each process in notations based on concurrent programming languages.

The most popular translation-based method is TANGRAM [8, 9, 85], which has been developed by Philips to facilitate the design of entire systems. TANGRAM is similar to traditional languages like Pascal or C, with the additional two features that statements may execute in parallel, and processes may communicate along fixed channels. TANGRAM programs are translated into circuits in a syntax-directed way. This means each language construct is mapped directly to a corresponding sub-circuit. Each sub-circuit can be considered as a process that communicates with other processes (sub-circuits). The synthesized circuits are so-called quasi-delay-insensitive [30], i.e. operate correctly independent of the delays in gates and wires, but assume negligible skew in forking wires.

TANGRAM has been successfully employed to design and fabricate several low-power chips, such as an asynchronous low-power 80C51 microcontroller [44], a power-efficient error corrector for the Digital Compact Cassette (DCC) player [104], and a standby circuit for a pager decoder [48]. Philips has introduced Tangram-compiled asynchronous 80C51 microcontrollers [44, 7] into consumer electronics, such as pagers and cell phones.

1.2.2.2 Petri Net-Based Methods

Synthesis methods for large-scale systems based on Petri Nets, such as ACK [54], derive a circuit implementation by first decomposing the specification into synthesizable parts, and then apply the synthesis methods for individual asynchronous controllers presented in the previous section.

1.2.2.3 HDL-Based Methods

Finally, two approaches based on Hardware Description Languages have been introduced.

Blunno and Lavagno [10] have presented a method which is based on Verilog. The method restricts specifications to a so-called asynchronous synthesizable subset. The chosen subset supports in a behavioral specification style such control constructs as sequencing, concurrency, and non-deterministic choice. The output of the method are Petri Nets which can be synthesized using the methods presented in the previous section on single asynchronous controllers.

Ligthart et al. [59] have also presented means of reusing commercial HDL synthesis tools, in a recent startup company, where so-called Null Convention Logic is used at the gate-level.

1.2.2.4 Control-Data-Flow-Graph-Based Methods

Two synthesis approaches have been presented that are based on (Control-)Data Flow Graphs.

Cortadella and Badia [21] have proposed a synthesis method that starts from Data Flow Graphs and synthesizes the control unit in such a way that each datapath block is controlled by a dedicated sub-controller.

Kim et al. [49] have proposed a synthesis method from Control-Data Flow Graphs Their approach subdivides sub-controllers for each datapath block even further, assigning a sub-sub-controller to each of the processes bound to a functional unit.

Both of these approaches are strictly deterministic and "template-based":

they do not include options for design-space exploration.

1.3 Thesis Contributions

In this thesis, several new efficient CAD techniques are proposed for the design of asynchronous control circuits. The contributions include *(i)* two new and very efficient algorithms for hazard-free two-level logic minimization, and *(ii)* a new synthesis and optimization method for distributed control from Control-Data-Flow-Graphs. The first contribution is targeted to optimizing logic for individual controllers. The second contribution is targeted to optimizing large-scale asynchronous systems.

The research for this dissertation has led to several publications [101, 100, 38, 39, 102, 98, 99][1].

Algorithms for Hazard-Free Two-Level Logic Minimization

The avoidance of hazards is key for any asynchronous system, and hazard-free logic minimization is an important step in many asynchronous CAD tools. However, none of the existing minimizers for hazard-free two-level logic minimization can synthesize very large circuits.

The first contribution of the thesis is two very efficient two-level logic minimizers for hazard-free logic minimization: Espresso-HF and Impymin.

Espresso-HF is an algorithm to solve the *heuristic* hazard-free two-level logic minimization problem. The method is heuristic solely in terms of the cardi-

[1]This thesis does not include my research on other aspects of asynchronous design such as synthesis techniques for low-power [75, 80], and the design of delay-insensitive adders [17, 16].

nality of solution. In all cases, it guarantees a hazard-free solution, and obtains near-optimum solutions. The algorithm is based on ESPRESSO-II [88, 35], but with a number of significant modifications to handle hazard-freedom constraints. ESPRESSO-HF also includes a new and efficient algorithm to check for existence of a hazard-free solution, without generating all prime implicants. It is the first complete hazard-free two-level minimizer to be based on the synchronous ESPRESSO-II framework.

IMPYMIN is an algorithm to solve the *exact* hazard-free two-level logic minimization problem. The algorithm obtains its efficiency by *(i)* employing implicit set representations to simultaneously manipulate a large number of objects such as prime implicants (rather than each object separately), and by *(ii)* mapping (and solving) the optimization problem into a higher-dimensional Boolean space. The former involves the use of data structures such as BDDs [15] and zero-suppressed BDDs [67]. The latter is based on a new theoretical approach to coping with hazard-freedom constraints: the generation of so-called dynamic-hazard-free prime implicants is reformulated as a *synchronous* prime implicant generation problem. This is achieved by incorporating hazard-freedom constraints within a synchronous function by adding new auxiliary variables. The formulation allows the use of existing synchronous tools to solve critical sub-steps of the asynchronous two-level synthesis problem.

Both ESPRESSO-HF and IMPYMIN can solve all currently available examples, including examples that have never been previously solved. For examples that can be solved by the currently fastest minimizer HFMIN [37, 41], the two new minimizers are typically several orders of magnitude faster. In particular, IMPYMIN can find

a minimum-size cover for all benchmark examples in less than 813 seconds, and ESPRESSO-HF can find very good covers — at most 3% larger than a minimum-size cover — in less than 105 seconds.

ESPRESSO-HF and IMPYMIN are somewhat orthogonal. On the one hand ESPRESSO-HF is typically faster than IMPYMIN. On the other hand, IMPYMIN computes a cover of minimum size, whereas ESPRESSO-HF is not guaranteed to find a minimum cover but typically does find a cover of very good quality.

Synthesis and Optimization for Distributed Control from Control-Data-Flow-Graphs

Most existing approaches for synthesis of large-scale asynchronous systems are deterministic and "template-based". Interestingly, some efficient manual designs have been presented to which none of these methods has access. Thus, there is still a serious lack of approaches providing systematic design-space exploration.

The second contribution of this thesis is a new approach for the automated synthesis and optimization of large-scale asynchronous systems. In particular, this thesis is the first to introduce, formalize and automate a wide-ranging and powerful set of transformations, which can be used for the optimization of asynchronous distributed control. Unlike previous approaches, these new transforms can be applied in a systematic way to explore the design space and find optimal distributed controller implementations[2].

The proposed method starts with a given scheduled and resource-bounded

[2]These transforms, while at a much higher level of synthesis, are loosely analogous to the powerful transforms of SIS (collapse, extract, etc.) used for design-space exploration in multi-level logic synthesis.

Control-Data Flow Graph (CDFG) [66]. *Global transforms* are first applied to the entire CDFG; unoptimized controllers are then extracted; and, finally, *local transforms* are then applied to the individual controllers. The result is a highly-optimized set of interacting distributed controllers.

The transforms include aggressive timing- and area-oriented optimizations such as: global communication channel multiplexing and symmetrization; loop parallelism; introduction of global "relative timing"-based simplification; multiplexor pre-selection; sharing of local signals; and the removal of unnecessary handshaking wires. Several of these optimizations have not been previously formalized or provided by any other existing asynchronous CAD tool, or else in only a limited way. For example, Kim et al.'s recent approach [49] is also based on CDFGs, however their method does not perform design space exploration, and is limited to handling less concurrent specifications than the new proposed approach.

Finally, as a detailed case study, the transformations are applied to the well-known *differential equation solver* high-level synthesis benchmark [109, 66]. A highly-optimized asynchronous implementation by Yun et al. [109] was manually designed, using a number of aggressive timing- and area-based optimizations. Such an implementation cannot be obtained using other existing CAD tools. We demonstrate that a very similar optimized design can be simply and automatically derived through systematic application of the new transformations.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 gives background on fundamentals of asynchronous systems, asynchronous control specifications and implemen-

tations, and synchronous and asynchronous logic synthesis. Chapter 3 describes the ESPRESSO-HF algorithm for heuristic 2-level hazard-free logic minimization. Chapter 4 introduces a new approach to exact 2-level hazard-free logic minimization where hazard-freedom constraints are captured by a constructed synchronous function. Based on this result, a new implicit minimizer IMPYMIN is introduced. This chapter also includes a comparison of our two new tools with the state-of-the art synchronous counterparts. Chapter 5 presents a new approach for the automated synthesis and optimization of large-scale asynchronous systems. Finally, Chapter 6 presents conclusions and future work.

Chapter 2

Background

The purpose of this chapter is to review background material that is relevant for the later chapters of the thesis.

An asynchronous system can be regarded as a distributed set of interacting components that communicate with each other. The set of components can typically be partitioned into control and datapath components. Section 2.1 surveys basic issues in asynchronous system design such as asynchronous communication protocols between components, as well as asynchronous datapath fundamentals. After this system-level view, Section 2.2 then focuses in depth on asynchronous control components. In particular, an introduction to finite-state machine-based control specifications and implementations is given. Section 2.3 and Section 2.4 then aim at an even lower level, the logic level. First, Section 2.3 reviews basic ideas and concepts of combinational logic synthesis, including fundamental definitions, two-level logic minimization, and decision diagrams. Then, Section 2.4 presents the main ideas of asynchronous combinational logic synthesis that are used to implement control components.

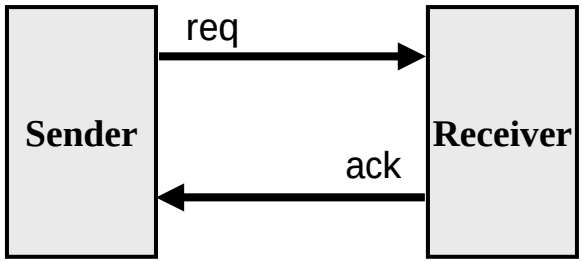


Figure 2.1: Asynchronous Communication

2.1 Asynchronous Systems

2.1.1 Introduction

An asynchronous system typically consists of a number of components, or modules, that communicate with each other. These components, in turn, are typically divided into control and datapath components. The following presentation first reviews a number of common asynchronous communication protocols. Then, fundamentals of designing asynchronous datapath components are discussed, including data encoding schemes and completion generation. (Asynchronous control components will be discussed in detail in Section 2.2.)

2.1.2 Asynchronous Control Communication

Typical asynchronous communication protocols are based on a *handshake protocol* between a sender and a receiver, as indicated in Figure 2.1. Such a protocol involves *requests* from the sender to the receiver to initiate an action, and corresponding *acknowledgments* from the receiver to the sender to indicate completion of the action.

There are several ways of encoding request and acknowledgments on wires.

The most common ones are *2-phase*, *4-phase*, and *Pulse-Mode*, which will be discussed in detail below. Other protocols can be found in [30, 47]. For purposes of this thesis research, the asynchronous systems presented in Chapter 5 will use both 4-phase and 2-phase protocols, where the 2-phase protocol will be further optimized (i.e., by removing explicit *ack* wires, to be discussed). Pulse-Mode is an interesting alternative that has not been explored yet but would fit in very well with the proposed approach. While the focus in this subsection is on control communication, i.e. synchronization, the approaches can be extended to communicating data by using the data encoding schemes of Section 2.1.3.1.

2.1.2.1 2-Phase

2-phase handshaking [97, 13] consists of two signal transitions: a *request* that starts the handshake and an acknowledgment that completes it. An example of a 2-phase handshake is the sequence: $req+, ack+$. The next handshake between these modules would then be $req-, ack-$. The third handshake would then again be $req+, ack+$, and so on.

An advantage of 2-phase handshaking is that only two signal transitions are involved. A disadvantage is that, in practice, more complex logic implementations may be required to enable up and down-going transitions in alternate handshakes.

2.1.2.2 4-Phase

4-phase handshaking [8] consists of four signal transitions: two requests and two acknowledgments. The transitions are ordered in the sequence: $req+, ack+, req-, ack-$. The next handshake would again consist of the same four transitions.

An advantage of 4-phase handshaking is that after a complete handshake the wires are all reset to the same initial state, and thus, in practice, the protocol requires a simpler logic implementation than 2-phase handshaking. A disadvantage is that it involves more transitions to complete a communication than 2-phase handshaking. However, a number of techniques have been proposed to overlap the return-to-zero transitions with other useful actions [62].

2.1.2.3 Pulse-Mode

Pulse-Mode handshaking [86] typically consists of two pulses: one request and one acknowledgment. Thus, it combines advantages of both 2-phase and 4-phase handshaking, since it only has two events per cycle (not four) and yet wires are also reset to their original state. However, pulse-mode circuits must be designed with care to ensure correctness. For example, it must be guaranteed that pulses always maintain a certain minimum pulse width since otherwise a receiver may *overlook* a pulse.

2.1.3 Asynchronous Datapath

When modules communicate data in addition to control signals, techniques must be used to encode the data. Several encoding styles exist. This section introduces two common styles, *dual-rail* and *single-rail*, and then gives an overview of alternative styles. In addition, in some data encoding styles it is necessary to generate a *completion signal* to indicate the validity of the data. Thus, in the following, first data encoding styles are discussed, and then techniques for completion signal generation are surveyed.

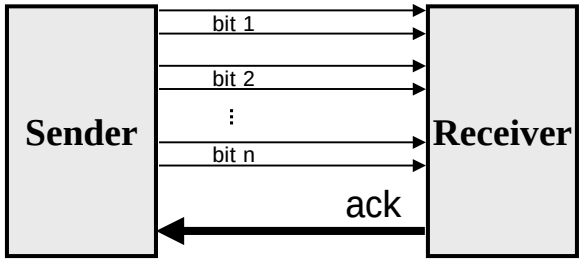


Figure 2.2: Dual-Rail Data Communication

2.1.3.1 Data Encoding

Dual Rail

Dual-rail encoding [30] involves two wires for each bit, as shown in Figure 2.2. The code 01 on the two wires represents the data value *1* and the code 10 represents the data value *0*. Code 00 is used to separate data values and called a *spacer*. Code 11 is not used.

In dual-rail encoding no separate request wire from the sender to the receiver is usually necessary; however, an acknowledge wire from receiver to sender is typically required.

Operation between the sender and the receiver works as follows. Initially, the sender has invalid data, and hence only spacer codes (i.e. 00) appear on the wires. Next, the sender changes the data wires from spacers to the codes for the data bits to be sent. When the receiver has detected that all data bits have been received by checking that no wire pair represents a spacer, it sends an acknowledgment to the sender (typically $ack+$). The sender now resets the wire pairs to spacers. Once the resetting is detected by the receiver, it sends an $ack-$.

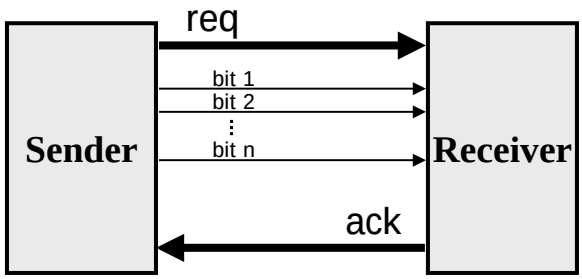


Figure 2.3: Single-Rail Encoding

Single Rail

Single-rail design [30] uses one wire for each bit, as indicated in Figure 2.3. In this encoding scheme, the request signal is used to indicate the validity of the data. The request signal must reach the receiver only after all data bits are valid. This represents a local one-sided timing constraint, called a *bundling constraint*.

Single-rail encoding has the advantage over dual-rail encoding that it uses fewer wires; however, as indicated, it comes with an added one-sided timing constraint. Another advantage of using single-rail is that it enables the use of existing synchronous function blocks within an asynchronous design. That is, the datapath blocks themselves may glitch, as long as a clean bundling signal *req* is transmitted to indicate data validity.

Other Encoding Schemes

There is a wide variety of other encoding schemes. One particularly interesting class of encoding schemes are *Delay-Insensitive (DI) codes* [105, 47] which have the property that a receiver can detect when the sent data is valid. The above introduced dual-rail encoding is one such code, but it can be generalized in many ways. Examples include using transitions on wires instead of levels to represent

data, or using fewer than $2n$ wires to encode n data bits. LEDR, introduced by Dean et. al [31], for example, is a DI code that uses $2n$ wires but does not require a spacer phase between sending successive data bits.

2.1.3.2 Completion Signal Generation

An additional task that is often necessary in asynchronous design is *completion signal generation*, i.e. to indicate when a task has been completed. In particular, a sender that computes data, i.e. a functional unit, must detect when its computation has completed, before it issues a completion signal to the receiver. Techniques for completion detection vary, depending on the data encoding scheme.

If a functional unit encodes its internal data using a *dual-rail code* [63, 106], completion detection is typically implemented using a "completion detector" unit. The detector checks each pair of wires, implementing a bit, and determines if the bit is valid, i.e. no spacer code (00) is present. When all bits are detected as valid, the unit asserts its output. Typical detectors are thus implemented as an AND of two-input ORs[1].

Alternatively, if a functional unit encodes its internal data using *single-rail* [97, 85], completion detection is already part of the implementation: it is the bundled request signal of Figure 2.3. In this implementation style, the request is generated through a worst-case matched delay, which equals or exceeds the longest path in the corresponding data path. Effectively, the completion generation signal is asserted a constant amount of time after the functional unit starts computation, and can thus can be modeled as a fixed delay line. This model can be extended to

[1]The AND may be replaced by an asynchronous join element, called a C-element [97].

include *data-dependent delays* by selecting a number of fixed delay-lines depending on the input data, using an approach called *speculative completion* [74, 83].

In general, there is a wealth of other completion generation techniques [30]. For the purpose of this thesis, no particular technique is assumed. The control synthesis method to be presented in Chapter 5 only assumes that control and datapath communicate using request and acknowledgment signals.

2.2 Asynchronous Controllers

This section gives an overview of one important class of asynchronous controllers, or finite-state machines (AFSM), called *Burst-Mode* [79, 29, 81, 41]. The new hazard-free logic minimization algorithms of Chapters 3 and 4 are suitable for Burst-Mode machines, and the distributed control synthesis method of Chapter 5 is currently targeted to Burst-Mode implementations[2]. In the remainder of this section, first, Burst-Mode specifications, and their targeted implementation style are introduced. Then, so-called *extended* Burst-Mode specifications, which allow somewhat more general specifications, are reviewed. For a more detailed introduction to Burst-Mode specifications and synthesis, see [57].

2.2.1 Burst-Mode Specifications

A Burst-Mode asynchronous finite state machine allows multiple-input changes and is specified by a form of state diagram, called a *Burst-Mode (BM) specification* [79]. An example of a Burst-Mode specification is shown in Figure 2.4. This specification

[2]The synthesis approach would also fit in well with other controller design styles, though this has still to be explored.

describes a simple controller having 3 inputs (a, b, c) and 2 outputs (y, z).

A Burst-Mode specification contains a finite number of *states*, a number of labeled arcs connecting pairs of states, and a distinguished *start state*. Initial wire values are either specified or assumed 0. Arcs are labeled with possible transitions, along which the system moves from one state to another. Each transition consists of a non-empty set of input changes (an *input burst*) and a set of output changes (an *output burst*). Input and output bursts are separated by a slash, `/`. A rising transition is indicated by a "+" and a falling transition is indicated by a "−". Every input burst must be non-empty; if no inputs change, the system is stable.

In a given state, when all the inputs in some input burst have changed value, the system generates the corresponding output burst and moves to a new state. Inputs in a given input burst may arrive in any order at arbitrary times. However, once an input burst is complete, no further input changes may occur until the resulting output changes have occurred (will be explained in detail below in *Target Implementation*).

Burst-Mode specifications must obey three properties. First, the so-called *maximal set property* stipulates that no arc leaving a state may possess an input burst that is a subset of any other arc leaving that state. This property guarantees that for each state, the machine can unambiguously decide whether to follow a transition or to wait for additional inputs. Second, in a given state, *only specified input bursts may occur*; other input combinations are forbidden. For example, in Figure 2.4, input change $a-$ is disallowed in state B, since it is not described by the specification. Third, a given state is always entered with the same set of input values; that is, each state has a *unique entry point*. If these requirements are met,

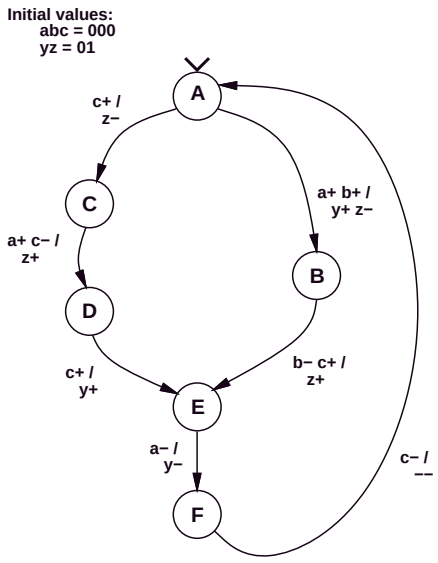


Figure 2.4: Example Burst-Mode Specification.

then it is guaranteed that a hazard-free implementation can be synthesized [57].

Many protocol-based concurrent designs can naturally be represented as burst-mode specifications. Further, generalizations of burst-mode specification have recently been made [111] to extend the set of machines to an even wider and more practical class (called *extended burst-mode*, cf. Section 2.2.2).

Target Implementation

A burst-mode specification can be realized as a *Huffman machine*, as shown in Figure 2.5. These machines must be hazard-free and consist of combinational logic with primary inputs, primary outputs and fed-back state variables [103]. State is stored on the feedback loops, which may have attached delay elements.

The machines that are considered behave as follows. Initially, the machine is stable in some state. Inputs in a specified input burst may then change value

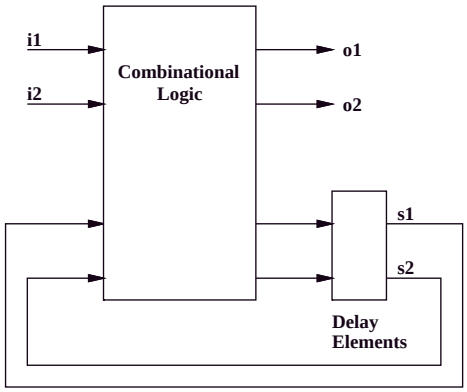


Figure 2.5: Block Diagram of Huffman Machine.

monotonically in any order and at any time. Throughout this input burst, the outputs and state remain unchanged. When the input burst is complete, the outputs change value monotonically as specified, i.e. each output either changes its value once or keeps its value. A state change may also occur concurrently with the output change. In this case, the machine will be driven to a new stable state. In other words, only a *single* feedback cycle occurs. Alternatively, no state change may occur. In either case, no further input may arrive until the machine is stable. That is, the machine operates in a generalized form of *fundamental mode* [103]. When the machine is stable, the cycle is complete and the machine is ready to receive new inputs. Throughout the entire machine cycle, outputs and state variables must be *free of glitches* (i.e. *hazard-free*).

Huffman machines have been well-studied for many years [103]. One of their chief advantages is that input-to-output latency is *purely combinational*: no latches or delays are interposed in the output path.

However, most older Huffman-style designs had a number of problems, e.g., restricted operating mode, hazards, or poor performance. Recently, many of these

problems have been solved using burst-mode methods. A number of such design methods now exist [79, 76, 111, 29, 60].

Burst-mode methods have recently been applied to several large and realistic design examples, including a low-power infrared communications chip at HP Labs and Stanford [60], a second-level cache-controller [77], a SCSI controller [108], a differential equation solver [109], and an experimental instruction-length decoder at Intel [18]. A number of Burst-Mode CAD tools have also been developed, including a locally-clocked method [78], MEAT [29], and 3D [111]. A recent comprehensive CAD package for Burst-Mode synthesis, called MINIMALIST [38, 39, 41], includes many sophisticated algorithms and user scripts, and will be used (along with the 3D tool) to synthesize the controllers in Chapter 5.

2.2.2 Extended Burst-Mode Specifications

Extended Burst-Mode AFSMs (XBM) [110] were introduced to allow two important extensions. First, selected inputs may arrive early, and thus more concurrency is allowed: While in a Burst-Mode AFSM, only input signals corresponding to the current input burst may arrive, in an extended Burst-Mode AFSM, input signals that may arrive early are attached to previous bursts and marked with an asterisks (*directed don't care*). However, each input burst needs to contain at least one signal that cannot arrive early (*compulsory signal*).

Second, extended burst-mode AFSMs allow sampling of level inputs in input bursts (called *conditionals*), indicated by brackets, e.g. $[C+], [C-]$. The value of a conditional in an input burst is sampled only after all transition signals of the burst have occurred. Thus, this is a set-up condition: conditionals must arrive and

be a stable before any compulsory event arrives (see [81, 110] for details).

The targeted asynchronous systems which are presented in Chapter 5 are highly concurrent and therefore some XBM controllers will be used.

2.3 Basic Logic Synthesis

This section first reviews a number of fundamental definitions in Boolean algebra and logic synthesis. Then, two-level logic minimization algorithms are surveyed. Finally, various decision diagrams are introduced, to efficiently represent Boolean functions. This background forms the basis for the new approaches presented in Chapters 3 and 4.

2.3.1 Definitions

The following definitions are taken from Rudell [89], and standard textbooks [32, 66] with small modifications.

Let $\mathbf{B} := \{0, 1\}$ be the set of binary values. $\mathbf{B}^n$ can be modeled as a binary n-cube, and each element $e = (e_1, \ldots, e_n) \in \mathbf{B}^n$ is called a minterm. Figure 2.6a. shows $\mathbf{B}^3$, the three-dimensional Boolean space, and its minterms. Note that the well-known binary Boolean algebra is given by the the set $\mathbf{B}$ together with the operations + (also called disjunction, sum, OR) and $\cdot$ (conjunction, product, AND).

A *Boolean function* f of n variables, $x_1, \ldots, x_n$, is a mapping $f : \mathbf{B}^n \rightarrow \{0, 1, *\}$. Here, the symbol $*$ denotes a *don't care* condition, i.e. the value of the function does not matter. Note that a minterm indicates which values are assigned to the variables of a function, i.e. $x_1 = e_1, x_2 = e_2$, and so on.

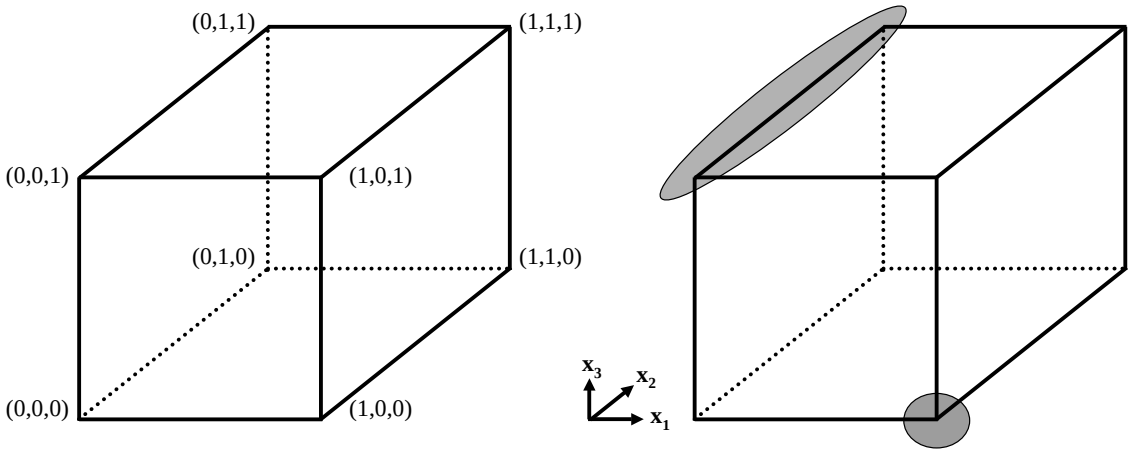


Figure 2.6: a. Binary cube $\mathbf{B}^3$ and its minterms; b. Minterm $x_1\overline{x}_2\overline{x}_3$ and cube $\overline{x}_1x_3$

The *ON-set* of a Boolean function f is defined as the set of minterms for which the function has value 1. Similarly, the *OFF-set* and *DON'T-CARE-set* are defined as the set of minterms for which the function has value 0 and *, respectively.

Boolean functions as defined above are often referred to as *single-output* Boolean functions. A *multi-output* Boolean function is a mapping $f : \mathbf{B}^n \rightarrow \{0, 1, *\}^m$. Note that each of the output functions $f_1, \ldots, f_m$ has its own ON-set, OFF-set, and DON'T-CARE-set associated with it. For the sake of simplicity of presentation, only single-output functions are considered in the remainder of this section.

Each variable x_i has two *literals* associated with it: an *uncomplemented* (or *positive*) literal x_i, and a *complemented* (or *negative*) literal $\overline{x}_i$ or x_i'. The literal x_i ($\overline{x}_i$) represents a Boolean function which evaluates to 1 (0) for minterms with $e_i = 1$, and to 0 (1) for minterms with $e_i = 0$.

A *product* term is a Boolean product (AND) of literals. That is, a product evaluates to 1 for a minterm e, if each literal *included* in the product evaluates to

1 for the minterm e. Otherwise, the product evaluates to 0. In the former case, the product is said to *contain* minterm e. Note that each minterm corresponds to a product that only contains the given minterm. More specifically, the minterm $e = (e_1, \ldots, e_n)$ corresponds to the product $x_1^{e_1} \cdots x_n^{e_n}$, where $x_i^{e_i}$ denotes the positive (negative) literal of x_i if $e_i = 1(0)$. For example, the minterm $e = (1, 0, 1)$ corresponds to the product $x_1\overline{x_2}x_3$, which is often used as a convenient abbreviation.

Since a product corresponds to a set of adjacent minterms in the binary n-cube, a product is also referred to as a *cube*. Figure 2.6b. shows a cube and a minterm in $\mathbf{B}^3$.

A cube c is *contained in* a cube d ($c \subseteq d$) if each minterm contained in c is also contained in d. The *intersection* of cubes c and d ($c \cap d$) is the cube which contains those minterms contained in both cubes. The *supercube* of cubes c and d, denoted *supercube(c,d)*, is the uniquely defined smallest cube that contains both cubes. For example, if $c = x_1\overline{x}_2$, and $d = \overline{x}_1\overline{x}_2x_3$, then *supercube(c,d)* $= \overline{x_2}$. In general, to compute the supercube each literal is considered. A literal is included in the supercube of two cubes if and only if it is included in both cubes.

A *sum-of-products* is a Boolean sum (OR) of products. That is, a sum-of-products evaluates to 1 for a given minterm if some product contains the minterm.

An *implicant* of a Boolean function is a cube which contains no minterm in the OFF-set. A *prime implicant* is an implicant contained in no other implicant of the function. An *essential prime implicant* is a prime implicant containing at least one ON-set minterm which is not contained in any other prime implicant.

A *cover* of a Boolean function is a set of implicants interpreted as a sum-of-products, which evaluates to 1 for all the minterms of the ON-set, and none of the

OFF-set.

A Boolean function f is *monotone increasing* in variable x_i if changing the value of x_i from 0 to 1 causes f to change its value from 0 to 1 or to stay constant. A similar definition in reverse defines a *monotone decreasing* function. A Boolean function is *unate*, or *monotone*, in variable x_i if it is either monotone increasing in x_i or monotone decreasing in x_i. A function is *unate*, or *monotone*, if it is unate in all its variables.

The *complement* of a Boolean function f is denoted by $\overline{f}$, and evaluates to 1 (0) if f evaluates to 0 (1).

The *cofactor* of a Boolean function f with respect to literal x_i is the Boolean function obtained by setting variable x_i to 1, i.e. $f_{x_i} = f(x_1, \ldots, 1, \ldots, x_n)$. Similarly, the cofactor with respect to literal $\overline{x}_i$ is defined as $f_{\overline{x}_i} = f(x_1, \ldots, 0, \ldots, x_n)$.

2.3.2 2-Level Logic Minimization

The *two-level logic minimization problem* is to find the minimum-cost sum-of-products, or cover, of a Boolean function. In digital design, such a cover can be implemented as a minimum-cost AND-OR (two-level) circuit. Here, the cost, or size, of a cover is often defined as the number of cubes in the cover, which will be used in the following presentation.

This section surveys the most significant previous algorithms for solving the two-level logic minimization problem.

Quine-McCluskey

The classic Quine-McCluskey algorithm [65, 89] to solve the two-level minimization problem is based on the insight that the implicants in a minimum-cost cover can be restricted to prime implicants. (Assume a minimum-cost cover included a non-prime implicant, then the non-prime implicant could be replaced by a prime implicant covering it.)

The Quine-McCluskey algorithm consists of two steps:

1. generate the set of all prime implicants;
2. select a minimum number of prime implicants such that each ON-set minterm is contained.

The first step — to generate the set of prime implicants — starts from the set of ON-set minterms, and it iteratively computes larger implicants by removing literals. The prime implicants are then those implicants from which no more literals can be removed.

The second step is a so-called *unate set covering problem* [66]. In general, a unate set covering problem is defined for two sets X and Y and a relation R defined over $X \times Y$. The set covering problem $\langle X, Y, R \rangle$ consists of finding a minimum subset S of Y such that for any $x \in X$, there exists an element $y \in S$ with $(x, y) \in R$. The two-level logic minimization problem thus reduces to solving the set covering problem $\langle$ ON-set(f), Prime(f), $\subseteq \rangle$.

The Quine-McCluskey algorithm solves the set covering problem using a covering matrix. Here, rows are labeled with the ON-set minterms and columns are labeled with the prime implicants. Each entry in the table thus corresponds to

a minterm and a prime implicant, and it is defined as 1 if and only if the minterm is covered by the prime implicant. The covering table is iteratively reduced in size by exploiting dominance relations between set of rows, or sets of columns, and by detecting essential prime implicants and including them in the solution. Branch-and-bound algorithms have been proposed to find the minimum solution from the reduced table.

More recent algorithms for two-level logic minimization follow the Quine-McCluskey algorithm but with a number of improvements.

Espresso-Exact

The Espresso-Exact algorithm follows the main principles of the above described Quine-McCluskey algorithm but employs different algorithms in each of the steps.

Espresso-Exact uses a recursive prime generation technique that exploits the wealth of research on efficient prime generation techniques (see [25] for an overview).

Espresso-Exact contributes the construction of a smaller reduced covering matrix where each row corresponds to *collections* of minterms all covered by the same subset of primes. In addition, Espresso-Exact also includes a much improved branch-and-bound algorithm.

Scherzo

Scherzo [25] is currently the state-of-the-art exact two-level logic minimization algorithm for synchronous minimization problems. Using *implicit* minimization

techniques, i.e. using data structures that facilitate the manipulation of a large number of objects simultaneously, SCHERZO is 10 to more than 100 times faster than the best previous minimization methods. In addition, SCHERZO has solved examples with 10^{20} prime implicants, which is clearly out-of-reach of the above introduced minimization algorithms.

SCHERZO will be introduced in detail in Section 4.3.1. Our proposed approach for exact *hazard-free* logic minimization to be presented in Chapter 4 is based on ideas of SCHERZO, but with a number of significant additions to cope with the added hazard-freedom constraints, which must be considered in asynchronous logic synthesis.

ESPRESSO-II

Since solving the 2-level logic minimization problem involves computationally intractable problems, heuristic approaches have been developed as well.

ESPRESSO-II [88, 35] is the state-of-the-art tool for synchronous *heuristic* two-level logic minimization. The output of ESPRESSO-II is a cover, which is in practice almost always (near-)minimum in cardinality. The tool is very efficient and is used world-wide.

ESPRESSO-II will be introduced in detail in Section 3.2. The proposed approach for heuristic *hazard-free* logic minimization, ESPRESSO-HF, to be presented in Chapter 3 is based on ideas of ESPRESSO-II, but with a number of significant changes to cope with hazard-freedom constraints.

2.3.3 Decision Diagrams

This section introduces data structures to efficiently represent Boolean functions and sets of products, called *Binary Decision Diagrams* and *Zero-Suppressed Binary Decision Diagrams*. Both of these data structures have been used in SCHERZO [25], discussed above, and will also be used in the new exact hazard-free two-level minimization algorithm presented in Chapter 4.

2.3.3.1 BDDs

Binary Decision Diagrams (BDDs) [15] are used to efficiently represent Boolean functions. An ordered BDD (OBDD) is a canonical representation of a function f which is obtained from the Shannon tree representation of f by reduction rules which (i) identify isomorphic subgraphs and (ii) delete each vertex that has the same left and right children.

Example 2.1 In Figure 2.7a) the Shannon tree of the function $f = ab+c$ is shown. To find the function value for a specific assignment to the variables, one follows the path from the root node to a terminal node, taking the left (right) branch if the corresponding variable is assigned the value 0 (1). The corresponding BDD obtained by above reduction rules is shown in part b) of the figure. Note that the BDD of f is just a compact representation of the Shannon tree of f. In particular, the same algorithm can be used to evaluate the function for an assignment to the variables. □

Important properties of BDDs include canonicity of representation (if the variable ordering is fixed), and the efficiency of binary operators, e.g. the Boolean

AND of two functions represented by BDDs can be efficiently computed in time proportional to the product of the number of nodes of the two BDDs.

2.3.3.2 ZBDDs

Zero-suppressed BDDs (ZBDDs) [67] are a variant of BDDs which were introduced to efficiently represent *sets of products*, e.g. the set of prime implicants of a function f. A ZBDD of a set of products is obtained from a tree representation of the set of products by reduction rules which (i) identify isomorphic subgraphs and (ii) delete each vertex whose right children points to 0 (i.e. the empty set). Note that to achieve small representations for sparse sets, the second reduction rule differs from the second reduction rule for BDDs. Another difference from BDDs is that a ZBDD makes decisions based on *literals* instead of *variables*.

Example 2.2 Consider Figure 2.7c), which shows the tree representation of the given set of products $\{b', a', a'b', a, ab'\}$. Here each path from the root node to a terminal 1 node corresponds to a product in the set. The product consists of those *literals* encountered on taking right branches on the path. Here, positive (negative) literals are denoted by a '+' superscript ('-' superscript). The ZBDD for this set of products obtained by above reduction rules is shown in part d) of the figure. □

Important properties of ZBDDs include canonicity of representation and efficient computation of set-operations, such as union and intersection.

2.3.3.3 Implicit Techniques

Based on Binary Decision Diagrams new efficient algorithms to solve standard problems in digital design have been developed. The main idea of *implicit* approaches

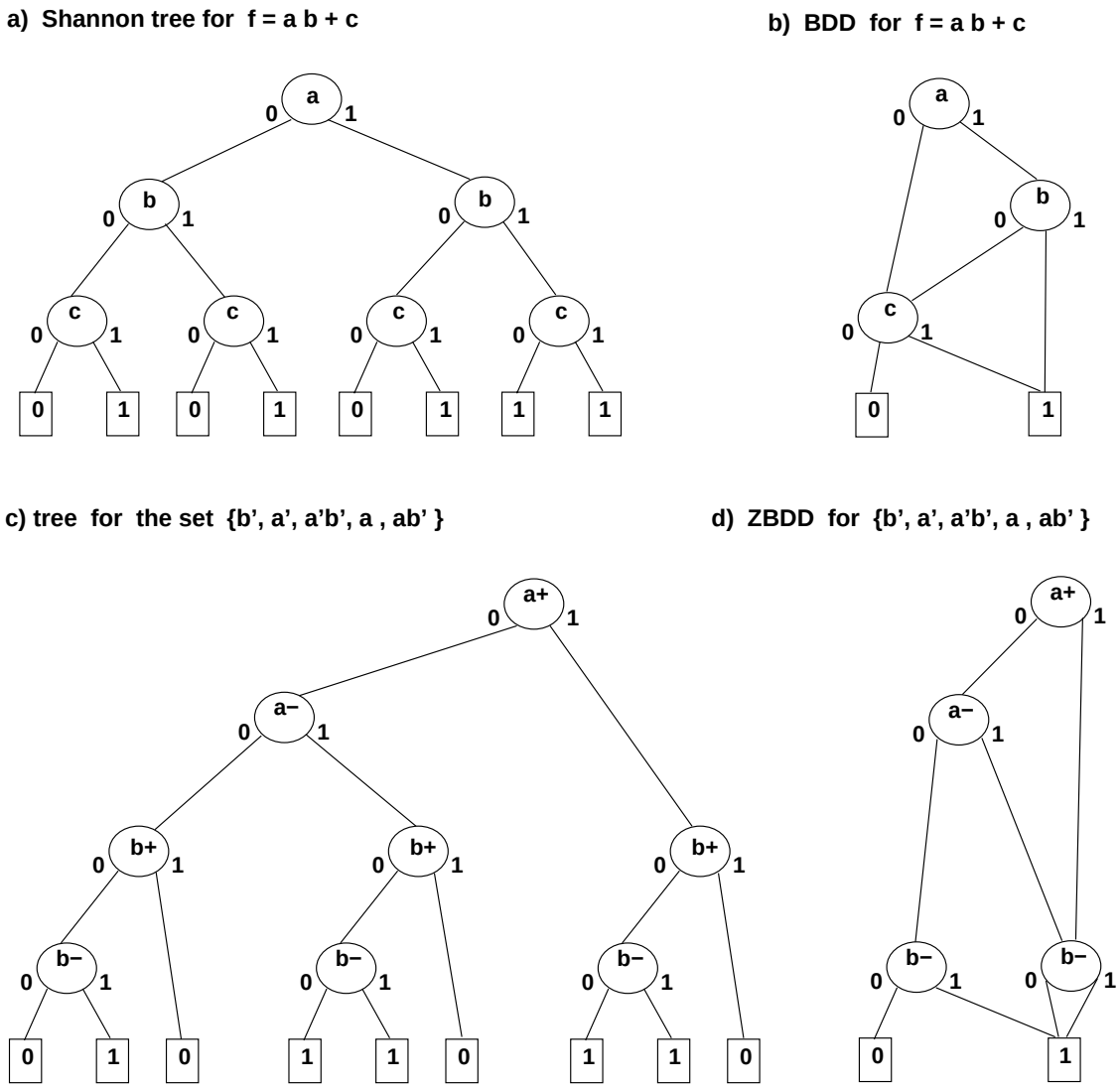


Figure 2.7: BDDs and ZBDDs

is to use compact data structures such as BDDs and zero-suppressed BDD to eliminate the need to explicitly represent each object of interest (e.g. prime implicants, minterms). Such implicit approaches typically can solve much more complex problems than previous methods.

One standard problem for which a new efficient implicit approach has been developed is the generation of all prime implicants. SCHERZO [25] incorporates an algorithm that computes the set of prime implicants for a Boolean function f as follows. It first computes a BDD for the Boolean Function f from an arbitrary unoptimized cover specified as a PLA file (e.g. from the set of ON-set minterms). Then, it uses the BDD representation of f to directly generate a ZBDD that represents the set of prime implicants of f. The computation time to generate the ZBDD as well as the size of the ZBDD are both independent of the number of prime implicants, and, in practice, this method is very efficient. In fact, it has been used to compute ZBDDs that represent up to 10^{20} prime implicants.

In this thesis, implicit techniques are used in Chapter 4, for a new efficient minimizer for 2-level hazard-free logic.

2.4 Asynchronous Combinational Logic Synthesis

The key challenge in asynchronous logic synthesis is to design hazard-free circuits, i.e. circuits that are guaranteed not to glitch when its inputs are changed from one input vector to another. This problem is the focus of Chapters 3 and 4.

In synchronous designs, glitches are typically filtered out by the global clock,

i.e. are permitted as long as the signal is stable when sampled. In contrast, asynchronous systems have no clock, and glitches may be interpreted as additional communication, and thus may result in system malfunction. Figure 2.8 visualizes some possible dynamic behaviors during input vector changes. The goal of hazard-free logic minimization is to find a circuit implementation whose output for a given (set of) input vector change(s) either stays constant or makes *exactly* one clean change.

Hazard-free logic minimization has been studied since the 1950's. The early works by Huffman [46] and McCluskey [64] focussed on the case when only one input in the input vector is allowed to change. Eichelberger [34] and Frackowiak [36] made significant contributions towards handling multiple-input changes. A complete and exact formulation of hazard-free logic minimization of two-level circuits for multiple-input changes was developed by Nowick and Dill [82]. Other important work on hazards include the work by Unger [103] for sequential circuits, and the work by Bredeson and Hulina [12, 11] and Kung [55] on multi-level logic synthesis.

The material of this section therefore focuses on hazards and hazard-free logic minimization[37, 103]. Hazard-free combinational logic synthesis is an important substep of burst-mode synthesis. For the following discussion, a combinational circuit model is assumed where gates and wires may have arbitrary finite delays (*unbounded wire delay model*). The goal of hazard-free synthesis is to generate a combinational circuit implementation which is guaranteed glitch-free regardless of gate and wire delays.

For simplicity, the focus in the following presentation is on single-output functions. A generalization of these definitions to multi-output functions is straight-

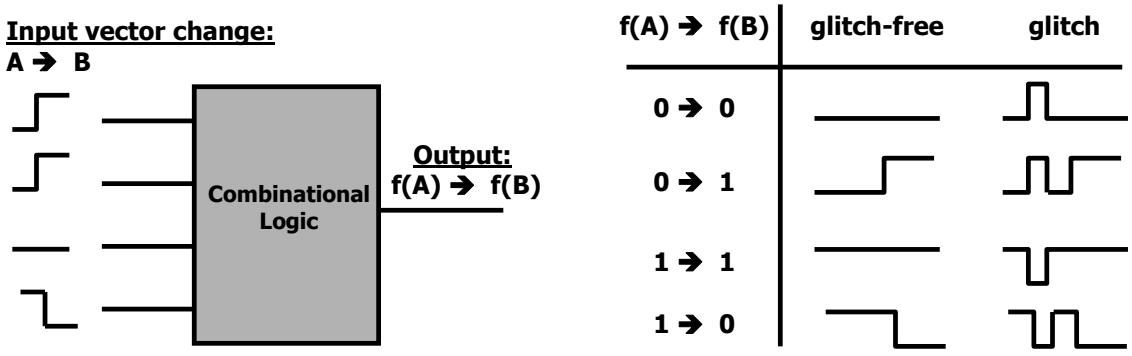


Figure 2.8: Hazard-Free Logic Synthesis: Glitch-Free Transitions and Transitions with Glitches

forward, and is described in [37]. More details on definitions and theorems presented in the following sections can be found in [5, 36, 12, 82].

2.4.1 Multiple-Input Change

Since the concern is the dynamic behavior of a combinational circuit as its inputs change value, the notion of a "multiple-input change", also called "input transition", or "input vector change" needs to be formalized.

Definition 2.3 *Let A and B be two minterms. The* **transition cube**, $[A, B]$, *from A to B has* **start point** A *and* **end point** B, *and contains all minterms that can be reached during a transition from A to B. More formally, $[A, B]$ is the uniquely defined smallest cube that contains A and B:* supercube(A,B). *A transition cube $[A, B]$ describes a* **input transition** *or* **multiple-input change** *from minterm A to B.*

A multiple-input change specifies what variables change value and what the corresponding *starting* and *ending* values are. Input variables are assumed to change

simultaneously. (Equivalently, since inputs may be skewed arbitrarily by wire delays, inputs can be assumed to change monotonically, *i.e.* at most once, in any order and at any time.) Once a multiple-input change occurs, no further inputs may change until the circuit has stabilized. In this thesis, only transitions where f is fully defined are considered; that is, for every $X \in [A, B]$, $f(X) \in \{0, 1\}$.

Figure 2.9 shows multi-input changes as arrows and transition cubes are surrounded by dotted lines.

2.4.2 Combinational Hazards

Combinational hazards fall into two classes: *function hazards* and *logic hazards*.

2.4.2.1 Function Hazards

A function f which does not change monotonically during an input transition is said to have a *function hazard* in the transition. Figure 2.9 shows a non-monotone and a monotone transition. The non-monotone transition includes a path from the start to the end point on which the function value changes more than once.

Definition 2.4 *A function f contains a* **static function hazard** *for the input transition from A to C if and only if: (1) $f(A) = f(C)$, and (2) there exists some minterm $B \in [A, C]$ such that $f(A) \neq f(B)$.*

Definition 2.5 *A function f contains a* **dynamic function hazard** *for the input transition from A to D if and only if: (1) $f(A) \neq f(D)$; and (2) there exist a pair of minterms, B and C, such that (a) $B \in [A, D]$ and $C \in [B, D]$, and (b) $f(B) = f(D)$ and $f(A) = f(C)$.*

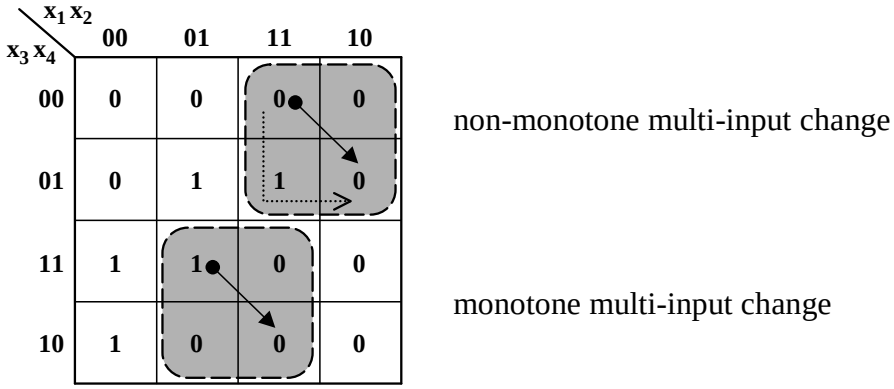


Figure 2.9: Function Hazard

If a transition has a function hazard, *no* implementation of the function is guaranteed to avoid glitches during the transition (assuming a circuit model of arbitrary gate and wire delays; see [82, 103]).

Therefore, only input transitions which are monotone are considered, i.e. the considered transitions are *function-hazard-free*. Sequential synthesis methods, such as burst-mode, which use hazard-free minimization as a substep, include constraints in their algorithms such that no transitions with function hazards are generated [79, 110].

2.4.2.2 Logic Hazards

If f is free of function hazards for a transition from input A to B, an implementation may still glitch due to delays in the actual gates and wires. In this case, the circuit is said to have a *logic hazard* for the input transition. Two examples of logic hazards are given in Figures 2.10 (top), where a $1 \rightarrow 1$ transition may result in a glitch, and 2.11 (middle), where a $1 \rightarrow 0$ transition may result in a glitch.

Definition 2.6 *A circuit implementing function f contains a* **static (dynamic)**

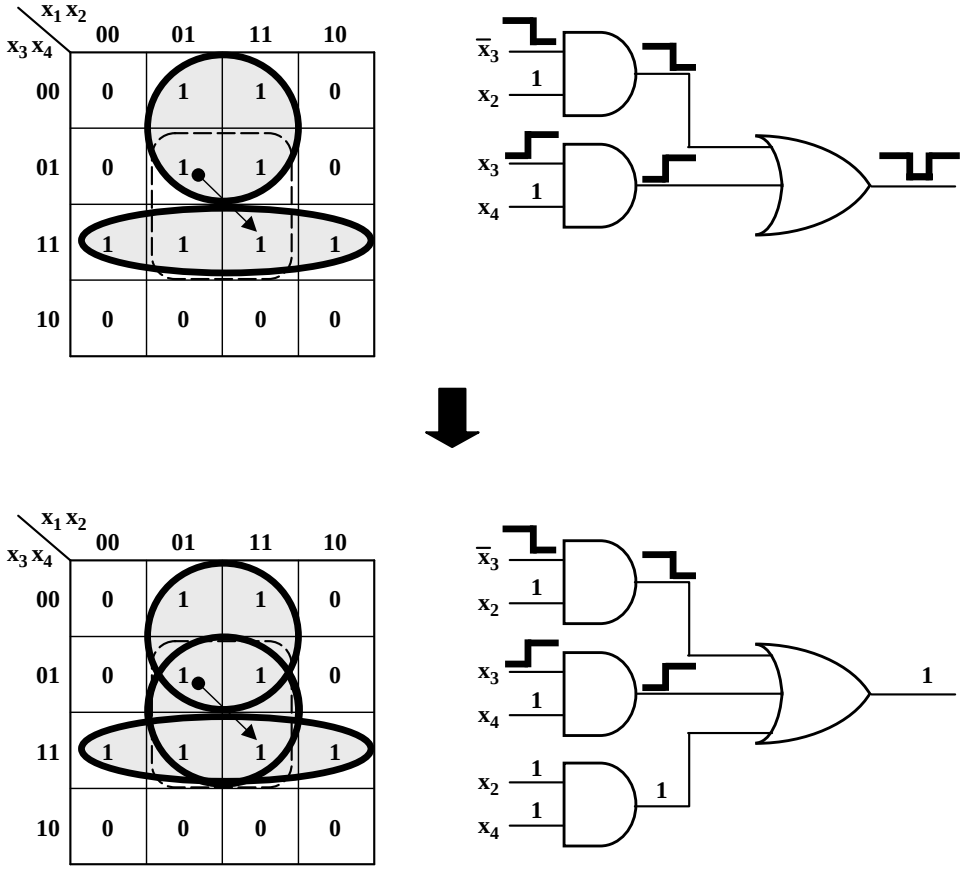


Figure 2.10: Static Logic Hazards

logic hazard *for the input transition from minterm A to minterm B if and only if: (1) $f(A) = f(B)$ ($f(A) \neq f(B)$), and (2) for some assignment of delays to gates and wires, the circuit's output is not monotonic during the transition interval.*

That is, a static logic hazard occurs if $f(A) = f(B) = 1$ (0), but the circuit's output makes an unexpected $1 \rightarrow 0 \rightarrow 1$ ($0 \rightarrow 1 \rightarrow 0$) transition (see Figure 2.10 top). A dynamic logic hazard occurs if $f(A) = 1$ and $f(B) = 0$ ($f(A) = 0$ and $f(B) = 1$), but the circuit's output makes an unexpected $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ ($0 \rightarrow 1 \rightarrow 0 \rightarrow 1$) transition (see Figure 2.11).

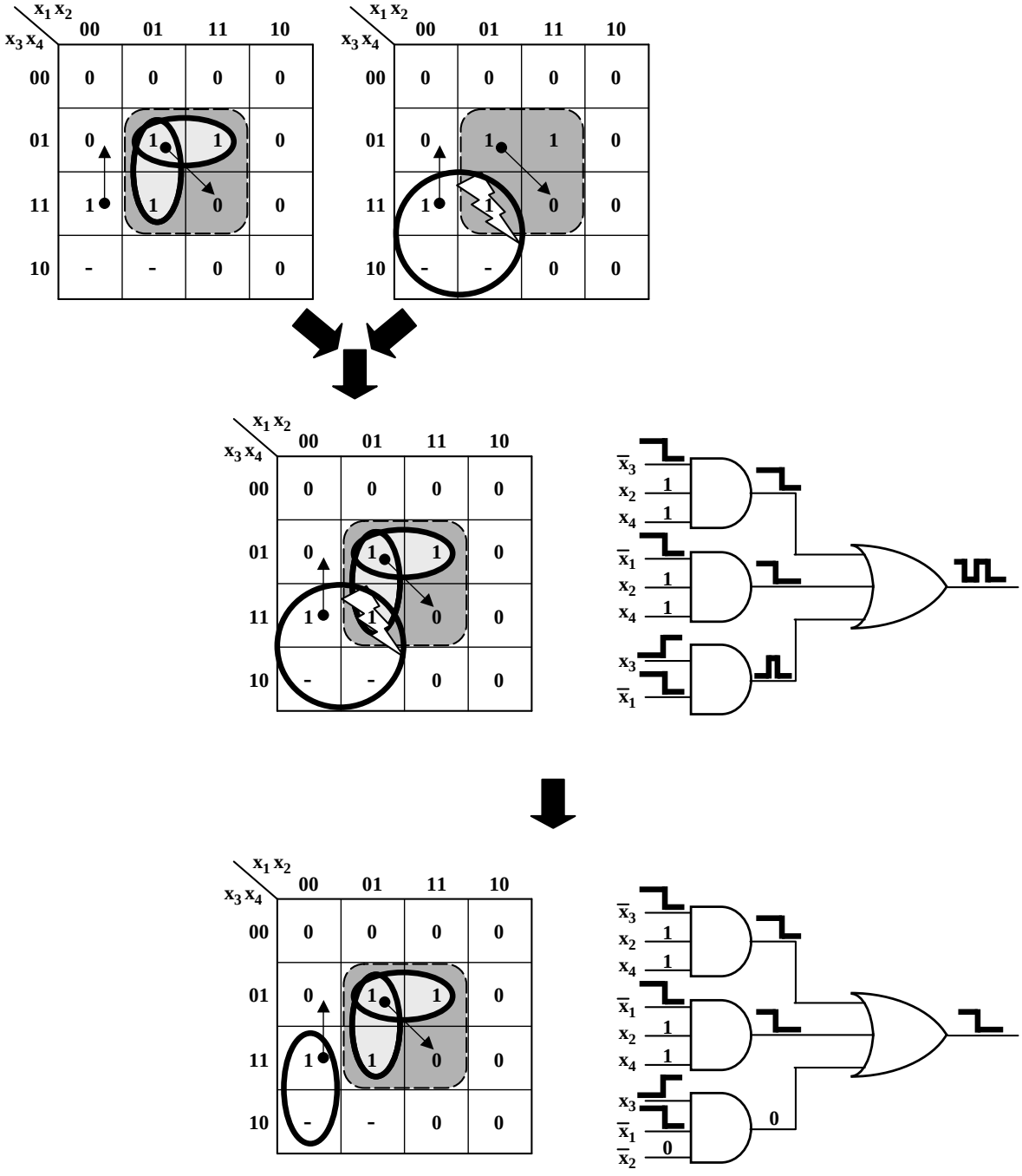


Figure 2.11: Dynamic Logic Hazards

2.4.3 2-Level Logic Minimization: Classic vs. Hazard-Free

The *2-level hazard-free minimization problem* can now be stated as follows: Given a function f, and a set T of specified function-hazard-free input transitions, find a minimum-cost AND-OR (sum-of-products) implementation of f, which is (logic) *hazard-free* for all the specified transitions.

This problem is a constrained variant of the classic 2-level minimization problem, i.e. to find a minimum-cost AND-OR implementation of a function f. Classic 2-level minimization can be reduced to a covering problem, where each ON-set minterm must be covered by (contained in) a prime implicant.

The 2-level *hazard-free* minimization problem can be solved in a very similar way. The main difference is that now cubes (instead of traditional individual minterms) must be covered, called *required cubes*, using so-called *dynamic-hazard-free(dhf)*-prime implicants (instead of prime implicants). These terms will be described in more detail in the sequel.

2.4.4 Conditions to Avoid Logic Hazards

Conditions to avoid logic hazards in a sum-of-products implementation for a given input transition are now described (for more details, see [82]).

Assume that $[A, B]$ is the transition cube corresponding to a *function-hazard-free* transition from minterm A to B for a function f. Function f is said to have a $f(A) \rightarrow f(B)$ transition in cube $[A, B]$. Now, hazard-free conditions for each of the four possible transition types are considered in turn, i.e. $1 \rightarrow 1$, $1 \rightarrow 0$, $0 \rightarrow 1$, and $0 \rightarrow 0$.

If for a given input transition the function value is 1 for both start and end

point (a $1 \rightarrow 1$ transition), *some* product must hold its value at 1 throughout the transition to ensure no hazards. For this reason, the entire transition cube is called a *required cube*, which must be completely contained in some product of the cover. In Figure 2.10, the top implementation is hazardous as it consists of two products but neither covers the entire transition, i.e. the dotted required cube is not contained in any product. In contrast, the bottom implementation includes a product that contains the required cube and is thus hazard-free.

Lemma 2.7 *If f has a $1 \rightarrow 1$ transition in cube $[A, B]$, then the cube $[A, B]$ is called a* **required cube**. *The implementation is free of logic hazards for the input change from A to B* if and only if *the required cube $[A, B]$ is contained in some cube of cover F.*

For the $1 \rightarrow 0$ case, first, each $1 \rightarrow 1$ sub-transition must be free of logic hazards, so the corresponding required cubes must each be contained in some product. The K-map at the top left of Figure 2.11 shows a transition cube that gives rise to two required cubes. Second, no product in the implementation may *illegally intersect* the $1 \rightarrow 0$ transition, i.e. intersect the transition, but not contain the transition's start point. Otherwise, the product may glitch *in the middle* of the transition, i.e. a dynamic hazard will result. For this reason, the $1 \rightarrow 0$ transition cubes are also called *privileged cubes*. Figure 2.11 (top right) shows an illegal intersection of the privileged cube, resulting in a hazard as visualized by the middle part of the figure. Removal of the illegal intersection leads to a hazard-free cover (bottom).

Lemma 2.8 *If f has a $1 \rightarrow 0$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B* if and only if

1. *Every maximal subcube* $[A, X] \subset [A, B]$ *where* f *is 1, called a* **required cube**, *is contained in some cube of cover* F.

2. *Every cube* $c \in F$ *intersecting* $[A, B]$ *(*$[A, B]$ *is called a* **privileged cube***), also contains the start point* A*, i.e., no product illegally intersects the privileged cube* $[A, B]$.

A $0 \rightarrow 1$ transition may be regarded as a $1 \rightarrow 0$ transition in reverse, and the same conditions apply. More specifically, a $0 \rightarrow 1$ transition from A to B has the same hazards as a $1 \rightarrow 0$ transition from B to A.

Finally, for a $0 \rightarrow 0$ transition, there are no additional constraints.

Lemma 2.9 *If* f *has a* $0 \rightarrow 0$ *transition in cube* $[A, B]$*, then the implementation is free of logic hazards for the input change from* A *to* B.

There are two additional sets of definitions which will be useful in later chapters.

Transitions fall into two broad categories. $1 \rightarrow 1$ transitions and $0 \rightarrow 0$ transitions are called *static transitions*. $1 \rightarrow 0$ transitions and $0 \rightarrow 1$ transitions are called *dynamic transitions*.

Finally, a useful degenerate special case for privileged cubes is defined. A privileged cube is called **trivial**, if the function is only 1 at the start point and is 0 for all other minterms included in the transition cube. In this case, any product that intersects such a privileged cube always covers the start point. All trivial privileged cubes can safely be removed from consideration without loss of information.

2.4.5 Hazard-Free 2-Level Logic Minimization Problem

A *hazard-free cover* is a cover (sum-of-products implementation) of a function which is hazard-free for a *set* of specified input transitions. (It is assumed below that the function is defined for the entire transition cube of all specified transitions; the function is undefined for all other input states.)

An implicant which does not illegally intersect any privileged cube is called a *dynamic-hazard-free implicant* (or *dhf-implicant*). Only dhf-implicants may appear in a hazard-free cover. A *dhf-prime implicant* is a dhf-implicant contained in no other dhf-implicant.

Finally, the *two-level hazard-free logic minimization problem* is therefore to find a minimum-cost hazard-free cover of a function *(i)* using only dhf-prime implicants *(ii)* where every required cube is covered. Formally:

Theorem 2.10 (Hazard-Free Covering [82])

A sum-of-products F is a hazard-free cover for function f for the set T of specified input transitions if and only if:

(a.) No product of F intersects the OFF-set of f;

(b.) Each required cube *of f is contained in some product of F; and*

(c.) No product of F intersects any (non-trivial) privileged cube *illegally.*

Definition 2.11 *A* **dhf-implicant** *is an implicant which does not intersect any privileged cube of f illegally. A* **dhf-prime implicant** *is a dhf-implicant contained in no other dhf-implicant. An* **essential dhf-prime implicant** *is a dhf-prime implicant which contains a required cube contained in no other dhf-prime implicant. The* **two-level hazard-free logic minimization problem** *is* to find a minimum

cost cover of a function using only dhf-prime implicants where every required cube is covered.

The difference between two-level hazard-free logic minimization and the well-know classic two-level logic minimization problem (e.g. solved by Quine-McCluskey algorithm) is that, in the hazard-free case, dhf-prime implicants replace prime implicants as the covering objects, and required cubes replace minterms as the objects-to-be-covered.

There exist Boolean functions and sets of transitions such that the covering conditions of Theorem 2.10 cannot be satisfied [103, 82]. This case occurs if conditions (b) and (c) cannot be satisfied simultaneously.

Example. A complete hazard-free minimization example is shown in Figure 2.12, adapted from [81]. Part a. shows the three specified transitions. In Part b. the set of required cubes is indicated, and Part c. shows a minimal hazard-free cover. Part d. shows a minimum-cost cover that is not hazard-free, since it contains a logic hazard due to an illegal intersection. In fact, the cover also fails to cover the required cube $\overline{x}_1 x_2 \overline{x}_3$.

2.4.6 Existing Hazard-Free Minimization Algorithms

The first exact hazard-free minimizer was developed by Nowick and Dill [82]. It has recently been extended to exact hazard-free multi-output and symbolic minimization by Fuhrer, Lin and Nowick [37, 41]. The latter method, called HFMIN, was until recently the fastest minimizer for exact hazard-free minimization.

HFMIN makes use of ESPRESSO-II to generate all prime implicants, then to transform them into dhf-prime implicants, and finally employs ESPRESSO-II's

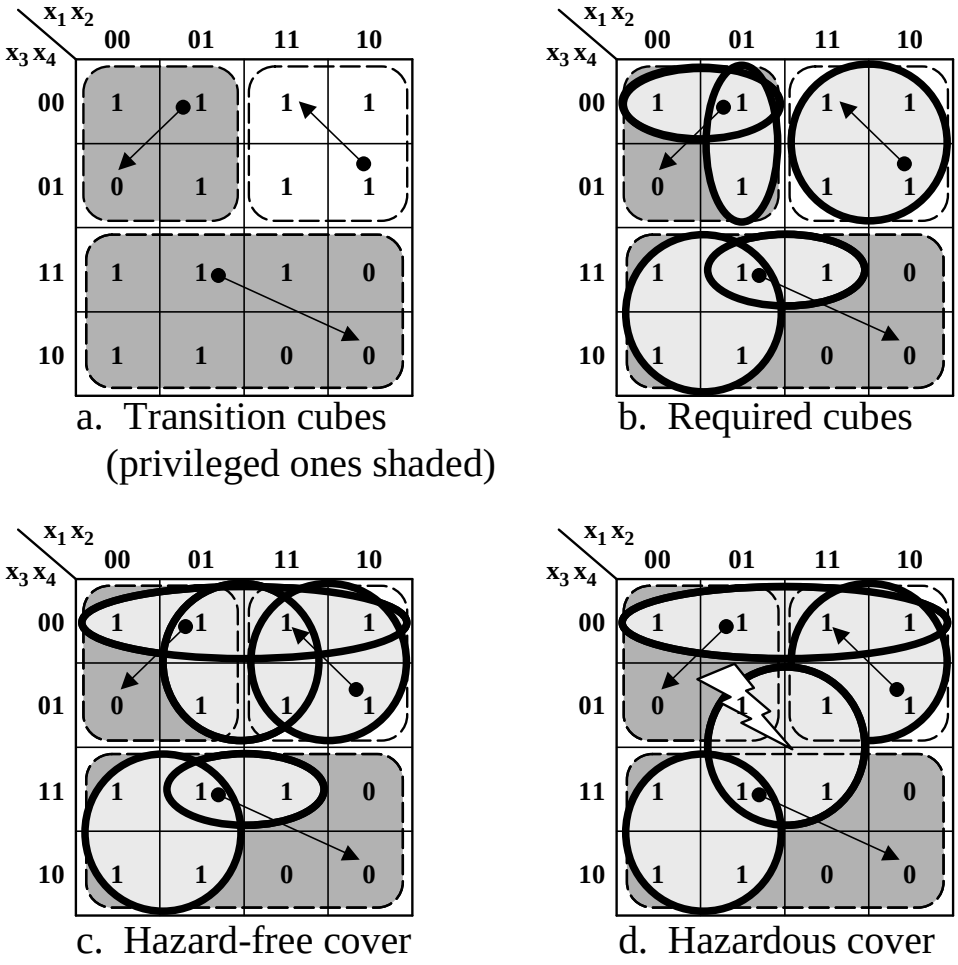


Figure 2.12: Two-Level Hazard-Free Minimization Example

MINCOV to solve the resulting unate covering problem. Each of the algorithms used in the above three steps is critical, i.e. has a worst-case run-time that is exponential. As a result, in practice, HFMIN *cannot* solve several of the more difficult examples. Thus, new more powerful techniques must be developed.

Very recently, Rutten [92, 91], and Myers and Jacobson [70] have proposed alternative exact methods. A comparison of the proposed new methods with their approaches is given in Chapter 4.

Chapter 3

Heuristic Hazard-Free Logic Minimization: ESPRESSO-HF

This chapter introduces the first research contribution, ESPRESSO-HF, a new efficient heuristic minimizer for hazard-free two-level logic.

3.1 Introduction

Hazard-free 2-level logic minimization is a bottleneck in asynchronous CAD packages [79, 29, 76, 111, 54, 38]. While the currently-used Quine-McCluskey-like exact hazard-free minimization algorithm, HFMIN [37], has been effective on small- and medium-sized examples, it has been unable to produce solutions for several large design problems [54, 90].

This chapter introduces ESPRESSO-HF, which is a new algorithm to solve the heuristic hazard-free two-level logic minimization problem. The method is heuristic solely in terms of the cardinality (i.e. number of products) of solution. In all cases,

it guarantees a hazard-free solution. The algorithm is based on the well-known synchronous algorithm ESPRESSO-II [88, 35], but with a number of significant modifications to handle hazard-freedom constraints. ESPRESSO-HF also includes a new and much more efficient algorithm to check for existence of a hazard-free solution, without generating all prime implicants.

The new prototype tool can solve all examples that are available so far, very efficiently, and almost always obtains an exactly minimum cover. It also solves several examples which have not been solved before. In fact, ESPRESSO-HF can find very good covers for all currently-available examples — at most 3% larger than a minimum-size cover — in less than 105 seconds.

The remainder of this chapter is structured as follows. Section 3.2 gives background on the state-of-the-art synchronous heuristic two-level logic minimizer, ESPRESSO-II. Section 3.3 then gives a top-level overview of ESPRESSO-HF. In Section 3.4 the individual steps of ESPRESSO-HF are presented in more detail. Section 3.5 presents a new contribution which is a by-product of the ESPRESSO-HF algorithm: to determine, given a Boolean function f and a set of input transitions, if a hazard-free cover exists. Section 3.6 gives experimental results, and Section 3.7 gives conclusions.

3.2 Background on ESPRESSO-II

Since the basic minimization strategy of ESPRESSO-HF for hazard-free minimization is similar to the one used by ESPRESSO-II, this section gives background on ESPRESSO-II.

ESPRESSO-II, developed in the early 1980s, is a very powerful tool for synchronous heuristic two-level logic minimization, and the tool is used world-wide.

The input to ESPRESSO-II is the Boolean function to be minimized, specified in terms of an arbitrary set of implicants (e.g. all ON-set minterms, or possibly larger cubes). This set of implicants represents an initial unoptimized cover, or solution. The output of ESPRESSO-II is a cover, which is in practice almost always (near-) minimum in cardinality. ESPRESSO-II iteratively minimizes the size of the cover by applying three main operators in its main loop, until no further improvement is possible:

- EXPAND enlarges each implicant of the current cover, in turn, into a prime implicant.
- IRREDUNDANT makes the current cover irredundant by deleting a maximal number of redundant implicants from the cover.
- REDUCE sets up a cover that is likely to be made smaller by the following EXPAND step. To achieve this goal, each cube in a cover is maximally reduced, in turn, to a smaller cube such that the resulting set of cubes is still a cover.

Example. Figure 3.1 illustrates the EXPAND operator. As indicated in the example, an implicant may possibly be expanded in alternative dimensions, and the actual algorithm uses heuristics to decide which direction is best.

Figure 3.2 illustrates the main loop of the algorithm. In Part A. an initial cover of the function is given as its set of ON-set minterms. Part B. presents the resulting cover after EXPAND is applied to only one minterm, the one labeled 1.

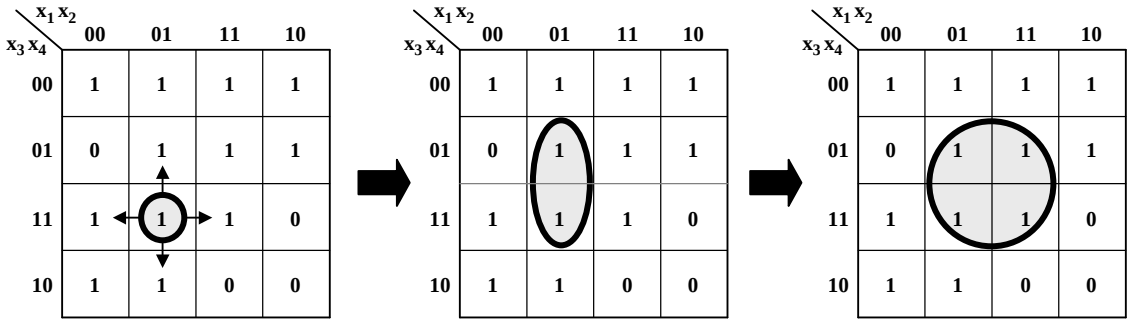


Figure 3.1: Expand in ESPRESSO-II

As it is expanded it includes other cubes of the cover, which are in turn deleted from the cover. Part C. shows the cover as obtained after applying the EXPAND operator to the entire initial cover, resulting in a cover of size five. Next, the application of the operator IRREDUNDANT removes the one redundant cube from the cover, and the result is shown in Part D. That is, the cover now consist of four cubes. Then, operator REDUCE is applied to maximally reduce cubes in turn; the outcome is shown in Part E. In fact, two of the four cubes could be transformed into smaller cubes, yet the collection of cubes still covers all ON-set minterms. Finally, EXPAND is applied a second time. As it consider one of the smaller cubes, it is determined that it can be expanded to overlap the other smaller cube, and thus reducing the size of the cover from four to three. The resulting cover is a minimum cover and no further application of the above operators will change it. □

ESPRESSO-II also employs additional powerful operators, such as ESSENTIALS and LAST_GASP. ESSENTIALS identifies all essential prime implicants before the main loop is entered, in order to simplify the covering problem. LAST_GASP is applied after the main loop is exited, to try to escape a suboptimal local minimum; if successful the main loop is entered again.

One key reason for the efficiency of ESPRESSO-II is the so-called *unate re-*

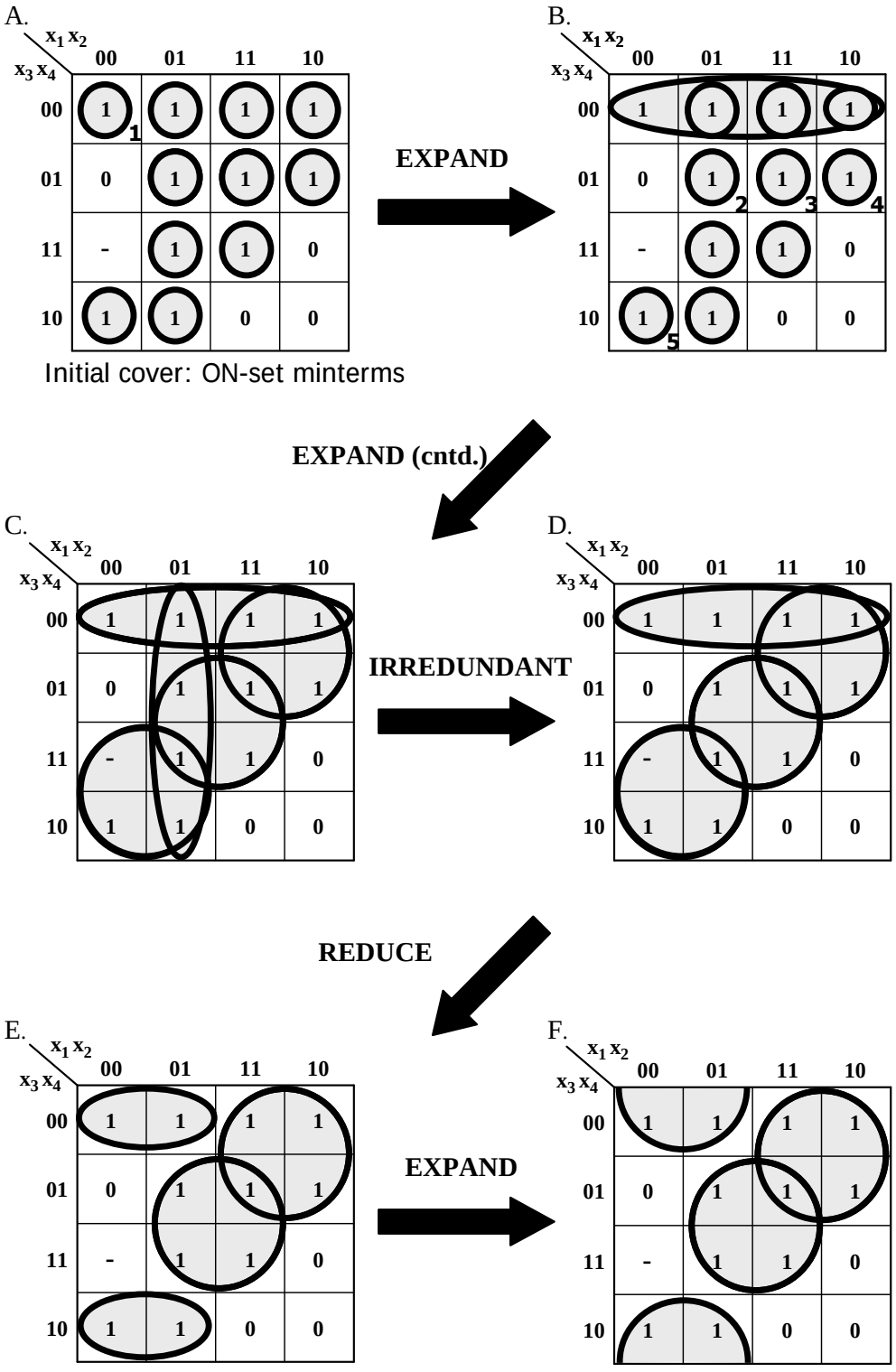


Figure 3.2: ESPRESSO-II Example

cursive paradigm, i.e., to decompose operations recursively leading to efficiently solvable sub-operations on unate functions (cf. Section 2.3.1).

3.3 Overview of ESPRESSO-HF

The goal of heuristic hazard-free minimization is to find a very good (i.e. a small, but not necessarily exactly minimum) solution to the hazard-free covering problem.

The basic minimization strategy of the new algorithm, ESPRESSO-HF for hazard-free minimization is similar to the one used by ESPRESSO-II, i.e. based on the above introduced main operators EXPAND, REDUCE, and IRREDUNDANT. However, additional constraints are imposed to ensure that the resulting cover is hazard-free, and the algorithms that implement the main operators are significantly different. In fact, our algorithms are so different that we decided not to re-use any of the ESPRESSO-II code.

One key distinction is in the use of the *unate recursive paradigm* in ESPRESSO-II, i.e. to decompose operations recursively leading to efficiently solvable sub-operations on unate functions. To the best knowledge of the author, the unate recursive paradigm cannot be applied directly to ESPRESSO-II-like heuristic hazard-free minimization. (In [92, 91], a unate recursive method was proposed, but only for use in the dhf-prime generation step for exact hazard-free minimization; see Section 4.4.3.) The intuitive reason for this is that the operators in ESPRESSO-II manipulate covers. For example, the "many" ON-set minterms (objects-to-be-covered) can typically be stored compactly and manipulated efficiently as an ON-set cover of "a few" cubes. In contrast, in hazard-free synthesis, required cubes cannot be combined into larger cubes without loss of information, which means that the basis

for the unate recursive paradigm, i.e. the concept of covers, becomes obsolete.

Thus, ESPRESSO-HF follows the basic steps of ESPRESSO-II, modified to incorporate hazard-freedom constraints, but without the use of unate recursive algorithms. However, because of the constraints and coarser granularity of the hazard-free minimization problem, high-quality results are still obtained even for large examples.

In this section, the basic steps of the new algorithm are described, concentrating on the new constraints that must be incorporated to guarantee a cover to be hazard-free. Then, the individual steps are described in detail in later sections.

As in ESPRESSO-II, as an invariant, the cover is never increased in size. An additional invariant, after the initialization phase, is that the current cover always represents a *valid solution*, i.e. a cover of f that is also hazard-free. Pseudocode for the new algorithm is shown in Figure 3.3.

The first step of ESPRESSO-HF is to read in PLA files[66] specifying a Boolean function, f, and a set of specified function-hazard-free transitions, T. These inputs are used to generate the set of required cubes Q, the set of privileged cubes P and their corresponding start points S, and the OFF-set R. Generation of these sets is immediate from the earlier lemmas (see also [82])[1].

The set of required cubes Q can be regarded both as an initial cover F of the function, and also as the set of objects to be covered. Unlike ESPRESSO-II, however, the given initial cover Q does *not* in general represent a valid solution: while Q is a cover of f, *it is not necessarily hazard-free.* Therefore, processing begins by transforming this potential hazardous cover into an (unoptimized) hazard-free

[1]The algorithm does not need an explicit cover for the don't-care set because the operations only require the OFF-set to check if a cube is valid.

```
Espresso-HF(f,T)
  Q = generate_set_of_required-cubes(f,T)
  P = generate_set_of_privileged-cubes(f,T)
  S = generate_set_of_start-points(f,T)
  R = OFF-set(f)
  Q^f = {supercube_dhf(q) | q ∈ Q}
  If "undefined" ∈ Q^f then no solution is possible; exit
  Minimize Q^f with respect to single cube containment
  F = Q^f
  (F, E) = expand_and_compute_essentials(F)
  Remove all cubes from Q^f that are already covered by E
  F = F - E
  F = irredundant(F)
  do
     φ2 = |F|
     do
              φ1 = |F|
              F = reduce(F)
              F = expand(F)
              F = irredundant(F)
     while (|F| < φ1)
     F = last_gasp(F)
  while (|F| < φ2)
  F = F ∪ E
  F = make_dh_prime(F)
```

Figure 3.3: The ESPRESSO-HF Algorithm

cover. To do so, each required cube is expanded into the *uniquely defined* minimum dhf-implicant (formalized by the $supercube_{dhf}$ operator) covering it, or the detection that this is impossible, denoted by "undefined". The latter case indicates that the hazard-free minimization problem has no solution (see Section 3.5). Otherwise, the result is an initial hazard-free cover, F, and set of objects to be covered, Q^f.

The next step is to identify all *essential dhf-prime implicants*, i.e. implicants that need to be included in any cover, using a modified EXPAND step. This algorithm uses a novel approach to identifying *equivalence classes* of implicants, each of which is treated as a single implicant. Essential dhf-prime implicants, as well as all required cubes covered by them, are then removed from F and Q^f, respectively, resulting in a smaller problem to be solved by the main loop. Before the main loop, the current cover is also made irredundant.

Next, as in ESPRESSO-II, ESPRESSO-HF enters a loop, and repeatedly applies the three operators REDUCE, EXPAND, and IRREDUNDANT to the current cover until no further improvement in the size of the cover is possible. Since the result may be a suboptimal local minimum, the operator LAST_GASP is then applied to find a better solution using a different method. EXPAND uses new hazard-free notions of *essential parts* and *feasible expansion*. The other steps differ from ESPRESSO-II as well.

In the end, there is a final step to make the resulting implicants dhf-prime, MAKE_DHF_PRIME, since it is desirable to obtain a cover that consists of only dhf-prime implicants. The motivation for this step will be made clear in the sequel.

In addition to the steps shown in Figure 3.3, the implementation has several optional pre- and postprocessing steps.

3.4 Main Steps of ESPRESSO-HF

Now that an overview of the ESPRESSO-HF algorithm has been presented, details of the individual steps are introduced.

3.4.1 Dhf-Canonicalization of Initial Cover

In ESPRESSO-II, the initial cover of a function is provided by an arbitrary set of implicants (e.g. all ON-set minterms, or possibly larger cubes). This cover is an initial seed solution, which is iteratively improved by the algorithm. By analogy, in ESPRESSO-HF, the initial cover is provided by the set of required cubes, Q. However, *unlike* ESPRESSO-II, this initial specification does not in general represent a solution: though Q is a cover, *it is not necessarily hazard-free*. Therefore, processing begins by expanding each required cube into the *uniquely defined* minimum dhf-implicant containing it. This expansion represents a new *canonicalization step*, transforming a potentially hazardous initial cover Q into a hazard-free initial cover Q^f.

Example. Consider the function f in the Karnaugh map of Figure 3.4. A set T of specified multiple-input transitions is indicated by arrows. There are two $1 \rightarrow 0$ transitions, each corresponding to a privileged cube: $p1 = \overline{x}_1\overline{x}_3$ (start point $p1_{start} = \overline{x}_1 x_2 \overline{x}_3 \overline{x}_4$) and $p2 = x_1 x_4$ (start point $p2_{start} = x_1 x_2 \overline{x}_3 x_4$). The initial cover is given by the set Q of required cubes (not shown in the figure): $\{\overline{x}_1\overline{x}_3\overline{x}_4$, $\overline{x}_1 x_2 \overline{x}_3$, $x_1\overline{x}_3$, $x_1\overline{x}_3 x_4$, $x_1 x_2 x_4$, $x_2 x_3 x_4$, $x_2 x_3 \overline{x}_4\}$. This initial cover is hazardous. In particular, consider the required cube $r = x_2 x_3 x_4$ (indicated in the figure), corresponding to the $1 \rightarrow 1$ transition from $x_1 x_2 x_3 x_4 = 0111$ to 1111. Required cube r illegally intersects privileged cube $p2$, since it intersects $p2$ but does not contain

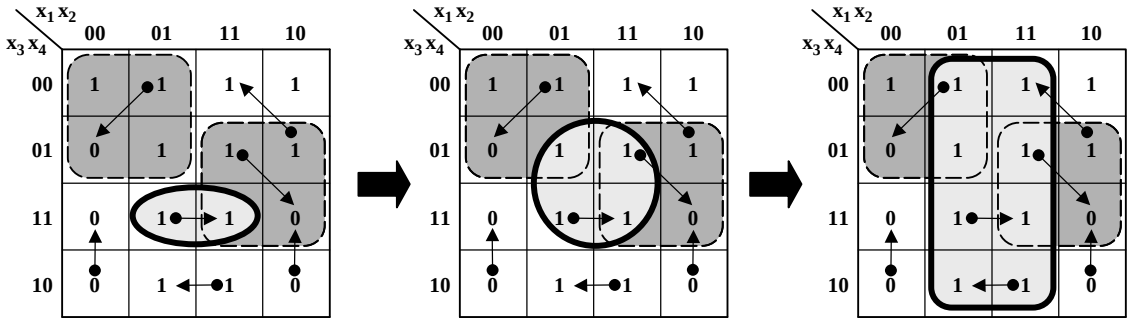


Figure 3.4: Canonicalization Example

$p2_{start}$. To avoid illegal intersection, r must be expanded to the smallest cube which also contains $p2_{start}$: $r^{(1)} = \text{supercube}(\{r, p2_{start}\})$. However, this new cube $r^{(1)} = x_2x_4$ now illegally intersects privileged cube $p1$, since it does not contain $p1_{start}$. Therefore, cube $r^{(1)}$ in turn must be expanded to the smallest cube containing $p1_{start}$: $r^{(2)} = \text{supercube}(\{r^{(1)}, p1_{start}\})$. The resulting expanded cube, $r^{(2)} = x_2$, has no illegal intersections and is therefore a dhf-implicant. □

In this example, $r^{(2)}$ is a hazard-free expansion of r, called a **canonical required cube**; it can therefore replace r in the initial cover. (Note that such a canonicalization is feasible if and only if the hazard-free covering problem has a solution; see Section 3.5.)

It is now shown that each required cube has a *unique* corresponding canonical required cube. Suppose there were two distinct minimal dhf-implicants, q_1 and q_2, each of which contains some required cube r. In this case, a dhf-implicant can be constructed which is smaller than either cube: the intersection of q_1 and q_2, $q_{12} = q_1 \cdot q_2$. Clearly, implicant q_{12} contains r. Furthermore, if q_{12} were not a dhf-implicant, then it would intersect some privileged cube p illegally, i.e. intersect p but not contain its start point p_{start}. However, this would mean that both original

implicants, q_1 and q_2, intersected p, but at least one of them (say q_1) did not contain p_{start}. As a result, q_1 would not be a dhf-implicant, since it would illegally intersect p, thus contradicting our assumption. Therefore, q_{12} is a dhf-implicant which contains r, and so q_1 and q_2 could not have been minimal, thus contradicting the initial assumption. In conclusion, each required cube has a unique corresponding canonical required cube, which contains it.

Based on the above discussion, an initial potentially hazardous set Q of required cubes is replaced by the corresponding hazard-free set Q^f of canonical required cubes. This set is then minimized with respect to single-cube containment [66]. Q^f is a valid hazard-free cover of the function to be minimized, and is therefore used as an *initial cover* for the minimization process in ESPRESSO-HF.

Interestingly, Q^f has a second role as well: it is used as a simplified set of *objects to be covered* in the covering problem. In particular, Q^f defines a new covering problem: each cube of Q^f ($not\, Q$) must be contained in some dhf-implicant. It is straightforward to show that the two covering problems are equivalent, i.e. a dhf-implicant p contains a required cube r in Q *if and only if* p also contains the corresponding canonical required cube of r in Q^f. To see this, suppose that p contained r but did not contain the canonical required cube of r. In this case, p could not be a dhf-implicant, since it must illegally intersect at least one of those privileged cubes that caused r to be expanded into its canonical required cube.

In the above example, any dhf-implicant which contains required cube $r = x_2x_3x_4$ must also contain canonical required cube $r^{(2)} = x_2$. Therefore, the hazard-free minimization problem is unchanged, but canonical required cubes are used. An advantage of using Q^f is that it may have smaller size than Q, i.e. being a more

$supercube_{dhf}$ (set of cubes $C = \{c_1, \ldots, c_n\}$)
    $r = supercube(\{c_1, \ldots, c_n\})$
    while (r intersects some privileged cube p_i illegally)
        $r = supercube(\{r, s_i\})$ where s_i is the start point of p_i
    if r intersects the OFF-set then return "undefined" else return r

Figure 3.5: $Supercube_{dhf}$ Computation

efficient representation of the problem. Also, since the cubes in Q^f are in general larger than the corresponding ones in Q, the EXPAND operation may be sped up.

To conclude, the new set of canonical required cubes Q^f replaces the original set of required cubes Q as both (i) the initial cover, and (ii) the set of objects to be covered. Henceforth, the term "set of required cubes" will be used to refer to set Q^f.

Now, a formalization of the above notion of canonicalization is given.

Definition 3.1 *The* **dhf-supercube** *of a set of cubes C with respect to function f and transitions T, indicated as $supercube_{dhf}^{(f,T)}(C)$, is the smallest dhf-implicant containing the cubes of C.*

The superscript (f, T) is omitted when it is clear from the context. The computation of $supercube_{dhf}(C)$ is achieved by the simple algorithm shown in Figure 3.5. Note that a dhf-implicant that contains every cube in C may not exist.

The *canonical required cube* of a required cube r can now be defined as the *dhf-supercube* of the set $C = \{r\}$. The computation of dhf-supercubes for larger sets will be needed to implement some of the operators presented in the sequel.

3.4.2 Expand

As in ESPRESSO-II, one of the most powerful steps is EXPAND. In ESPRESSO-II, the goal of EXPAND was to enlarge each implicant of the current cover in turn into a prime implicant. As an implicant is expanded, it may contain other implicants of the cover which can be removed, and hence the cover's cardinality is reduced. If the current implicant cannot be expanded to contain another implicant completely, then, as a secondary goal, the implicant is expanded *to overlap* as many other implicants of the current cover as possible.

In ESPRESSO-HF, the primary goal is somewhat similar: to expand a dhf-implicant of the current cover to contain as many other dhf-implicants of the cover as possible. However, EXPAND in ESPRESSO-HF has two major differences. First, unlike ESPRESSO-II, expansion in some literal (*i.e.,* "raising of entries") may *imply* that other expansions be performed. That is, raising of entries is now a *binate problem* [66], i.e. a selection may *imply* other selections, not a unate problem. Second, ESPRESSO-HF's EXPAND uses a different strategy for its secondary goal. By the Hazard-Free Covering Theorem, introduced in Section 2.4.5, each required cube needs to be contained in some cube of the cover. Therefore, as a secondary goal, an implicant is expanded *to contain as many required cubes as possible.*

In the remainder of this section, the implementation of the EXPAND operator in ESPRESSO-HF is described. Pseudocode for the expansion of a single cube is shown in Figure 3.6. This same step is applied in turn to each cube in the cover.

Expand_cube(cube a, req-set Q^f, priv-set P, cover-set F, OFF-set R)

$F_a = F - a$
$Q_a = Q^f$
$P_a = P$
$R_a = R$
$free\_entries$ =complement_pos_cube_notation(a)
while ($F_a \neq \oslash$)
    update($a, free\_entries, F_a, Q_a, P_a, R_a$)
    $F_a = \{c \in F_a | supercube_{dhf}(\{a, c\})$ is defined $\}$
    Let c_b be the best candidate in F_a
    $a = supercube_{dhf}(\{a, c_b\})$
    $F_a = single\text{-}cube\text{-}contain\{F_a \cup \{a\}\} - \{a\}$
while ($Q_a \neq \oslash$)
    update($a, free\_entries, F_a, Q_a, P_a, R_a$)
    $Q_a = \{q \in Q_a | supercube_{dhf}(\{a, q\})$ is defined $\}$
    Let q_b be the best candidate in Q_a
    $a = supercube_{dhf}(\{a, q_b\})$
    $Q_a = single\text{-}cube\text{-}contain\{Q_a \cup \{a\}\} - \{a\}$

Figure 3.6: Expand (for a cube a)

3.4.2.1 Determination of Essential Parts and Update of Local Sets

As in ESPRESSO-II, a list of *free entries* is maintained, to accelerate the expansion [88], but these entries are defined and used differently in ESPRESSO-HF. Free entries indicate which literals of an implicant are candidates for removal, i.e. potential directions for expansion during the expansion process.

To explain this concept in a unified way, the current implicant and its free entries are represented in positional cube notation [88]. As an example, $x_1\overline{x_2}x_4$ is represented in positional cube notation as 01 10 11 01, i.e. where literal $x_i(\overline{x_i})$ is encoded as 01 (10). Thus, each literal in $x_1\overline{x_2}x_4$ has a corresponding 0 in the positional cube notation, and changing, or *raising*, the 0 to 1 corresponds to removing this literal from the implicant.

Initially, a free entry is assigned a 1 (0) if the current implicant to be expanded, a, has a 0 (1) in the corresponding position. For the above example, the free entries are: 10 01 00 10. An *overexpanded cube* is defined as the cube a where all its free entries have been raised simultaneously.

As in [88], an *essential part* is a free entry which can never, or always, be raised. However, our definition of essential parts is different from ESPRESSO-II, since a hazard-free cover must always be maintained. Essential parts are determined in procedure *update*, described below.

First, as an optimization, entries are determined that can *never be raised* and removed them from $free\_entries$. This is achieved by searching for any cube in the OFF-set R that has distance 1 from a, in this entry, using the same approach as in ESPRESSO-II.

Next, parts that can *always be raised* are determined, raised and removed from $free\_entries$. This step differs from ESPRESSO-II. In ESPRESSO-II, a part can always be raised if it is 0 in all cubes of the OFF-set, R. That is, it is guaranteed that the expanded cube will never intersect the OFF-set. In contrast, in ESPRESSO-HF, it must be ensured that the resulting expanded implicant is also *hazard-free*: it cannot intersect the OFF-set, nor can it illegally intersect a privileged cube. Unlike ESPRESSO-II, this is achieved by searching for any column that (i) has only 0s in R *and* (ii) where, for each privileged cube p in P having a 1 in this column, the corresponding start point p_{start} will be contained by the expanded cube a.

Example. Figure 3.7(top left) shows the set of required cubes, which forms an initial hazard-free cover. Consider the cube $x_2x_3x_4$ (11010101, in positional cube notation). As in ESPRESSO-II, the 0-entries for literals $\overline{x}_2$ and $\overline{x}_4$ can never

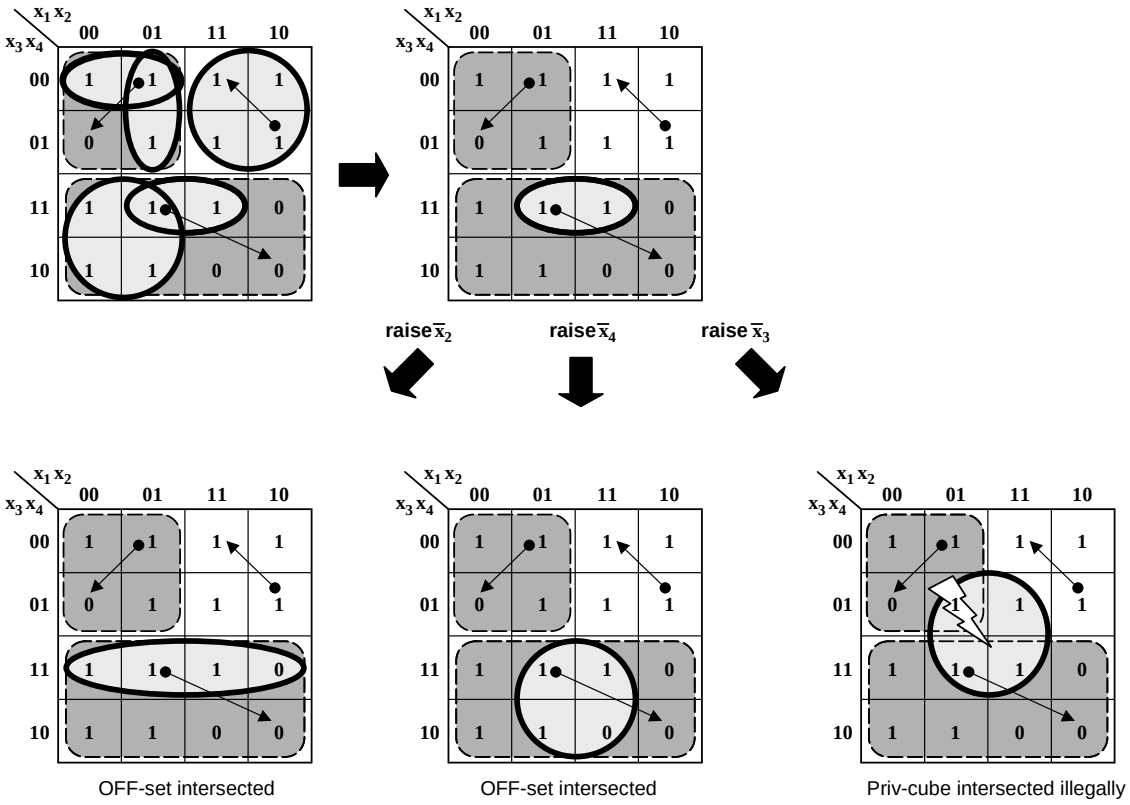


Figure 3.7: Essential Parts Example

be raised, since the cube would intersect the OFF-set. However, after updating the free entries, ESPRESSO-II would indicate that literal $\overline{x}_3$ can *always* be raised, since the resulting cube will never intersect the OFF-set. In contrast, in ESPRESSO-HF, raising $\overline{x}_3$ results in an *illegal intersection* with privileged cube $\overline{x}_1\overline{x}_3$, so it cannot "always be raised". □

Since hazard-free minimization is somewhat more constrained, the expansion of a cube a can be accelerated by the following new operations on two local sets: P_a, R_a. These sets are associated with cube a, and P_a (R_a) is initially assigned the set of privileged (OFF-set) cubes. Both sets are *updated* as expansion proceeds (in procedure *update*).

1. Remove privileged cubes from P_a where the corresponding start point is already covered by a (since no further checking for illegal intersection is required).

2. Move privileged cubes from set P_a to the local OFF-set R_a if the overexpanded cube does not include the corresponding start points (since a can never be expanded to include these start points, therefore one must avoid intersection with these privileged cubes entirely).

3. Move privileged cubes from P_a to the local OFF-set R_a where $supercube_{dhf}$({a, start point}) intersects the OFF-set (a can never be expanded to include these start points, therefore one must avoid intersection with the cubes entirely).

3.4.2.2 Detection of Feasibly Covered Cubes of F

In ESPRESSO-II, a cube a in F is expanded to cover another cube d through a *supercube* operation. A cube d in F is said to be *feasibly covered* by cube a if supercube($\{a,d\}$) is an implicant. In ESPRESSO-HF, this definition needs to be modified to insure that the resulting cube is a *dhf-implicant*, i.e. allows *hazard-free covering.*

Definition 3.2 *A cube d in F is* **dhf-feasibly covered** *by a if $supercube_{dhf}(\{a,d\})$ is defined.*

This definition insures that the resulting expanded cube, $supercube_{dhf}(\{a,d\})$, is (i) an implicant (does not intersect OFF-set), and (ii) is also a dhf-implicant (does not intersect any privileged cube illegally). Effectively, this definition canonicalizes the resulting supercube to produce a dhf-implicant. That is, $supercube_{dhf}(\{a,d\})$

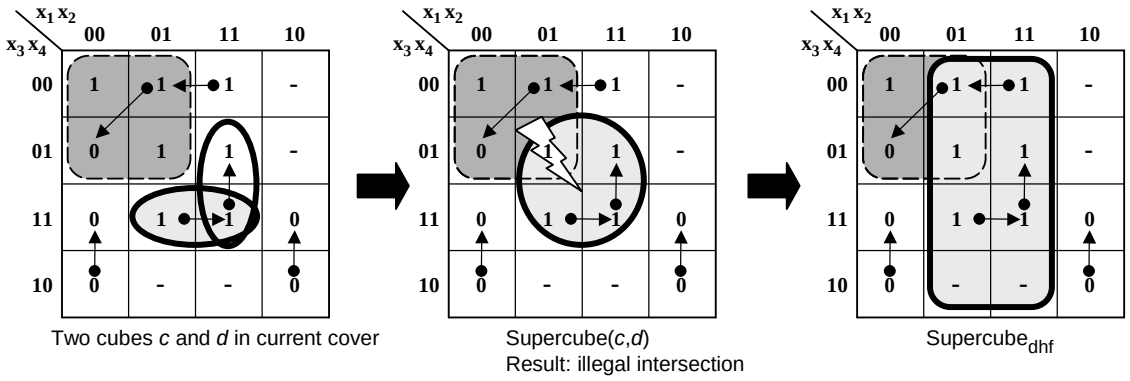


Figure 3.8: Expand Example

may properly contain supercube($\{a$,$d\}$), since the former may be expanded through a series of implications in order to reach the unique minimum dhf-implicant which contains both a and d. Using this definition, the following is an algorithm to find dhf-feasibly covered cubes of F. An example is given in Figure 3.8.

While there are cubes in F that are dhf-feasibly covered by cube a, iterate the following:

> Replace a by $supercube_{dhf}(\{a, d\})$, where d is a dhf-feasibly covered cube such that the resulting cube will cover as many cubes of the cover as possible. Covered cubes are then removed, using the "single-cube-containment" operator, thus reducing the cover cardinality. Determine essential parts and update local sets (see above).

3.4.2.3 Detection of Feasibly Covered Cubes of Q^f

The expansion of cube a still continues even after cube a can no longer be feasibly expanded to cover entirely any other cube d of F. This is motivated by the Hazard-Free Covering Theorem (cf. Section 2.4.5), which states that each required cube

needs to be contained in some cube of the cover. Thus, *as a secondary goal*, cube a is expanded to contain as many required cubes as possible. The strategy used in this sub-step is similar to the one used in the preceding one, i.e. while there are cubes in Q^f that are dhf-feasibly covered by cube a, iterate the following:

> Replace a by $supercube_{dhf}(\{a, q\})$, where q is a dhf-feasibly covered *required cube* such that the resulting cube will cover as many required cubes not already contained in a as possible. Covered required cubes are then removed, using the "single-cube-containment" operator. Determine essential parts and update local sets (see above).

3.4.2.4 Constraints on Hazard-Free Expansion

In ESPRESSO-II, an implicant is expanded until no further expansion is possible, i.e. until the implicant is prime. Two steps are used: (i) expansion to overlap a maximum number of cubes still covered by the overexpanded cube; and (ii) raising of entries to find the largest prime implicant covering the cube.

In ESPRESSO-HF, however, these remaining EXPAND steps are not implemented, based on the following observation. The result of the new EXPAND steps (cf. 3.4.2.2 and 3.4.2.3) guarantees that a dhf-implicant can never be further expanded to contain additional required cubes. Therefore, by the Hazard-Free Covering Theorem, no additional objects (required cubes) can possibly be covered through further expansion. In contrast, in ESPRESSO-II, expansion steps (i) and (ii) may be beneficial, since they can result in covering of additional ON-set minterms. Because of this distinction, the benefit of further expansion is mitigated. Therefore, in general, the new EXPAND algorithm makes no attempt to

transform dhf-implicants into dhf-prime implicants. However, since expansion to dhf-primes is important for literal reduction and testability, it is included as a final post-processing step: MAKE_DHF_PRIME (see Figure 3.3).

3.4.3 Essentials

Essential prime implicants are prime implicants that need to be included in any cover of prime implicants. Therefore, it is desirable to identify them as soon as possible to make the resulting problem size smaller.

In this section, a novel characterization of essential dhf-prime implicants is proposed. This formulation significantly accelerates the minimization process. In fact, the hazard-free minimization problem is highly constrained by the notion of covering of *required cubes*, allowing a powerful new method to classify essentials as *equivalence classes*. The idea is now illustrated by an example.

Example. Consider Figure 3.9. The required cube, $r = x_2x_3x_4$ (shaded), is covered by precisely two dhf-prime implicants: $p1 = x_2x_4$ and $p2 = x_3x_4$, which cover no other required cubes. Neither $p1$ nor $p2$ is an essential dhf-prime, since r is covered by both. And yet, clearly, either $p1$ or $p2$ (not both) *must* be included in any cover of dhf-primes. Also, assuming the standard cost function of cover cardinality, $p1$ and $p2$ are of equal cost. □

Definition 3.3 *Two dhf-prime implicants are* **equivalent** *if they cover the same set of required cubes. An equivalence class of dhf-prime implicants is* **maximal** *if it covers a set of required cubes which is not covered by any other equivalence class. A maximal equivalence class of dhf-prime implicants is* **essential** *if they cover at least one required cube which is not covered by any other maximal equivalence class.*

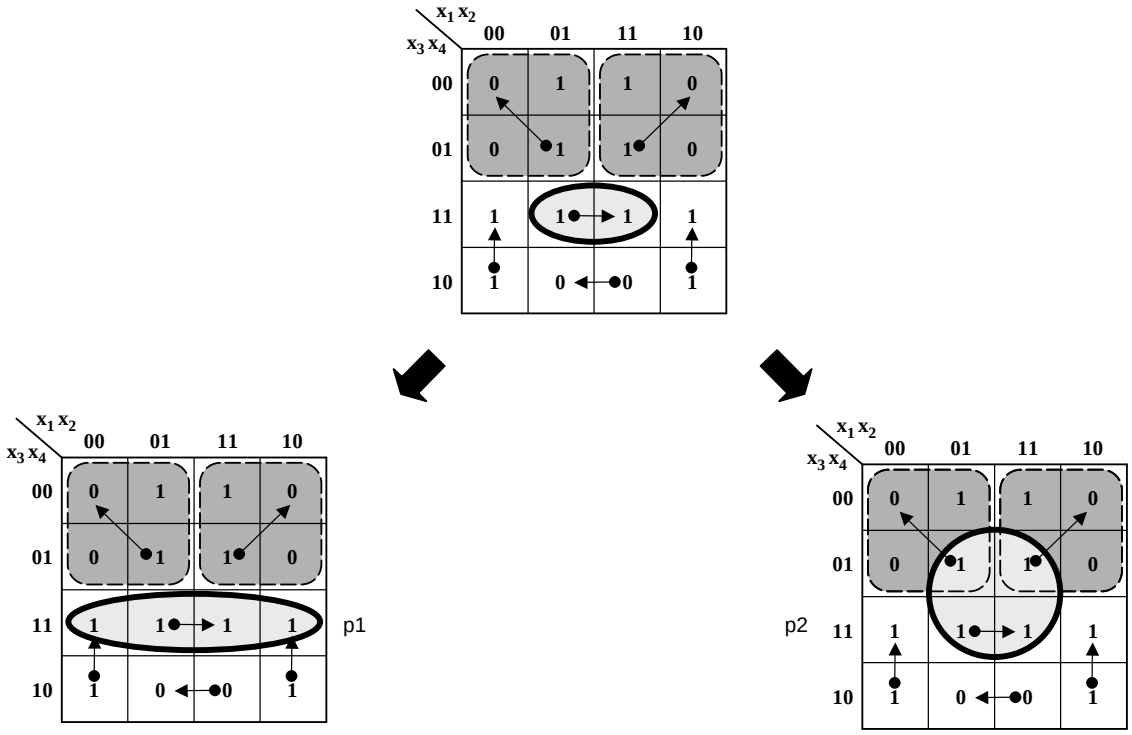


Figure 3.9: Essential Example

In the above example, the set $\{x_2x_4, x_3x_4\}$ of dhf-primes is an *equivalence class*, since both dhf-primes cover the same set of required cubes $\{x_2x_3x_4\}$. In fact, the class is an *essential equivalence class*, since it is the only equivalence class that covers the required cube $x_2x_3x_4$.

ESPRESSO-II computes essentials after the initial EXPAND and IRREDUNDANT steps. In contrast, ESPRESSO-HF computes essentials as part of a modified EXPAND step. The new algorithm is outlined as follows:

The algorithm starts with the initial hazard-free cover, Q^f, of required cubes. To simplify the presentation, assume that one seed cube is selected and expanded greedily, using EXPAND, to a dhf-implicant p. This implicant is characterized by the set, Q^p, of required cubes which it contains. Dhf-implicant p thus corresponds to the equivalence class of dhf-primes that cover Q^p. Since EXPAND guarantees that p covers a maximal number of required cubes, this equivalence class is also maximal. Moreover, this class is an *essential equivalence class* if Q^p contains some required cube, q^f, which cannot be expanded into any other maximal equivalence class.

To check if q^f can be expanded into a different maximal equivalence class, a simple pairwise check is used: for each required cube s^f not covered by p, determine if $supercube_{dhf}(\{q^f, s^f\})$ is feasible. If no such feasible expansion exists for q^f, q^f is called a **distinguished required cube**, and therefore the equivalence class corresponding to p is essential. Otherwise, the process is repeated for every required cube q^f contained in Q^p. If p corresponds to an essential equivalence class, then p is removed from the cover. In addition, all required cubes covered by p are removed, since it is ensured that they will be covered. This step can result in "secondary

essential" equivalence classes. In fact, due to the removal of required cubes, more dhf-prime implicants become equivalent to each other. As a consequence, further equivalence classes may become essential.

The procedure iterates until all essentials are identified.

The above discussion seems to imply that the essentials step is more or less quadratic in the number of required cubes, i.e. very inefficient. However, by making use of techniques similar to the ones described in the EXPAND section 3.4.2, e.g. by using an overexpanded cube, the number of necessary $supercube_{dhf}$-calls can be reduced dramatically. Therefore, in practice, essentials can be identified efficiently. In fact, this new concept of *essential equivalence classes* can dramatically reduce the minimization problem complexity and has a very positive impact on both run-time and quality of solution (see Section 3.6).

3.4.4 Reduce

The goal of the REDUCE operator is to set up a cover that is likely to be improved, i.e. decreased in size, by the following EXPAND step. In the REDUCE step, to achieve this goal, each cube c in a cover F is maximally reduced, in turn, to a cube $\tilde{c}$, such that the resulting set of cubes, $\{F - c\} \cup \tilde{c}$ is still a cover.

ESPRESSO-II uses the unate recursive paradigm to maximally reduce each cube. Since ESPRESSO-HF is a required-cube covering algorithm, there is no obvious way to use this paradigm. Fortunately, the hazard-free problem is more constrained, making it possible to use an efficient enumerative approach based on required cubes.

The new REDUCE algorithm is as follows. The algorithm reduces each

cube c in the cover in order. In particular, a cube c is reduced to the smallest dhf-implicant $\tilde{c}$ that covers all required cubes that are *uniquely covered* by c (i.e. contained in no other cube of the cover F). That is, if $r_1, \ldots, r_l$ is the set of required cubes that are uniquely covered by c, then c is replaced by $\tilde{c} = supercube_{dhf}(\{r_1, \ldots, r_l\})$.

Note that the outcome of this algorithm depends on the order in which the cubes c of the cover F are processed. Suppose c_i is reduced before c_j, and that c_i and c_j cover some required cube r but no other cube of F covers r. If c_i is reduced to a cube $\tilde{c_i}$ that does not cover r, then c_j cannot be reduced to a cube that does not cover r.

3.4.5 Irredundant

ESPRESSO-II uses the unate recursive paradigm to find an irredundant cover. However, in ESPRESSO-HF, the same algorithm cannot be employed, since a "redundant cover" (according to covering of minterms) may in fact be irredundant with respect to covering of required cubes.

Therefore, as in REDUCE, the new approach is required-cube based. Considering the Hazard-Free Covering Theorem, it is straightforward that IRREDUNDANT can be reduced to a covering problem of the cubes in Q^f by the cubes in F. That is, the problem reduces to a minimum-covering problem of (i) required cubes, using (ii) dhf-implicants in the current cover. In practice, the number of required cubes and cover cubes usually make the covering problem manageable. ESPRESSO-II's MINCOV can be used to solve this covering problem exactly, or heuristically (using its heuristic option).

3.4.6 Last Gasp

The inner loop of ESPRESSO-HF may lead to a suboptimal local minimum. In this case, a loop iteration will result in no improvement to the cover, but the current solution is sub-optimal. LAST_GASP is a powerful step, introduced in ESPRESSO-II, to radically transform the cover in order to avoid such local minima, and thus to allow further iterative improvement.

In ESPRESSO-II, LAST_GASP proceeds as follows. Each cube $c \in F$ is independently reduced to the smallest cube containing all ON-set minterms not covered by any other cube of F. Cubes that can actually be reduced by this process are added to an initially empty set G. Each such $g \in G$ is then expanded in turn with the goal to cover at least one other cube of G, and, if achieved, the expanded cube is added to F. Finally, the IRREDUNDANT operator is applied to F with the hope to escape the above-mentioned local minimum.

In contrast, ESPRESSO-HF employs a required cube based approach. The new operator proceeds as follows. First, it computes for each $c \in F$, the smallest dhf-implicant containing all *required cubes* covered by c that are not covered by any other cube in F. The remaining strategy is then identical to ESPRESSO-II, i.e. adding reduced cubes to a set G, expanding each cube in G, and applying the IRREDUNDANT operator.

3.4.7 Make-dhf-Prime

The cover being constructed so far does not necessarily consist of dhf-primes, but only of dynamic hazard-free implicants. It is usually desirable eventually to expand each dhf-implicant of the cover to make it dhf-prime as a last step. This can be

achieved by a modified EXPAND step. A simple greedy algorithm will expand an implicant c to a dhf-prime: While dhf-feasible, raise a single entry of c.

3.4.8 Pre- and Postprocessing Steps

ESPRESSO-HF includes optional pre- and postprocessing steps. In particular, the efficiency of ESPRESSO-HF depends very much on the size of the ON-set and OFF-set covers that are given to it. Thus, ESPRESSO-HF includes an optional *preprocessing* step which uses ESPRESSO-II to find covers of smaller size for the initial ON-set and OFF-set[2]. ESPRESSO-HF also includes a *postprocessing* step to reduce the literal count of a cover, similar to ESPRESSO-II's MAKE_SPARSE.

3.5 Existence of a Hazard-Free Solution

As indicated earlier, for certain Boolean functions and sets of transitions, no hazard-free cover exists [82]. In many applications, it is important to be able to determine quickly if a solution exists.

The currently-used exact hazard-free minimization method HFMIN [40] is only able to decide if a hazard-free solution exists *after generating all dhf-prime implicants*. A hazard-free solution does not exist if and only if the dhf-prime implicant table includes at least one required cube not covered by any dhf-prime implicant [82]. However, since the generation of all dhf-primes may very well be infeasible[3] for even medium-sized examples, it is important to find a more efficient

[2]ON-set and OFF-set are necessary to form the initial set of required cubes, Q. More importantly, the OFF-set is used to check if a cube expansion is valid, see Figure 3.5.

[3]This refers to "explicit representations"; we will show later that "implicit representations" very often are feasible.

approach.

A new theorem is now introduced to check for the existence of a hazard-free solution, without the need to generate all dhf-prime implicants. This theorem leads directly to a fast and simple algorithm that is incorporated into ESPRESSO-HF.

Theorem 3.4 *Given a function f and a set, T, of specified function-hazard-free input transitions of f, a solution of the two-level hazard-free logic minimization problem exists if and only if $supercube_{dhf}(q)$ is defined for each required cube q.*

The proof is immediate from the discussion in Section 3.4.1.

Example. Consider the left Karnaugh map in Figure 3.10. To check for existence of a hazard-free solution, consider each required cube in turn, and try to transform it into the uniquely-defined minimum dhf-implicant which covers it. That is, $supercube_{dhf}(q)$ is computed for each required cube q. Consider the required cube $x_1x_2x_4$ (the highlighted cube in the middle of the left Karnaugh map). To compute $supercube_{dhf}(x_1x_2x_4)$, note that privileged cube x_3 (the cube in the lower half of the Karnaugh map) is intersected illegally, and therefore the required cube is not a dhf-implicant. Thus, the idea is now to expand the required cube to also include the start points of the privileged cube that are intersected illegally. Formally, $supercube_{dhf}(x_1x_2x_4) = supercube_{dhf}(x_2x_4)$.

The second Karnaugh map shows the expanded required cube x_2x_4. However, this cube again intersects a privileged cube (cube $\overline{x}_1\overline{x}_3$ - the cube in the upper left) illegally. Thus, it needs to be expanded again to also include the corresponding start point. Formally, we get $supercube_{dhf}(x_1x_2x_4) = supercube_{dhf}(x_2x_4) = supercube_{dhf}(x_2)$.

The third Karnaugh map shows this expanded cube x_2. However, this cube

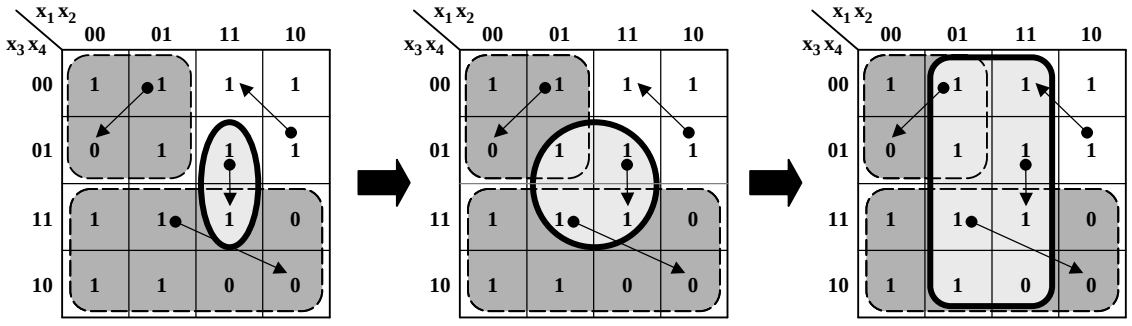


Figure 3.10: Existence Example

intersects the OFF-set, i.e. is not a legal implicant. In sum, it can be concluded that there is no dhf-implicant that covers the required cube $x_1x_2x_4$. Formally, it has been determined that $supercube_{dhf}(x_1x_2x_4)$ is "undefined". Thus, a hazard-free cover does not exist for this example. □

In conclusion, the new efficient check for existence of a solution is based on a simple iteration: each required cube in turn is expanded to see if it is contained in *some* dhf-implicant. As a result, there is no need to generate all dhf-prime implicants.

3.6 Results

A prototype version of the ESPRESSO-HF algorithm has been implemented in C++. The new algorithms are so different that it was decided not to re-use any of the original ESPRESSO-II code.

The ESPRESSO-HF tool was run on several well-known benchmark circuits [37] on an ULTRA-SPARC 140 workstation (Memory: 89 MB real/ 230 MB virtual), as shown in Figure 3.11. The results for the exact columns were obtained by running the exact hazard-free minimizer by Fuhrer/Lin/Nowick [37], called HFMIN.

This method uses ESPRESSO-II to generate all prime implicants, then transforms them into dhf-prime implicants, and finally employs MINCOV to solve the resulting unate covering problem. Each of the algorithms used in the three steps is critical, i.e. has a worst-case run-time that is exponential.

Two of the fifteen examples could not be solved by the exact minimizer HFMIN (in the allotted time of 20 hours). For *stetson-p1*, the generation of all prime implicants was not possible. For *cache-ctrl*, the exact minimizer HFMIN was unable to transform the set of prime implicants into the set of dhf-prime implicants.

The new heuristic minimizer ESPRESSO-HF was able to solve all examples. For all but one example that could be solved by the exact minimizer HFMIN, ESPRESSO-HF finds the optimal solution. It is worth pointing out that many examples were very positively influenced by the new notion of essential equivalence classes. Quite a few examples can be minimized by just the essential step, resulting in a guaranteed minimal solution. The detection of essentials is crucial for speed and size.

3.7 Conclusions

To the best knowledge of the author, the presented algorithm ESPRESSO-HF is the first heuristic method based on ESPRESSO-II to solve the hazard-free minimization problem in the presence of multiple-input changes. The new tool can solve all examples that are available so far, and almost always obtains an absolute minimum-size cover. This includes examples that have not been solved before.

In practice, ESPRESSO-HF overcomes the three bottlenecks of the exact method — prime implicant generation, transformation of prime implicants to dhf-

		HFMIN			ESPRESSO-HF		
name	*i/o*	*#p*	*#c*	*time*	*#e*	*#c*	*time*
cache-ctrl	20/23	*	97	impossible	50	99	105
dram-ctrl	9/8	45	22	1	22	22	1
pe-send-ifc	12/10	454	27	9	27	27	1
pscsi-ircv	8/7	20	12	1	12	12	1
pscsi-isend	11/10	204	23	3	23	23	1
pscsi-pscsi	16/11	65060	77	1656	55	78	11
pscsi-tsend	11/10	190	22	3	22	22	1
pscsi-tsend-bm	11/11	188	23	3	23	23	1
sd-control	18/22	1718	34	172	23	35	3
sscsi-isend-bm	10/9	87	22	1	22	22	1
sscsi-trcv-bm	10/9	113	24	1	21	24	1
sscsi-tsend-bm	11/10	93	20	2	20	20	1
stetson-p1	32/33	*	60	> 72000	34	60	21
stetson-p2	18/22	1574	37	151	26	37	2
stetson-p3	6/4	10	7	1	7	7	1

Figure 3.11: Comparison of the Heuristic Hazard-Free Minimizer ESPRESSO-HF with the Exact Hazard-Free Minimizer HFMIN. (#p - number of dhf-prime implicants, #c - number of cubes in solution, time - run-time in seconds, #e - number of essential equivalence classes, * - stopped due to exceeded time or memory limits)

prime implicants, and solution of the covering problem — each of which being solved by an algorithm with exponential worst-case behavior.

ESPRESSO-HF also employs a new and much more efficient algorithm to check for existence without generating all prime implicants.

Chapter 4

Exact Implicit Hazard-Free Logic Minimization: IMPYMIN

After having discussed a new *heuristic hazard-free two-level minimizer* in the previous chapter, this chapter focuses on the second contribution of the thesis: an *exact hazard-free two-level minimizer*. Thus, the goal of such a minimizer is to find a minimum-cost solution, and it is used whenever run-times are reasonable.

4.1 Introduction

Current exact minimizers can only handle small- and medium-size examples. In contrast, the new minimization method, called IMPYMIN, can solve much more complex benchmarks.

IMPYMIN uses a so-called *implicit* approach, where the use of compact data structures such as BDDs [15] and zero-suppressed BDDs [67] eliminates the need to explicitly represent each object of interest (e.g. prime implicants, minterms).

The algorithm is based on a novel theoretical approach to hazard-free two-level logic minimization. In particular, the problem of generating dynamic-hazard-free prime implicants is reformulated as a *synchronous* prime implicant generation problem. This is achieved by capturing hazard-freedom constraints within a constructed auxiliary function by adding new variables. Once this new function is constructed, prime implicants are generated using existing prime generation techniques, and the dhf-primes are extracted from these synchronous primes. Using the new formulation, a very efficient implicit minimizer for hazard-free logic can be built. In particular, the new formulation makes it possible to use the implicit set covering solver of SCHERZO [25], the state-of-the-art minimization method for synchronous two-level logic, as a black box. IMPYMIN can find a minimum-size cover for all benchmark examples in less than 813 seconds.

The reminder of this chapter is structured as follows. In Section 4.2, a novel framework is proposed, which recasts the dhf-prime implicant generation problem into a prime generation problem for a new auxiliary *synchronous* function, with extra inputs. Based on this approach, Section 4.3 presents a new implicit method for exact 2-level hazard-free logic minimization. Experimental results and comparisons with related work are given in Section 4.4, and Section 4.5 presents conclusions.

4.2 A Novel Approach to Incorporating Hazard-Freedom Constraints Within a Synchronous Function

4.2.1 Overview and Intuition

This subsection gives a simple overview of the proposed approach. Formal definitions and further details are provided in the remaining subsections.

The approach is to recast the generation of dhf-prime implicants of an *asynchronous* function (f, T) into the generation of prime implicants of a *synchronous* function g. Here, hazard-freedom constraints are directly incorporated into the function g by adding extra inputs. (The exact definition of g is given in 4.2.2.) An overview of the method is best illustrated by a simple example.

Example. Consider Figure 4.1. The Karnaugh map in part A represents a function (f, T) defined over the set of 3 variables $\{x_1, x_2, x_3\}$. f is the function and, T is the set of specified transitions. The hazard-free two-level logic minimization problem is to find a minimum-cost cover of function f which is hazard-free for each of the specified transitions in T. The shaded area corresponds to the only non-trivial privileged cube of f (the second privileged cube $[101, 100]$ is trivial, cf. Section 2.4.4).

Now, a new *synchronous* function g is defined, shown in part B. g is obtained from f by adding a single new variable z_1. That is, g is defined over 4 variables: $\{x_1, x_2, x_3, z_1\}$. In general, to generate g, one new z-variable is added corresponding to each non-trivial privileged cube. (The exact details will be presented later.) The

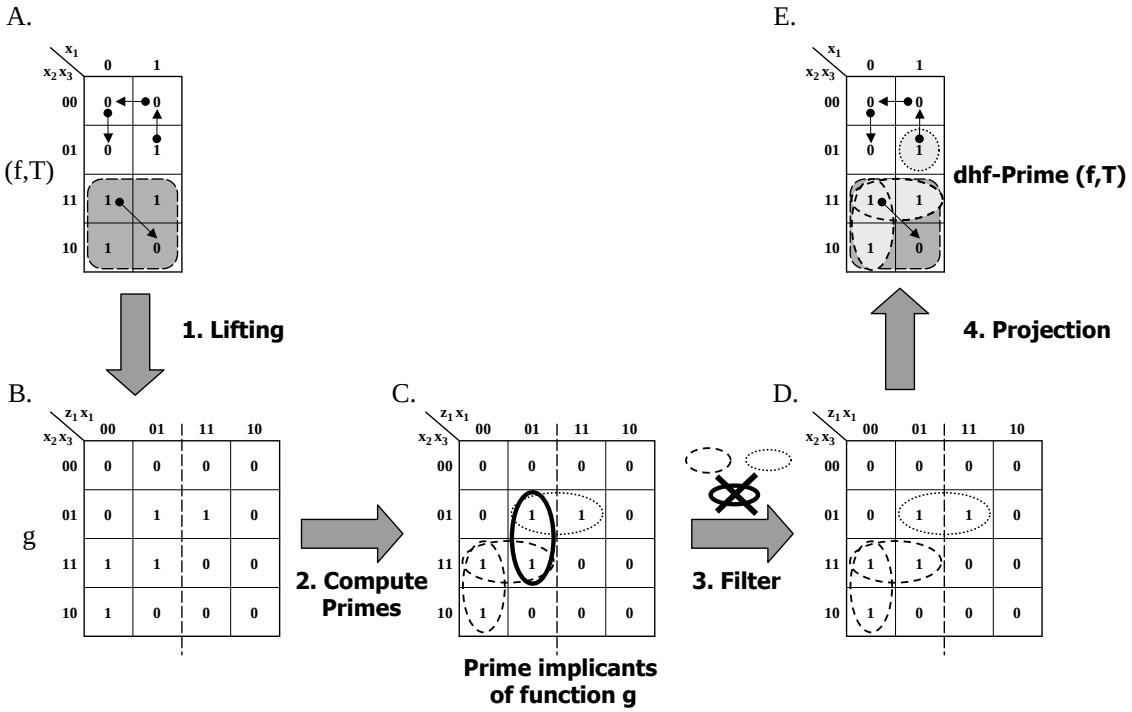


Figure 4.1: Example for Recasting Prime Generation. A) shows the function (f, T) whose dhf-primes are to be computed. B) shows the auxiliary synchronous function g, and C) shows the prime implicants of g. D) shows primes of g that do not intersect illegally. E) shows the final dhf-primes of f, after deleting the z_1 variable.

set of transitions T can now be disregarded. Next, the standard prime implicants of the synchronous function g are computed, and shown in part C as ovals. These primes of function g are closely related to the desired dhf-primes of the original function f. Then, a simple procedure is used to filter out those prime implicants that should be discarded, i.e. which correspond to those in f which intersect a privileged cube illegally. Finally, the remaining prime implicants of g are shown in part D. This set corresponds directly to the set of dhf-primes. In fact, "deleting" the z_1-dimension from the prime implicants generates the set of dhf-prime implicants of (f, T) (part E). □

Intuitively, the formulation is motivated by the fact that dhf-prime-implicants

are more constrained than prime implicants of the same function. While prime implicants are maximal implicants that do not intersect the OFF-set of the given function, dhf-prime-implicants, in addition, *must also not intersect privileged cubes illegally.* This means that there are two different kinds of constraints for dhf-prime-implicants: "maximality" constraints and "avoidance of illegal intersections" constraints. The idea is to unify these two types of constraints. That is, *avoidance constraints* are transformed into *maximality constraints*, so that dhf-primes can be generated directly from a synchronous function. Intuitively, this unification can be achieved by adding auxiliary variables, i.e. by "lifting" the problem into a higher-dimensional Boolean space.

In summary, the big picture is as follows. The definition of the auxiliary function g ensures that all dhf-prime implicants of f (*dhf-Prime(f,T)*) can be easily obtained from the set of prime implicants of g ($Prime(g)$). While $Prime(g)$ may also include certain products which are non-hazard-free, these are filtered out easily, using a post-processing step.

The remaining subsections now present each of the steps of Figure 4.1 in detail, and then conclude with a mathematical formulation of the concept.

4.2.2 The Auxiliary Synchronous Function g

This section explains how the synchronous function g is derived from an asynchronous function f with set of specified input transitions T (denoted (f, T)). For simplicity, assume for now that f is a single-output function.

The main concept behind the definition of auxiliary function g is illustrated in Figure 4.2. When g is formed, one new dimension is added for each privileged

cube. In the 0-half-space (of the new dimension) g is defined as f and in the 1-half-space g is defined as f with one slight modification: the area that corresponds to the privileged cube is filled in with 0's. Note that in this section the focus is on *how* the auxiliary function g is defined. The motivation behind the definition of function g will be explained in detail in the next section.

In general, suppose function f is defined over the set of variables $\{x_1, \ldots, x_n\}$, and that the set of transitions T gives rise to the set of non-trivial privileged cubes $PRIV(f,T) = \{p_1, \ldots, p_l\}$. The idea is to define an auxiliary function g over $\{x_1, \ldots, x_n, z_1, \ldots z_l\}$; that is, *one new variable is added per privileged cube.* Formally, g is defined as follows:

$$g(x_1, \ldots, x_n, z_1, \ldots, z_l) = f(x_1, \ldots, x_n) \cdot \prod_{1 \leq i \leq l} (\overline{z_i} + \overline{p_i})$$

Function g is the product of f and some function which depends on the new inputs. The intuition behind the definition of g is that, in the $z_i = 0$ half of the domain, g is defined as f, while in the $z_i = 1$ half of the domain, g is defined as f but with the i-th privileged cube p_i "filled in" with all 0's (i.e., p_i is "masked out").

Example. For an example, refer again to Figure 4.2 which shows a Boolean function (f,T) with privileged cube x_2 (highlighted in gray). Consider the auxiliary function g, with added variable z_1. In the $z_1 = 0$ half, function g is identical to f. In the $z_1 = 1$ half, g is identical to f except that g is 0 throughout the entire cube $z_1 x_2$, which corresponds to the privileged cube in the original function f. In particular, function g is defined as $g = f \cdot (\overline{z_1} + \overline{p_1})$, where $p_1 = x_2$. $\square$

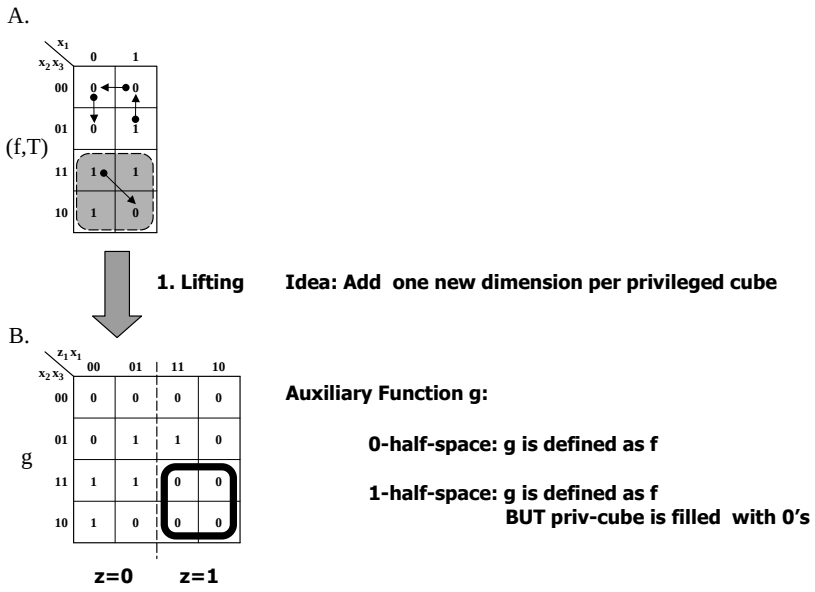


Figure 4.2: Auxiliary Synchronous Function

4.2.3 Prime Implicants of Function g

Once the synchronous function g is obtained, its set of prime implicants $Prime(g)$ is obtained using standard synchronous techniques. A nice feature of the new formulation is that highly-optimized existing synchronous CAD tools can be used for this step [25].

Now, the role of function g is explained by considering its prime implicants $Prime(g)$. The motivation is as follows. Consider the highlighted minterm of function f in Figure 4.3, and its corresponding minterm of function g. Computing prime implicants can be thought of as an expansion process. As indicated in the figure the minterm in the original function f can be expanded in only one dimension, while the corresponding minterm in the auxiliary function g can be expanded in two dimensions. Figures 4.4 and 4.5 visualize the two possible expansions. Note that while either expansion is possible, they are mutually exclusive. Intuitively, the

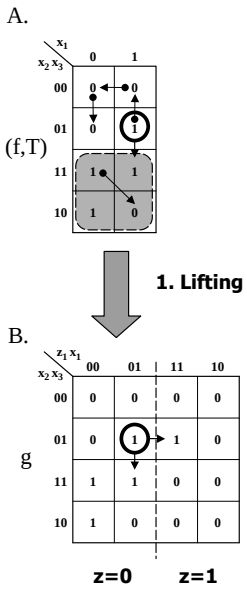


Figure 4.3: Possible Expansions in f and g

first expansion corresponds to an expansion in the original domain. The second expansion does not correspond to an expansion in the original domain but rather it *guarantees* that further expansions will never intersect the highlighted privileged cube in the original domain.

Now, technical detail is provided. First, consider a function (f, T) that has only *one* privileged cube p_1. Let q be any implicant of the function g that is contained in the $z_1 = 0$ plane of g. Since the $z_1 = 0$ plane is defined as f, q also corresponds to an implicant of f. Consider the expansion of q into the $z_1 = 1$ plane of function g. There are 2 possibilities: either (i) q can expand into $z_1 = 1$ plane, or (ii) q cannot expand into the $z_1 = 1$ plane. In case (i), expansion of q into the $z_1 = 1$ plane means that g is identical to f in the expanded region. Therefore, q does not intersect privileged cube p_1 in the original function f (if it did, g would have all 0's in p_1 in the $z_1 = 1$ plane, and expansion would be impossible). In case (ii), expansion into the $z_1 = 1$ plane is impossible. In this case, q must intersect p_1 in

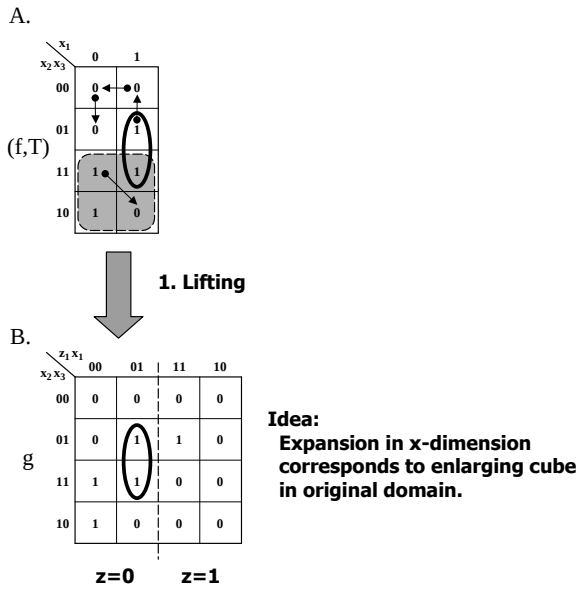


Figure 4.4: Expansion in an Original x Dimension

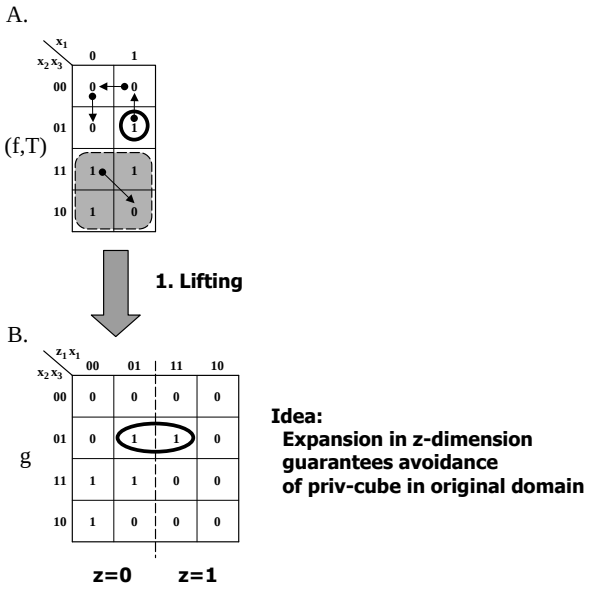


Figure 4.5: Expansion in a New z Dimension

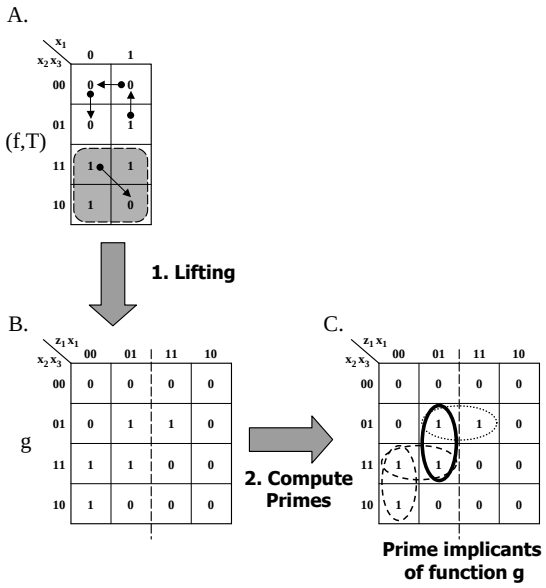


Figure 4.6: Prime Implicants of Auxiliary Function g

function f (g has all 0's in p_1).

In summary, q may or may not be able to expand from $z_1 = 0$ into $z_1 = 1$ planes. Expansion can occur *precisely* if q does not intersect the privileged cube p_1 in the $z_1 = 0$ plane (i.e. in the original function f), since function g is identically defined as f in both planes "outside" the privileged cube. Expansion cannot occur if q intersects the privileged cube p_1, because in the $z_1 = 1$ plane, the privileged cube is filled in entirely with 0's.

Example. Consider the minterm $q_1 = \overline{z_1}x_1\overline{x_2}x_3$ of g in Figure 4.3, which corresponds to the minterm $x_1\overline{x_2}x_3$ of f. q_1 can be expanded into the $z_1 = 1$ plane into the prime implicant of g: $x_1\overline{x_2}x_3$ (highlighted oval in Figure 4.5). Intuitively, the expansion is possible since q_1 does not intersect the privileged cube, i.e. the cube $\overline{z_1}x_2$, which corresponds to the privileged cube x_2 of the original function f. Note that the expansion of q_1 removed the $\overline{z_1}$-literal from q. This fact will be exploited below.

Next, consider the implicant $q_2 = \overline{z_1}x_1x_3$ of g in Figure 4.4. It *cannot* be expanded into the $z_1 = 1$ plane: it intersects the privileged cube, and therefore the corresponding region in the $z_1 = 1$ plane is filled with 0's. Note that prime generation is an expansion process until no further expansion is possible. □

Given the above background, the general case can now be presented, i.e. where (f, T) may have more than one privileged cube. We show that the support variables of each prime of g *precisely* define which privileged cubes are intersected by the corresponding implicant in f.

Let q be any prime implicant of g:

$$q = x_{i_1} \cdots x_{i_{\hat{n}}} z_{j_1} \cdots z_{j_{\hat{l}}}$$

Here, x_{i_k} is a positive or negative x-literal[1]. However, z_{j_k} can *only* be a negative z-literal. The reason is that g is a negative unate function in z-variables (by the definition of g), and therefore prime implicants of g will never include positive z-literals. The notation q^x indicates the *restriction of q to the x-literals*, i.e. $q^x = x_{i_1} \cdots x_{i_{\hat{n}}}$. Note that q^x is an implicant of f by the definition of g.

It is now shown that the presence, or absence, of $\overline{z_i}$ literals in prime implicant q, indicates which privileged cubes are intersected by q^x (i.e. in the original function f). If q includes literal $\overline{z_i}$, then q^x intersects privileged cube p_i in function f. To see this, note that since q is prime, clearly q cannot be expanded into the $z_i = 1$ plane. As a result, as explained above, q^x must intersect privileged cube p_i in the original function f. On the other hand, if q does not include $\overline{z_i}$, then q^x does not intersect p_i. Intuitively, the primes, $Prime(g)$, are maximal in two senses: they

[1]Note that q may not depend on all of the x-variables.

are maximally expanded in f, or maximally non-intersecting of privileged cubes, in some combination, which is explicitly indicated by the set of support of the primes.

In sum, the key observation is that the set of support of a prime implicant q of g *immediately* indicates which privileged cubes are intersected by the corresponding implicant q^x in the original function f. This observation will be critical in obtaining the final set of dhf-prime implicants of f, $dhf\text{-}Prime(f,T)$.

4.2.4 Transforming Prime(g) into dhf-Prime(f,T)

Once $Prime(g)$ is computed, $dhf\text{-}Prime(f,T)$ can be directly computed. The key insight for this computation is that the prime implicants of $Prime(g)$ fall into 3 classes with respect to a specific privileged cube p_i. Each prime q is distinguished based on *if* and *how* it intersects the privileged cube p_i in the original function f, i.e. based on the intersection of q^x with p_i:

- Class 1: Prime implicants q that do not intersect the privileged cube, i.e. q^x does not intersect p_i.
- Class 2: Prime implicants q that intersect the privileged cube legally, i.e. q^x intersects p_i and contains its start point.
- Class 3: Prime implicants q that intersect the privileged cube illegally, i.e. q^x intersects p_i but does not contain its start point.

Prime implicants q that fall into Classes 2 and 3 (i.e. q^x intersects some privileged cube) can be immediately identified by the observation of the previous subsection. Those which fall into Class 3 can then be identified, and removed, using

a simple containment check (i.e. determine if q^x contains the start point of each intersected privileged cube).

The set $dhf\text{-}Prime(f,T)$ can therefore be computed as follows. Start with $Prime(g)$. Filter out all prime implicants that fall in Class 3 with respect to the first privileged cube. Then, filter out all prime implicants that fall in Class 3 with respect to the second privileged cube, and so on. Finally, a set is obtained such that each of its elements is a valid dhf-implicant of (f,T) if restricted to the x-variables. The reason is that, first, all primes of g are implicants of f if restricted to x-variables, and second, the filtering removed any element that intersected any privileged cube illegally. Therefore, the set only includes dhf-implicants. In fact, it also contains *all* dhf-prime-implicants of (f,T). This will be proved in the next subsection.

Example. Figure 4.7C shows function g and its prime implicants, $Prime(g) = \{x_1\overline{x_2}x_3, \overline{z_1}x_1x_3, \overline{z_1}x_2x_3, \overline{z}_1\overline{x}_1x_2\}$. Part D shows the result of filtering out primes that illegally intersect regions corresponding to privileged cubes in f. In this case, $\overline{z_1}x_1x_3$ (oval with thick dark border) falls into Class 3 with respect to p_1: it is deleted since it has a $\overline{z_1}$-literal, i.e. intersects the region corresponding to privileged cube p_1 and, in addition, does not contain the start point $\overline{z}_1\overline{x}_1x_2x_3$. However, $x_1\overline{x_2}x_3$ (oval with dotted border) falls into Class 1: it is not deleted since it does not have a $\overline{z_1}$-literal and therefore does not intersect the region corresponding to the privileged cube p_1. The remaining two primes $\overline{z_1}x_2x_3$ and $\overline{z}_1\overline{x}_1x_2$ (ovals with dashed border) fall into Class 2: they intersect the region corresponding to p_1 and also contain the start point. Part E, in Figure 4.8, shows the result of Step 4 which deletes the z-literals in each cube, $\{x_1\overline{x_2}x_3, x_2x_3, \overline{x_1}x_2\}$, which is $dhf\text{-}Prime(f,T)$. Note that the in-

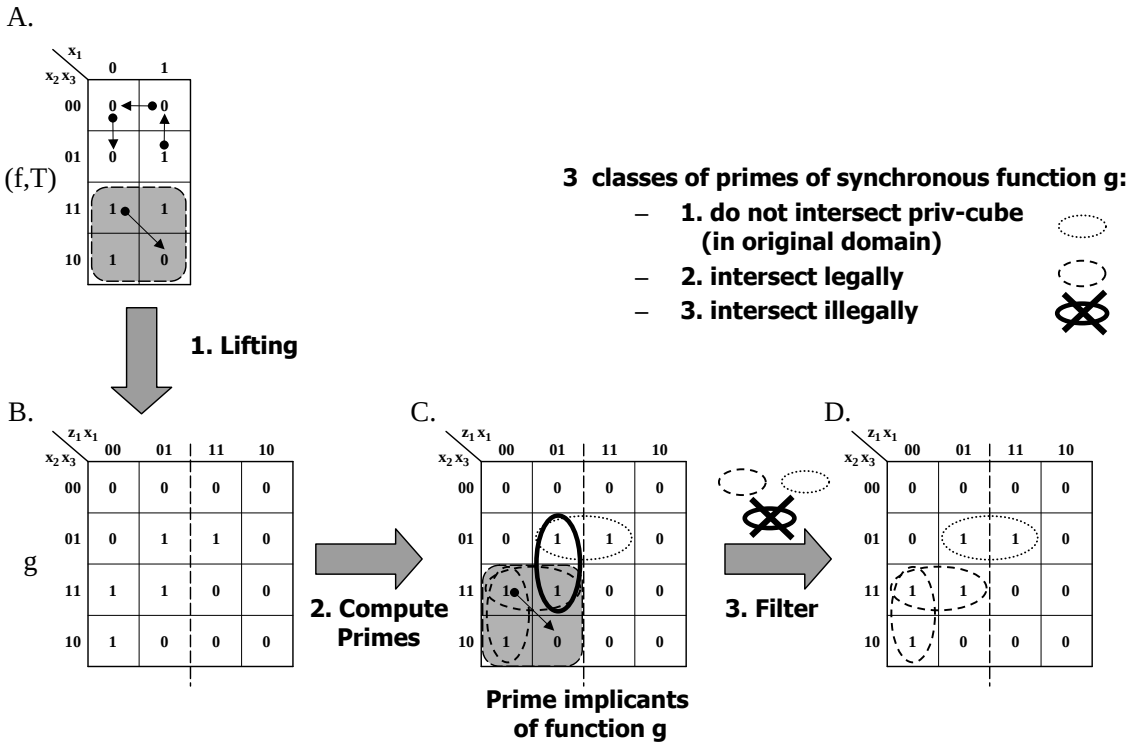


Figure 4.7: Filtering Primes

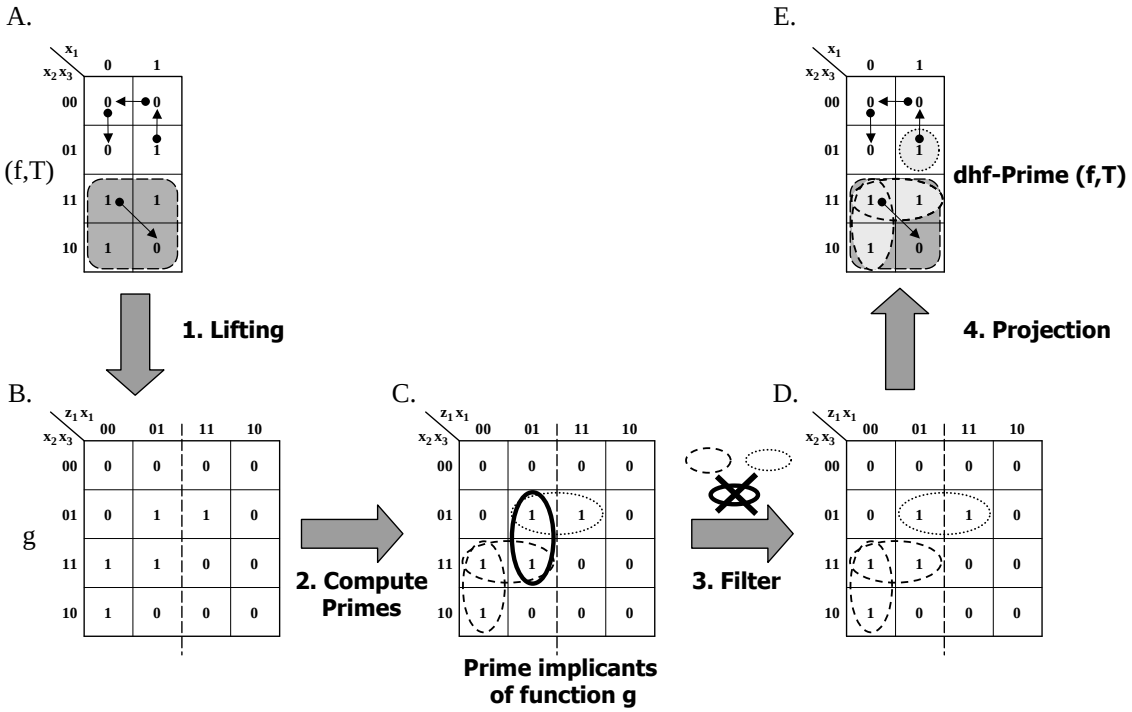


Figure 4.8: Projection

troduction of the z_1-variable ensures that the dhf-implicant of f, $x_1\overline{x_2}x_3$, which is *not* a prime implicant of f, since it is contained by the prime implicant, x_1x_3, is nevertheless correctly generated. □

4.2.5 Formal Characterization of dhf-Prime(f,T) in terms of Function g

Based on the above discussion, this subsection now formalizes the new approach: a characterization of $dhf\text{-}Prime(f,T)$ in terms of an auxiliary function g.

The following notations are used. g_{z_i} and $g_{\overline{z_i}}$ denote the positive and negative cofactors of g with respect to variable z_i, respectively. $RemZ$ denotes an operator on a set of cubes which removes all z-literals of each cube. As an example,

$RemZ\,(\{x_1x_2z_1, x_1x_3\overline{z_2}, x_1x_3z_1z_3\}) = \{x_1x_2, x_1x_3\}$.[2] The SCC-operator on a set of cubes (single-cube-containment) removes those cubes contained in other cubes [66].

Theorem 4.1 *Given* (f,T). *Let* $PRIV(f,T) = \{p_1,\ldots,p_l\}$ *be the set of non-trivial privileged cubes*[3], *and* $START(f,T) = \{s_1,\ldots,s_l\}$ *be the set of corresponding start points. Define*

$$g(x_1,\ldots,x_n,z_1,\ldots,z_l) = f(x_1,\ldots,x_n)\cdot\prod_{1\le i\le l}(\overline{z_i}+\overline{p_i})$$

Then the set dhf-Prime(f,T) can be expressed as follows:

$$SCC\Bigg(\bigcap_{1\le i\le l}\Big[RemZ\,(Prime\,(g_{z_i})) \cup \{q\in RemZ\,(Prime\,(g_{\overline{z_i}}))\,|q\supseteq s_i\}\Big]\Bigg) \tag{4.1}$$

Intuition: $RemZ\,(Prime\,(g_{z_i}))$ includes implicants of f that do not intersect the privileged cube p_i. $\{q\in RemZ\,(Prime\,(g_{\overline{z_i}}))\,|q\supseteq s_i\}$ includes implicants of f that legally intersect p_i, i.e. contain the corresponding start point s_i. The $\bigcap$ ensures that only those implicants remain that are legal with respect to all privileged cubes, *i.e.* that are dhf-implicants. The SCC removes implicants contained in other implicants to yield the final set of dhf-prime-implicants.

Proof: "$\subseteq$" (any product in dhf−Prime(f,T) is also contained in (4.1)): Let $q\in dhf\text{-}Prime(f,T)$, then q does not intersect any privileged cube illegally, i.e. for each privileged cube it holds that q either contains the corresponding start point or does not intersect the privileged cube at all.

[2] $RemZ$ can formally be expressed by existential quantification over z-variables, i.e. $RemZ(P) = \{x\in\{x_1,\overline{x}_1,x_2,\overline{x}_2,\ldots,x_n,\overline{x}_n\}^*\,|\exists z\in\{z_1,\overline{z}_1,z_2,\overline{z}_2,\ldots,z_l,\overline{z}_l\}^*: xz\in P\}$.

[3] In the theorem, $\overline{p_i}$ denotes the complement function of p_i. Example: $p_1 = x_1x_2\overline{x}_4$. Then, $\overline{p}_1 = \overline{x_1x_2\overline{x}_4} = \overline{x}_1+\overline{x}_2+x_4$.

Suppose q intersects legally $p_1, \ldots, p_{\hat{l}}$, and q does not intersect $p_{\hat{l}+1}, \ldots, p_l$ - i.e. q is an implicant of $\overline{p_{\hat{l}+1}}, \ldots, \overline{p_l}$ -, then $q\overline{z_1} \cdots \overline{z_{\hat{l}}}$ is an implicant of g.

$q\overline{z_1} \cdots \overline{z_{\hat{l}}}$ is a prime implicant of g because:

(i) Removing (any) $\overline{z_i}$ results in a cube which is not an implicant of $\overline{z_i} + \overline{p_i}$, and hence not an implicant of g.

(ii) Removing (any) positive or negative x_j literal (of q) results in a cube such that its restriction to the x-literals, q_{new} is not a dhf-prime implicant. Thus q_{new} either intersects the OFF-set of f, or intersects for some i privileged cube p_i, $i \in \{\hat{l} + 1, \ldots, l\}$ and is therefore no longer an implicant of $\overline{z_i} + \overline{p_i}$. In either case q_{new} is not an implicant of g.

Thus, for each i, q is by construction in at least one of $RemZ\left(Prime\left(g_{z_i}\right)\right)$ or $\{q \in RemZ\left(Prime\left(g_{\overline{z_i}}\right)\right) | q \supseteq s_i\}$). Therefore, q is contained in the intersection of those l sets. Also, q cannot be filtered out by the SCC-operator since by construction all cubes contained in (4.1) are dhf-implicants. Thus, q is contained in (4.1).

"$\supseteq$" (any product contained in (4.1) is also contained in dhf-Prime(f)):
Let $q \notin dhf\text{-}Prime(f, T)$. We show that q is not contained in (4.1).

Case (i): q is a dhf-implicant that is strictly contained in some dhf-prime implicant. Then q is filtered out because of the SCC-operator and therefore not contained in (4.1).

Case (ii): q is not a dhf-implicant. By construction all cubes contained in (4.1) are dhf-implicants: the intersection ensures that each cube is valid with respect to all privileged cubes, i.e. the cube either does not intersect or contains the start point. Thus, q cannot be contained in (4.1). □

4.2.6 Multi-Output Case

For simplicity of presentation only, it was assumed that f is a single-output function. However, it is well-known [93] that multi-output logic minimization can be reduced to single-output minimization. Based on this theorem, the above characterization carries over in a straightforward way to multi-output functions. All examples given later in the experimental results section are in fact multi-output functions.

4.3 Exact Hazard-Free Algorithm: IMPYMIN

Based on the ideas of the previous section for generating the set of dhf-prime implicants, this section now presents a new exact implicit minimization algorithm for multi-output 2-level hazard-free logic. Before presenting the new algorithm, the state-of-the-art *synchronous* exact two-level logic minimization algorithm, called SCHERZO [25, 28, 26, 27], is briefly reviewed. SCHERZO forms a basis of the new hazard-free minimization method.

4.3.1 Background on Implicit Synchronous 2-Level Logic Minimization: SCHERZO

Using *implicit* minimization techniques, SCHERZO is 10 to more than 100 times faster than the best previous minimization methods. SCHERZO has solved examples with 10^{20} prime implicants, which is clearly out-of-reach of classic minimization algorithms like the well-known Quine-McCluskey algorithm.

The new concepts of SCHERZO are as follows.

- SCHERZO uses data structures like BDDs [15] and ZBDDs [67] to represent

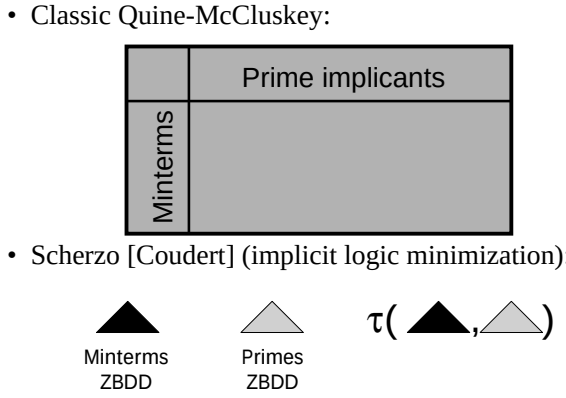


Figure 4.9: Classic vs. Implicit Logic Minimization

Boolean functions and sets of products very efficiently. Thus, the complexity of the minimization problem is shifted, and the cost of the cyclic core computation[4] is now independent of the number of products (e.g. the number of prime implicants) that are manipulated.

- SCHERZO includes new algorithms that operate on these implicit data structures. For example, classic techniques are based on a covering matrix where rows are labeled by the minterms (refer to Figure 4.9) and columns are labeled by the prime implicants. In contrast, SCHERZO operates on two much more compact ZBDDs: one for the minterms and one for the prime implicants. More generally, the motivation is that the logic minimization problem can be considered as a set covering problem over a lattice. More specifically, both the *covering objects*, P, and the *objects-to-be-covered*, Q, are subsets of the lattice $\mathcal{P}$ of all Boolean products (over the set of literals). A new cyclic core computation algorithm uses then two endomorphisms τ_P and τ_Q, which oper-

[4] A set covering problem can be reduced in size by repeated elimination of essential elements and application of dominance relations. The remaining set covering problem (if any) is called the cyclic core.

ate on Q and P respectively, to capture dominance relations and to compute the fixpoint C, which can be shown to be isomorphic to the cyclic core.

Below is a short description of SCHERZO's algorithmic approach[5]. Note that for the understanding of the thesis the actual implementation of these algorithms is not important. Rather it is of interest which data structures they manipulate and that the algorithms have been very effective in practice.

Algorithm: SCHERZO

Input: Boolean function f.

Output: All minimum 2-level implementations of f.

1. Compute the ZBDD $P^{(init)}$ of Prime(f) (the set of all prime implicants of f, or covering objects). Here, f is given as a BDD.

2. Compute the ZBDD $Q^{(init)}$ of the set of ON-set minterms of f, (*i.e.,* the objects to be covered).

3. Solve the implicit set covering problem $\langle Q^{(init)}, P^{(init)}, \subseteq \rangle$ (Note that "$\subseteq$" replaces "$\in$", usually used to describe the relation between the two sorts of objects of a covering problem, since the set covering problem is considered over a lattice, as explained above.)

   (a) Determining the cyclic core:

   Compute the fixpoint C, which is isomorphic to the cyclic core, produced

[5]The ZBDD based recursive algorithms that implement the steps efficiently can be found in [25].

by the following *rewriting rules* on the implicit set covering problem $\langle Q, P, \subseteq \rangle := \langle Q^{(init)}, P^{(init)}, \subseteq \rangle$,

$$\begin{aligned} \langle Q, P, \subseteq \rangle &\rightarrow \langle \max_{\subseteq} \tau_P(Q), \max_{\subseteq} \tau_Q(P), \subseteq \rangle \\ \langle Q, P, \subseteq \rangle &\rightarrow \langle Q - E, P - E, \subseteq \rangle, \\ &\quad \text{with } E = Q \cap P \end{aligned}$$

where τ_P and τ_Q are defined from $\mathcal{P}$ into $\mathcal{P}$ by:

$$\begin{aligned} \tau_Q(r) &= \sup_{\subseteq} \{q \in Q | q \subseteq r\} \\ \tau_P(r) &= \inf_{\subseteq} \{p \in P | r \subseteq p\} \end{aligned}$$

Intuition for τ-operators

To understand the rewriting rules consider first the following examples for *sup* (supremum) and *inf* (infimum): $\sup_{\subseteq}\{x_1x_2, x_2\overline{x_3}\} = x_2$ and $\inf_{\subseteq}\{x_1x_2, x_2\overline{x_3}\} = x_1x_2\overline{x_3}$.

Operator τ_Q maps each product r of the covering objects (initially a prime implicant) onto the supremum of all products (initially on-set minterms) that it covers. Basically, r is mapped onto the smallest cube r' such that r' still covers the same set of products as r. This process often reduces product r.

Operator τ_P maps each product r of the objects-to-be-covered (initially an on-set minterm) onto the infimum of all products (initially prime implicants) by which it is covered. Basically, r is mapped onto the

largest cube r' such that r' is still covered by the same set of products as r. This process often enlarges product r.

max removes cubes contained in other cubes. Each non-maximal covering object can be removed since it is contained by a "better" cube, i.e. one that covers more. Each non-maximal object-to-be-covered can be removed since the containment in another larger object-to-be-covered ensures its covering.

The intuition behind the τ-operators (together with *max*) is that they are very often not injective, that is, they may reduce the size of the covering problem. Basically, the τ operators capture dominance relations. Also, it can be shown that the *essential elements*[6] (above denoted by E) are those elements that are present in the intersection of P and Q at any iteration.

The rewriting rules for $\langle Q, P, \subseteq \rangle$ are iterated until no change: the fixpoint C is computed, which means that the cyclic core is determined and implicitly represented by Q and P.

(b) Solving the cyclic core:

The resulting fixpoint C is solved using a branch-and-bound method, modified to generate all minimum-cost solutions, and step 3(a).

(c) Solutions to covering problem:

Let F be the union of the sets E found during the computation of the fixpoint C in step 3(a). Let $Sol(C)$ be the set of solutions to C. Then the set of *all* solutions of the 2-level logic minimization of f is:

[6]Essential elements are products that are in every minimum solution

$$\bigcup_{S \in Sol(C)} \times_{r \in S \cup F} \{p \in P^{(init)} | r \subseteq p\}$$

Intuition: Each $r \in S \cup F$ represents an equivalence class of primes, which is the set of primes that cover r. Each solution includes exactly one of these primes. The Cartesian product therefore gives rise to the set of all solutions.

4.3.2 A New Implicit Minimization Algorithm: IMPYMIN

As explained in Section 2.3.2, 2-level hazard-free minimization is a covering problem where each required cube must be covered by a dhf-prime implicant.

As with synchronous logic minimization in SCHERZO, hazard-free logic minimization can also be considered over the lattice of the set of products (over the set of literals). The major difference from synchronous two-level logic minimization is in the setting up of the covering problem. In particular, a method is needed that computes the set dhf-Prime(f,T) efficiently, preferably in an implicit manner. To do so, the new characterization of dhf-Prime(f,T) of Section 4.2 is used.

The main flow of the new algorithm IMPYMIN is visualized in Figure 4.10. The four triangles on the bottom show the flow of the new dhf-prime implicant generation approach presented in Section 4.2. The result is the set of covering objects. The single triangle at the top corresponds to the generated set of required cubes, the objects to be covered. The two sets are than handed to the existing implicit set covering solver of SCHERZO, which generates minimum-cost solutions.

Now, the algorithm is described in detail.

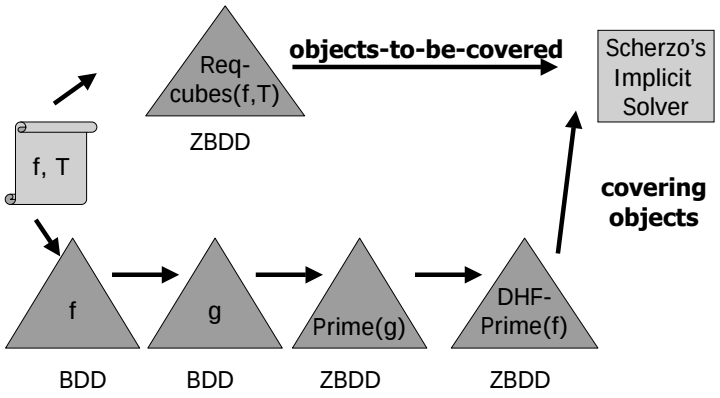


Figure 4.10: IMPYMIN

Algorithm: IMPYMIN

Input: Boolean function f,
set of input transitions T.

Output: All minimum hazard-free 2-level
implementations of (f, T).

1. Compute the ZBDD $P^{(init)}$ of $dhf\text{-}Prime(f, T)$.

2. Compute the ZBDD $Q^{(init)}$ of $REQ(f, T)$ (set of required cubes of (f, T)).

3. Solve the implicit unate set covering problem
   $\langle Q^{(init)}, P^{(init)}, \subseteq \rangle$.

A key challenge is to show that all the above steps can be performed in an *implicit* way.

An important feature of the algorithm is that both Step 1 and Step 3 can re-use synchronous techniques as black boxes. Step 1 is based on Theorem 4.1, which reduces dhf-prime implicant generation to prime generation of a synchronous function. As a result, a fast implicit synchronous prime generation algorithm [25]

can be employed, cf. Section 2.3.3.3. Step 3 is a set covering problem which can be solved by the same efficient solver as used in the synchronous tool SCHERZO.

Now, each of the steps is elaborated in turn.

4.3.2.1 Computation of the ZBDD of dhf-Prime(f,T)

Suppose that f is given as a BDD (if f is given as a set of cubes, then, first, a BDD for f is computed). From the BDD representing f, a BDD for the auxiliary synchronous function g can easily be computed, and then the ZBDD of $Prime(g)$ is computed using an existing recursive algorithm [25], cf. 2.3.3.3. From the ZBDD of $Prime(g)$, the final ZBDD of $dhf\text{-}Prime(f,T)$ can then be computed using Theorem 4.1. It remains to show that the necessary operations, $Prime(g_{z_i}), Prime(g_{\overline{z_i}}), RemZ$, and SCC, for these steps, can be implemented efficiently on ZBDDs:

- *Computing* $Prime(g_{z_i})$: Assuming that positive and negative literal nodes of the same variable are always adjacent in the ZBDD, only a simple traversal of the ZBDD of $Prime(g)$ is necessary. During the traversal each node labeled with a z_i variable is replaced by the result of the following operation. For each such node, the set union of the two successors corresponding to those products that include positive literal z_i and to those products that do not depend on z_i is computed. The resulting ZBDD, i.e. after the complete traversal, may actually include non-primes. That is, some cubes may be contained in other cubes. However, these cubes are filtered out by SCC (see below).

- *Computing the ZBDD of* $Prime(g_{\overline{z_i}})$: Analogously.

- *Computing the ZBDD of RemZ*: *RemZ* deletes all z-literals in the ZBDD. The ZBDD is traversed, and at each z_i- or $\overline{z_i}$-literal, the corresponding node is replaced with the ZBDD corresponding to the union of the two successors.

- *SCC (Single-Cube Containment)*: The last task, the application of the *SCC*-operator, which removes cubes contained in other cubes, is actually not performed in this step, since it is automatically handled in Step 3 of the algorithm.

To summarize, based on Theorem 4.1 it is possible to compute the covering objects, $dhf\text{-}Prime(f,T)$, in an implicit manner.

4.3.2.2 Computation of the ZBDD of REQ(f,T)

From the set of input transitions, T, the set of required cubes can easily be computed (see [82]). This set can then be stored as a ZBDD.

4.3.2.3 Solving the Implicit Covering Problem

The implicit set covering problem $\langle Q^{(init)}, P^{(init)}, \subseteq \rangle$ can be solved analogously to Step 3 of SCHERZO, i.e. passed directly to the unate set covering solver of SCHERZO.

4.3.3 A Note on the Efficiency of IMPYMIN

IMPYMIN appends z-variables in dhf-prime generation during the construction of the synchronous function g. It is worth pointing out that the algorithm does not become unattractive even in cases where many z-variables are necessary. Such cases typically arise when there are many dynamic transitions, and hence many

privileged cubes. In practice, the addition of many z-variables does not necessarily imply that the BDD for g will be much larger than the BDD for f (see discussion in Subsection 4.4.5).

Experimental results also indicate that IMPYMIN has significantly better runtime than existing asynchronous methods on large examples. It also performs hazard-free logic minimization nearly as efficiently as synchronous logic minimization for many examples. One reason is that the new characterization of the set of dhf-prime implicants, presented in Section 4.2, makes it possible to use state-of-the-art synchronous tools for implicit prime generation and implicit set covering solving (see Subsection 4.4.5 for a detailed discussion).

4.4 Experimental Results and Comparison with Related Work

Prototype versions of both of the two new minimizers — ESPRESSO-HF[7] and IMPYMIN — have been run on a number of well-known benchmark circuits [37, 101]. An ULTRA-SPARC 140 workstation with 89 MB real and 230 MB virtual memory was used.

Comparisons are made below, between various exact minimizers (IMPYMIN vs. HFMIN, IMPYMIN vs. Rutten, IMPYMIN vs. Myers/Jacobson) and between the two new exact and heuristic minimizers (IMPYMIN vs. ESPRESSO-HF), as well as

[7]The implementation of ESPRESSO-HF is not a simple modification of the ESPRESSO-II code: the new tool does not re-use any ESPRESSO-II code. The reason is that while ESPRESSO-HF is based on the same set of main operators - EXPAND, REDUCE, IRREDUNDANT - the algorithms that implement these operators, as explained in detail in Chapter 3, are actually very different from ESPRESSO-II.

comparisons between hazard-free minimizers and standard synchronous minimizers.

A quick summary is that the two new minimizers outperform existing techniques by several orders of magnitude on larger examples. In particular, IMPYMIN can find a minimum-size cover for all benchmark examples in less than 813 seconds, and ESPRESSO-HF can find very good covers — at most 3% larger than a minimum-size cover — in less than 105 seconds. Surprisingly, the two new tools are often comparably fast as the corresponding synchronous tools.

4.4.1 Comparison of Exact Hazard-Free Logic Minimizers: IMPYMIN vs. HFMIN

The table in Figure 4.11 compares the new exact minimizer IMPYMIN with the currently fastest available exact minimizer, HFMIN, by Fuhrer et al. [37].

For smaller problems, HFMIN is faster. It should be noted, though, that the implementation of IMPYMIN is not yet optimized[8]. However, the bottleneck of HFMIN becomes clearly visible already for medium-sized examples. For *sd-control* and *stetson-p2*, IMPYMIN is more than three times faster; for the benchmark *pscsi-pscsi* more than fifteen times.

For very large examples, IMPYMIN outperforms HFMIN by a large factor. While HFMIN cannot solve *stetson-p1* within 20 hours, IMPYMIN can solve it in just 813 seconds. The superiority of implicit techniques becomes very apparent for

[8]The BDD package that is employed is still very inefficient. In particular, it includes a static (i.e. not a dynamic) hashtable. The hashtable for small examples is unnecessarily large. In fact, the run-time is completely dominated by initializing the hashtables. If an appropriate-sized hashtable for smaller examples is used, then experiments indicate that IMPYMIN can solve the small examples as fast as HFMIN. The employed BDD package was originally used for the work presented in [33].

			HFMIN [37]	IMPYMIN
name	*i/o*	*#c*	*time(s)*	*time(s)*
cache-ctrl	20/23	97	impossible	301
dram-ctrl	9/8	22	1	13
pe-send-ifc	12/10	27	9	16
pscsi-ircv	8/7	12	1	10
pscsi-isend	11/10	23	3	15
pscsi-pscsi	16/11	77	1656	105
pscsi-tsend	11/10	22	3	13
pscsi-tsend-bm	11/11	23	3	13
sd-control	18/22	34	172	52
sscsi-isend-bm	10/9	22	1	11
sscsi-trcv-bm	10/9	24	1	13
sscsi-tsend-bm	11/10	20	2	13
stetson-p1	32/33	60	> 72000	813
stetson-p2	18/22	37	151	49
stetson-p3	6/4	7	1	8

Figure 4.11: Comparison of Exact Hazard-Free Minimizers (#c - number of cubes in minimum-cost solution, time - run-time in seconds)

the benchmark *cache-ctrl*. While HFMIN gives up (after many minutes of run-time) because the 230MB of virtual memory are exceeded, the new proposed method can minimize the benchmark in just 301 seconds.

4.4.2 Comparison of New Methods: IMPYMIN vs. ESPRESSO-HF

Figure 4.12 compares the two new minimizers ESPRESSO-HF and IMPYMIN. Besides run-time and size of solution, the table also reports the number of essentials (for ESPRESSO-HF) and the number of variables that need to be added (for IMPYMIN).

The two minimizers are somewhat orthogonal.

On the one hand, IMPYMIN computes a cover of minimum size, whereas ESPRESSO-HF is not guaranteed to find a minimum cover, but typically does find a cover of very good quality. In particular, ESPRESSO-HF finds always a cover that is at most 3% larger than the minimum cover size. It is worth pointing out that many examples were very positively influenced by the new notion of essentials. Quite a few examples can be minimized by *just* the essentials step, resulting in a guaranteed minimum solution; e.g. *dram-ctrl* and *pe-send-ifc*.

On the other hand, ESPRESSO-HF is typically faster than IMPYMIN. However, since neither tool has been highly optimized for speed, it is very important to analyze the intrinsic advantages and disadvantages of the two methods. Intuitively, both methods overcome the three bottlenecks of HFMIN — prime implicant generation, transformation of prime implicants to dhf-prime implicants, and solution of the covering problem — each of which being solved by an algorithm with exponential worst-case behavior. However, the way in which ESPRESSO-HF and IMPYMIN overcome these bottlenecks is very different. Whereas IMPYMIN uses implicit data structures (but still follows some of the same basic steps as HFMIN), ESPRESSO-HF follows a very different (heuristic) approach. Thus, the two methods are orthogonal in their approach to overcome these bottlenecks. Moreover, while ESPRESSO-HF is faster than IMPYMIN on all of the examples that were run, this does not mean that this is necessarily true for other examples.

In this context, it is important to note that very often the role data structures like BDDs play in obtaining efficient implementations of CAD algorithms is misunderstood. Using BDDs, many CAD problems can now be solved much faster than before the inception of BDDs. However, the naive approach of taking an ex-

		Espresso-HF			Impymin				Espresso-II [88]			Scherzo [25]	
name	*i/o*	*#c*	*time*	*#e*	*#c*	*time*	*#v*	*BDDf/g*	*#c*	*time*	*#e*	*#c*	*time*
cache-ctrl	20/23	99	105	50	97	301	39	795/1813	89	217	7	80	756
dram-ctrl	9/8	22	1	22	22	13	6	91/140	19	1	6	18	1
pe-send-ifc	12/10	27	1	27	27	16	5	158/299	20	1	1	20	1
pscsi-ircv	8/7	12	1	12	12	10	3	45/110	10	1	4	10	1
pscsi-isend	11/10	23	1	23	23	15	6	115/264	16	1	1	16	1
pscsi-pscsi	16/11	78	11	55	77	105	23	319/852	66	5	10	63	14
pscsi-tsend	11/10	22	1	22	22	13	4	113/223	16	1	3	16	1
pscsi-tsend-bm	11/11	23	1	23	23	13	4	112/231	16	1	3	16	1
sd-control	18/22	35	3	23	34	52	0	448/448	25	2	4	24	14
sscsi-isend-bm	10/9	22	1	22	22	11	3	98/153	15	1	5	15	1
sscsi-trcv-bm	10/9	24	1	21	24	13	5	96/189	15	1	3	15	1
sscsi-tsend-bm	11/10	20	1	20	20	13	4	123/214	15	1	2	15	1
stetson-p1	32/33	60	21	34	60	813	9	1463/1933	45	33	1	?	> 72000
stetson-p2	18/22	37	2	26	37	49	0	457/457	25	1	6	25	17
stetson-p3	6/4	7	1	7	7	8	1	30/41	7	1	6	7	1

Figure 4.12: Comparison of the Heuristic Hazard-Free Minimizer Espresso-HF, the Exact Hazard-Free Minimizer Impymin, the Heuristic Minimizer Espresso-II, and the Exact Minimizer Scherzo. (#c - number of cubes in solution, time - run-time in seconds, #e - number of essentials, #v - number of added variables, BDD f/g - BDD sizes without/with added variables)

isting CAD algorithm and augmenting it with BDDs does not necessarily lead to a good tool (see discussion in [25]). In particular, it is not easily possible to simply augment Espresso-HF or Hfmin with BDDs to obtain a high-quality tool. Instead, a new theoretical formulation was needed on the characterization of dhf-prime implicants (cf. Section 4.2.5), on which the new exact implicit minimizer Impymin could be based.

4.4.3 Comparison with Rutten's Work

An interesting alternative recent approach to the new characterization of dhf-prime implicants (cf. Section 4.2.5) was recently proposed by Rutten et al. [92, 91], as part of an exact hazard-free minimization algorithm. His new algorithm to computing dhf-prime implicants is very different from ours. His approach follows

a divide-and-conquer paradigm. In particular, the dhf-prime generation problem is split into three sub-problems with respect to a splitting variable. The first (second, third) sub-problem generates those dhf-prime implicants that have a positive literal (negative literal, don't care-literal) for the splitting variable. The underlying idea why this approach may be efficient is that it allows to determine illegal intersections of privileged cubes already during the splitting phase (see [92] for details), which can significantly reduce the recursion tree and lead fast to terminal cases. In the merging phase of the divide-and-conquer approach, the solutions to the sub-problems are then combined.

However, it is worth pointing out that a major difference of the proposed work in this thesis to Rutten's work is that his approach is *not* based on implicit representations, while ours is. Furthermore, while Rutten's work is promising, it has not been fully evaluated so far. In particular, he only presented run-times for functions that are *significantly smaller* than those that can be handled by our two new methods. To be precise, on the examples he reports, his own re-implementation of the existing HFMIN tool never takes more than a few seconds. Thus, Rutten evaluates his approach (and admittedly shows improvement) only on examples that can already easily be solved by existing algorithms. In contrast, as shown in the previous subsection, our new methods are more powerful, since they can solve examples efficiently that cannot be solved by HFMIN within several hours of run-time.

4.4.4 Comparison with the Approach of Myers and Jacobson

Myers and Jacobson [70] have recently proposed a new method for *single-output* two-level hazard-free logic minimization. Their method optimizes the number of literals rather than the number of cubes in a solution.

The proposed method is based on techniques originally used in hazard-free algorithms of speed-independent circuits [53]. The algorithm derives required cubes directly from a burst-mode specification. Then, each required cube is considered in turn, and a set of dhf-prime implicants covering the required cube is generated. Next, a covering problem is formulated where the objects-to-be-covered are the required cubes and the covering objects are all generated dhf-prime implicants.

Myers and Jacobson have applied their method only to *single-output* logic minimization so far. For the considered examples, they have reported very good run-times, comparable to ESPRESSO-HF and somewhat faster than IMPYMIN. (In addition, their solution is literal-exact, an option which is currently not available for IMPYMIN.) However, no results have been reported for the set of *multi-output* benchmarks, which we have used to evaluate IMPYMIN and ESPRESSO-HF. Those benchmarks pose much harder problems.

4.4.5 Comparison of Synchronous and Asynchronous Minimization

Some interesting points regarding the new tools for 2-level hazard-free minimization can best be highlighted by comparing them to the two corresponding state-of-the-

art tools for 2-level non-hazard-free minimization, ESPRESSO-II and SCHERZO (see Figure 4.12).

The table in Figure 4.12 compares both *run-time* and *cardinality* of solution for all four minimizers. In addition, the table indicates the number of identified *essentials* for the two heuristic minimizers, ESPRESSO-II and ESPRESSO-HF. Finally, for IMPYMIN, it reports the *number of added variables* and their impact on *BDD size*.

The run-time comparison indicates that, although our tools are not implemented as efficiently as their synchronous counterparts, they are comparably fast. Interestingly, the two new tools are actually faster than the synchronous tools for the two largest examples, *cache-ctrl* and *stetson-p1*. For our set of benchmarks, this seems to indicate that the *more constrained* asynchronous problem, which is to minimize a function f without hazards for a set of transitions T, may be easier than the corresponding synchronous problem, which is to minimize the same function f without any specified input transitions and without hazard-free constraints.

The comparison in terms of cardinality of solution indicates an increase in the asynchronous case compared with the synchronous case. In an earlier comparison [82], it was observed that the logic overhead for the asynchronous case was never greater than 6%. In contrast, in our table, there is a large variation in overhead, ranging from 0% (*stetson-p3*) to 60% (*ssci-trcv-bm*). The increase in overhead is due to the fact that we report on significantly more complex problems: while [82] only performed single-output minimization, we do multi-output minimization (on many of the same circuit examples), including for functions ranging up to 32 inputs and 33 outputs.

However, it is important to note that this table should not be used to draw general conclusions regarding how much logic overhead asynchronous designs incur due to the necessity to avoid hazards. Our benchmark functions have been generated by asynchronous synthesis methods, i.e. these functions do not really make much sense in a synchronous system. On the one hand, functions derived from asynchronous FSMs must have function-hazard-free input changes and critical race-free state changes, unlike those derived from synchronous FSMs. On the other hand, asynchronous FSMs are typically specified in a more controlled environment, with more don't cares. A truly fair comparison on this interesting point is much beyond the scope of thesis.

Figure 4.12 also compares the number of identified essentials, using both hazard-free and non-hazard-free algorithms. ESPRESSO-HF's new formulation of essential equivalence classes typically allows many more essentials to be identified than in ESPRESSO-II. For example, in *cache-ctrl*, ESPRESSO-HF identifies 50 essentials (out of an exact minimum cover of 97 cubes), while ESPRESSO-II identifies only 7 essentials (out of an exact minimum cover of 80 cubes). Thus, ESPRESSO-HF makes positive use of hazard-freedom constraints to obtain a very strong formulation of essentials, which has positive impact on both run-time and quality of solution.

Finally, IMPYMIN copes with hazard-freedom constraints in a data-structure-efficient way. To cope with hazard-freedom constraints, IMPYMIN appends z-variables in dhf-prime generation during the construction of the synchronous function g. However, as can be seen in the table, adding (sometimes many) variables in IMPYMIN does not lead to an explosion in terms of BDD size. To incorporate

hazard-freedom constraints, IMPYMIN (unlike SCHERZO) transforms the BDD of f into the BDD of auxiliary function g. The table, which compares the corresponding BDD sizes for the same BDD package and variable ordering, indicates that adding variables for this transformation increases the BDD size even for large examples only by a small factor, which is typically about 2. Thus, the BDD size of auxiliary function g is not much larger than the BDD size of f.

4.5 Conclusions

Two new minimization methods have been presented for multi-output 2-level hazard-free logic minimization: ESPRESSO-HF, a heuristic method based on ESPRESSO-II (in the previous chapter), and IMPYMIN, an exact method based on implicit data structures.

IMPYMIN performs exact hazard-free logic minimization nearly as efficiently as synchronous logic minimization by incorporating state-of-the-art techniques for implicit prime generation and implicit set covering solving. IMPYMIN is based on the new idea of incorporating hazard-freedom constraints within a synchronous function by adding extra inputs. We expect that the proposed technique may very well be applicable to other hazard-free optimization problems, too.

Both tools can solve all examples that we have available. These include several large examples that could not be minimized by previous methods[9]. In particular both tools can solve examples that cannot be solved by the currently fastest minimizer HFMIN. On the more difficult examples that can be solved by

[9] In publications on the 3D method (see e.g. [111, 108]), note that several of these examples appear but only *single-output* minimization is performed.

HFMIN, ESPRESSO-HF and IMPYMIN are typically orders of magnitude faster.

Chapter 5

Distributed Control Synthesis

A key bottleneck to the advancement of asynchronous design is the lack of high-quality CAD tools for the synthesis of large-scale systems which also allow design-space exploration. This chapter proposes a comprehensive new synthesis method to address this issue, based on transformations.

Nearly all existing CAD frameworks for large asynchronous systems generate only a single implementation for a given specification, and typically one of lower quality than a highly-optimized manual design. In contrast, the proposed approach has two significant advantages, which make it unique when compared to existing tools: (a) it provides a wide-ranging set of transformations to allow the user to produce a variety of implementations, thus facilitating design-space exploration; (b) the set of transformations includes aggressive and powerful restructuring optimizations which allow the designer to approach (or obtain) the same quality as that of optimized manual design.

In particular, the proposed method starts with a scheduled and resource-bounded Control-Data Flow Graph (CDFG) [66]. Global transformations are first

applied to the entire CDFG, unoptimized controllers are then extracted, and, finally, local transforms are applied to the individual controllers. The result is an optimized set of interacting distributed controllers. The new transforms include aggressive timing- and area-oriented optimizations, several of which have not been previously supported by existing asynchronous CAD tools.

As a detailed case study, the transformations are applied to the well-known *differential equation solver* high-level synthesis benchmark [66]. The resulting implementation has quality comparable to a highly-optimized manual design by Yun et al. [109]. Such an implementation cannot be obtained using existing asynchronous CAD tools.

5.1 Introduction

A key current limitation of asynchronous design is the lack of high-quality CAD tools for systematic design-space exploration and optimization of large-scale asynchronous systems. Traditionally, a number of asynchronous CAD tools are limited to the design of individual controllers [38, 39, 24, 110, 69, 3, 56, 52], and thus are only useful for one step in the overall synthesis flow.

For large-scale asynchronous systems, two design approaches are now widely-used: (i) *manual design,* and (ii) use of *syntax-directed CAD tools.* Manual design allows a number of aggressive optimizations, but is cumbersome, slow and error-prone, and it does not provide systematic and automated exploration of the design space. For example, the Intel asynchronous instruction-length decoder chip [87] took over two years to complete, using a combination of manual techniques and academic synthesis tools for designing individual controllers.

Alternatively, several automated approaches have been proposed for large-scale systems which are syntax-directed [13, 8, 9, 85]. These methods start from a high-level abstraction, such as a concurrent program, and obtain a circuit by translating each individual program construct into a corresponding sub-circuit. For example, the Philips' Tangram tool [8, 9, 85], developed by Kees van Berkel, Ad Peeters et al., provides only one implementation per specification. There are no options, other than some simple peephole techniques, for design-space exploration. If the user is dissatisfied with the circuit, the original program must be manually restructured and re-compiled in the hopes of some improvement. Other CAD approaches allow only restricted and non-systematic techniques for design-space exploration [62, 13]. Thus, a huge potential for optimization has been overlooked for a long time.

Recently, there is increasing interest in alternative approaches to the synthesis of large-scale asynchronous systems [2, 21, 54, 10, 14, 49, 59, 51, 73, 1]. Cortadella and Badia propose a synthesis style for the control unit where each datapath block is controlled by a dedicated sub-controller [21], while Kim et al.'s approach subdivides these sub-controllers even further, assigning a sub-sub-controller to each of the processes bound to a functional unit [49]. These approaches are strictly deterministic and "template-based". Only one approach [1] considers design space exploration, but at a higher level: for resource binding and allocation. Interestingly, some efficient manual designs have been presented to which none of these methods has access [109]. Thus, there is still a serious lack of approaches providing systematic design-space exploration.

The contribution of this chapter is a new approach for the automated syn-

thesis and optimization of large-scale asynchronous systems. In particular, the new approach is the first to introduce, formalize and automate a wide-ranging and powerful set of transformations, which can be used for the synthesis of asynchronous distributed control. Unlike previous approaches, these new transforms can be applied in a systematic way to explore the design space and find optimal distributed controller implementations.[1]

The new method starts with a given scheduled and resource-bounded Control-Data Flow Graph (CDFG) [66]. Global transforms are first applied to the entire CDFG, unoptimized controllers are then extracted, and, finally, local transforms are then applied to the individual controllers. The result is a highly-optimized set of interacting distributed controllers. The transforms include aggressive timing- and area-oriented optimizations such as: global communication channel multiplexing and symmetrization; loop parallelism; introduction of global "relative timing"-based simplification; multiplexor pre-selection; sharing of local signals; and the removal of unnecessary handshaking wires. Several of these optimizations have not been previously formalized or provided by any other existing asynchronous CAD tool, or else in only a limited way. For example, Kim et al.'s recent approach [49] is also based on CDFGs, however their method does not do design space exploration, and is limited to handling less concurrent specifications than our approach.

As a detailed case study, the transformations are applied to the well-known *differential equation solver* high-level synthesis benchmark [109, 66]. A highly-optimized asynchronous implementation by Yun et al. [109] was manually designed,

[1]These transforms, while at a much higher level of synthesis, are loosely analogous to the powerful transforms of SIS (collapse, extract, etc.) used for design-space exploration in multi-level logic synthesis.

using a number of aggressive timing- and area-based optimizations. Such an implementation cannot be obtained using existing CAD tools. We demonstrate that a very similar optimized design can be simply and automatically derived through systematic application of our new transformations.

The remainder of this chapter is organized as follows. Section 5.2 gives a simple overview of the approach, including the initial CDFG specification, final target architecture, and transformation method. The next three sections describe the three synthesis steps in detail: Section 5.3 presents the new global transformations, which operate on the CDFG itself; Section 5.4 explains the extraction of the individual unoptimized controllers; and Section 5.5 presents several local transformations, which optimize the interaction of a controller and the datapath. Section 5.6 compares the new approach with a number of other approaches in detail. Section 5.7 presents results on a differential equation solver benchmark, and compares with a manual design of Yun. Finally, Section 5.8 presents conclusions and future work.

5.2 Overview of Approach

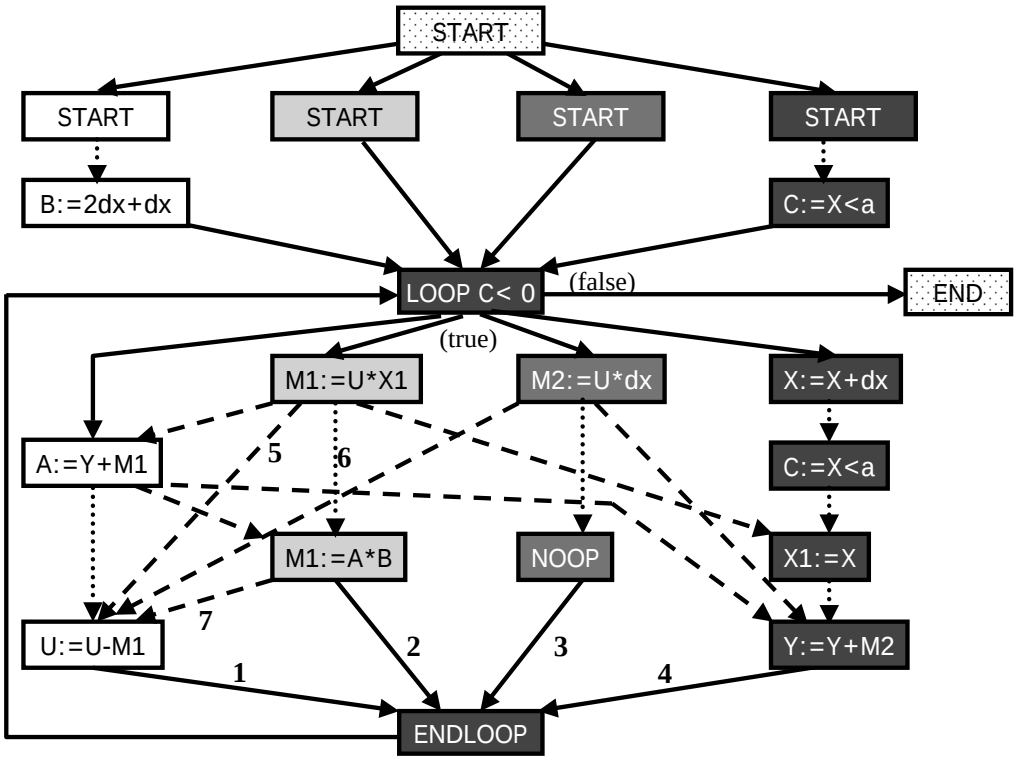
This section presents a basic overview of the synthesis and optimization method. The initial CDFG specification and the target architecture are first introduced. Then, a brief summary of the synthesis flow is presented.

5.2.1 CDFG Specification

The new synthesis method receives as an input a scheduled and resource-bounded CDFG [66] as shown in Figure 5.1. The shown CDFG represents the well-known *differential equation solver* (DIFFEQ) high-level synthesis benchmark [109, 66]. This benchmark will be used as a running example through the entire chapter to illustrate our new approach.

In the figure, all operation nodes bound to the same functional unit are placed within the same column. For example, the three RTL statements $B := 2dx + dx$, $A := Y + M1$, and $U := U - M1$ are all bound to the ALU1 unit. In total, there are four functional units: two ALUs (ALU1 and ALU2, first and last column) and two multipliers (MUL1 and MUL2). Note that the LOOP and ENDLOOP nodes are both bound to ALU2. Each functional unit has a START node bound to it. There is also one global START node and one global END node (dotted background) which are both not bound to any functional unit. In addition to the types of nodes in the example, the approach also allows IF and ENDIF nodes.

The CDFG includes arcs that are typically present in synchronous CDFGs, as well as new types of arcs that are specific to the asynchronous case. The former includes the data-dependency arcs (dashed arcs), as well as the control arcs going from and to the LOOP, ENDLOOP, IF, ENDIF, START, and END nodes. In the synchronous case, only these arcs would be used, as well as assignments to time slices that indicate the scheduling of operations. In the asynchronous case, however, scheduling information must be made explicit: precedence arcs are added between operations bound to the same unit to enforce the schedule (dotted arcs). Finally, the correct order of register writes and reads must be enforced (dashed arcs, like



Legend:

Nodes:

Each node is assigned to a functional unit. Nodes with same color are assigned to same functional unit.

- assigned to functional unit ALU1
- assigned to functional unit MUL1
- assigned to functional unit MUL2
- assigned to functional unit ALU2
- not bound to any functional unit

Arcs:

- control arcs
- data dependency and register assignment arcs
- functional unit scheduling arcs

Figure 5.1: CDFG for DIFFEQ

data-dependency arcs). Details will be given below. An operation node in a CDFG may "fire" if all its predecessors have "fired".

For the proposed approach, the CDFG is assumed to be *block-structured*: the set of nodes between IF and ENDIF nodes, and LOOP and ENDLOOP nodes are considered a *block*. Data dependency arcs, control flow arcs, and register allocation arcs may never cross block boundaries; these arcs can only enter or exit at the block root node (IF or LOOP). (This restriction simplifies the handling of data dependency constraints and register resources, which are allocated on a per-block basis.)

In the asynchronous case, where operations may take non-fixed (i.e., variable) amounts of time, a legal schedule of operations is obtained by a direct implementation of the constraint arcs. That is, each constraint arc is implemented (in the basic case) by a single wire or *global channel*, which is used to signal when the receiving CDFG node is allowed to execute. (Constraint arcs that connect two nodes bound to the same functional unit need, of course, not be implemented by wires. Details will be explained later.)

An operation node $R_1 := R_2 \, op \, R_3$ has the following constraint arcs that indicate when the operation node may fire and which consequences the firing on other operation nodes has:

1. **Control flow (solid arcs):** control arcs from and to START, END, IF, ENDIF, LOOP and ENDLOOP nodes
2. **Scheduling within a FU (dotted arcs):** scheduling arcs to order the operations assigned to a functional unit
3. **Data dependency (dashed arcs):**

- incoming arcs from operations that provide its operands R_2 and R_3
- outgoing arcs to operations that use the result R_1

4. **Register allocation (dashed arcs):**

   - incoming arcs from all operations that use the old register value of R_1, to avoid early overwriting of R_1
   - outgoing arcs to the next writes to R_2 and R_3 to avoid early writes of R_2 and R_3

Example: A constraint arc that has source node a and destination node b is denoted: (a, b). In Figure 5.1, the arc $(LOOP, A := Y + M1)$ is a control arc, and $(A := Y + M1, U := U - M1)$ is a scheduling arc for ALU1. The arcs $(M1 := U * X1, A := Y + M1)$ and $(A := Y + M1, M1 := A * B)$ illustrate the data dependencies incident to the node $A := Y + M1$. It is possible that arcs of different types connect the same nodes. For example, the arc $(M1 := U * X1, U := U - M1)$ is a register allocation constraint arc with respect to U, and it is also a data dependency arc with respect to $M1$. However, to simplify the figures, only single arcs are shown.

5.2.2 Target Architecture

The target architecture for the proposed approach is shown in Figures 5.2 and 5.3. Figure 5.2 first shows a system view. The part that is surrounded with a dashed box in Figure 5.2 is then shown in more detail in Figure 5.3.

The datapath of the target architecture consists of functional units (each with associated dedicated input muxes), as well as registers (each with an associated input mux), as indicated in Figure 5.2.

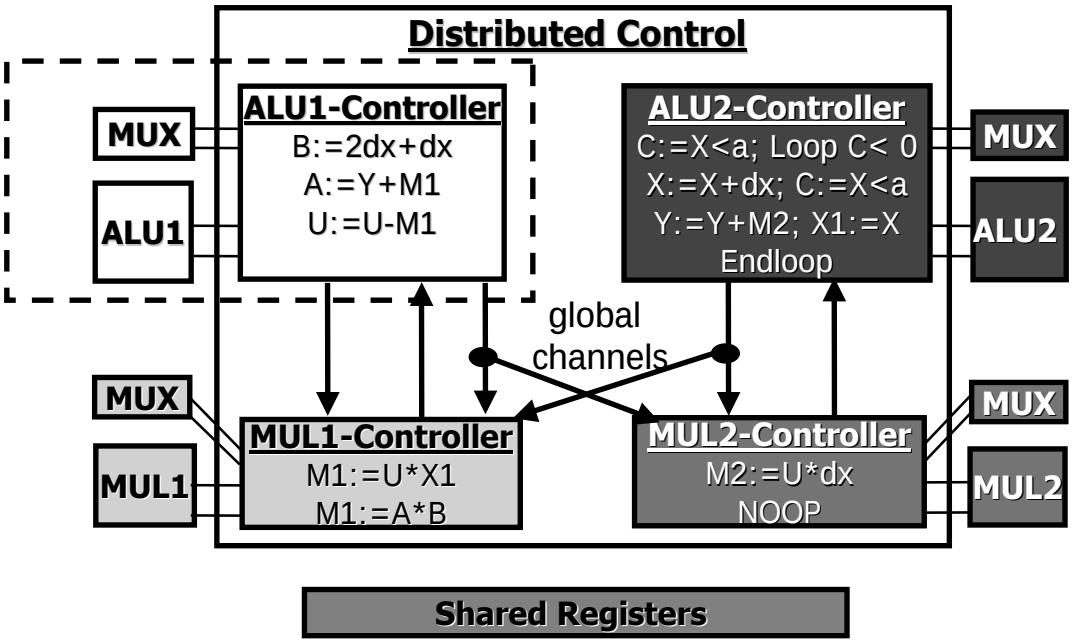
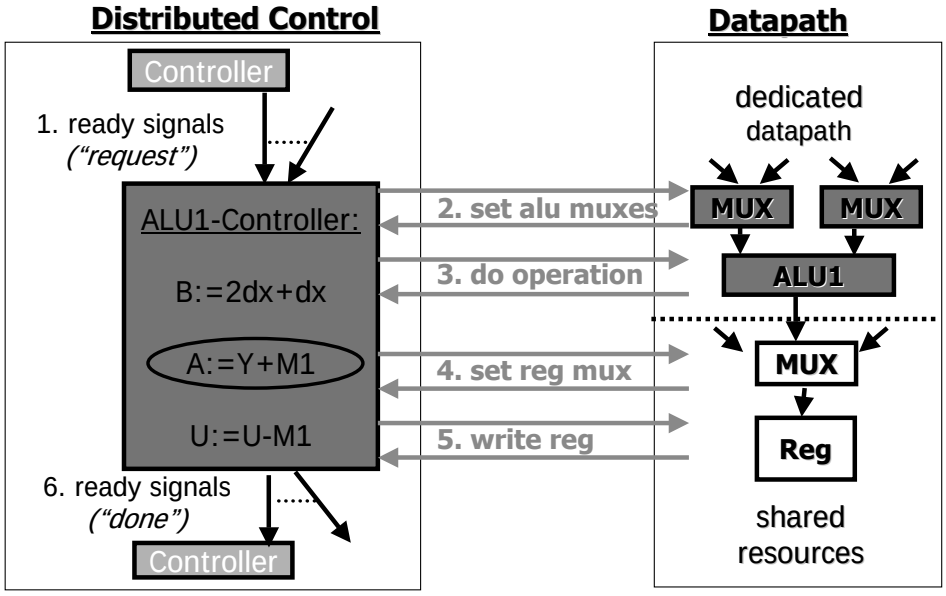


Figure 5.2: Target Architecture: DIFFEQ



Global communication (steps 1. and 6.): a single transition (ready+ or ready-)
Local communication (steps 2., 3., 4., and 5.): 4-phase (req+, ack+, req-, ack-)

Figure 5.3: Detailed Architecture: One Controller (ALU1) and Its Associated Datapath for DIFFEQ

The distributed control consists of one controller per functional unit. Each controller interacts with other controllers, with its dedicated functional unit and input muxes, and with registers and their input muxes. Note that the registers and their input muxes may be shared by other controllers.

The basic operating protocol, indicated in Figure 5.3, is as follows. A functional unit controller waits for a set of "ready" signals from other controllers. These signals indicate that the controller may execute the next RTL statement bound to the corresponding functional unit. Once enabled, the controller then interacts with the datapath according to the figure, i.e. by selecting the appropriate source input muxes, then activating its functional unit, then selecting the appropriate destination register mux, and finally latching the result. As a last step, the functional unit, in turn, signals to other controllers with "ready" signals that it has completed execution of the RTL statement.

Controller-controller communication (using "ready" signals) is implemented using a form of "transition signaling". Unlike in standard 2-phase transition-signaling protocol [13], where a transition pair on two wires ($req+/ack+$ or $req-/ack-$) completes a communication between sender and receiver, the proposed scheme is even simpler: no acknowledgment wire is used. Thus, controllers communicate with each other *by a single transition ($req+$ or $req-$) on one wire.* (This scheme is based on the observation that such a ready signal is typically the last event in executing an RTL statement, and no acknowledgment is required.) "Ready" signals serve thus two purposes: incoming signals to a functional unit are "request" signals, and outgoing signals are "done" signals. In contrast, the controller-datapath communication uses a standard 4-phase protocol. In a 4-phase protocol, a standard

return-to-zero handshake protocol is used: $req+, ack+, req-, ack-$.

5.2.3 Synthesis and Optimization Approach

Asynchronous Distributed Control Synthesis

Given: Resource-bound and scheduled CDFG.

Result: Optimized set of interacting controllers.

1. Apply global transformations to optimize controller-controller communication.
2. Extract one AFSM for each functional unit.
3. Apply local transformations for each AFSM to optimize controller-datapath communication.

Before considering the optimizing transforms, a basic unoptimized synthesis method is presented, which can always be used; it corresponds to the direct use of Step 2 in the synthesis flow. In an initial CDFG (see Figure 5.4 top), all RTL statements bound to the same functional unit (i.e., shown in the same column) will be controlled by a single functional unit controller. A number of constraint arcs run between distinct columns (RTL statements executed by different functional units). Each such constraint arc will be translated into a *global communication channel* between the corresponding functional unit controllers, in the target architecture (see Figure 5.4 middle). In particular, each communication channel is implemented by a single wire ("ready" signal). Note that while constraint arcs connect individual RTL statements in the CDFG, communication channels connect functional unit

controllers (which may implement more than one RTL statement).

Each functional unit controller is formally extracted from the CDFG (Step 2), and can be synthesized using *extended burst-mode* finite state machines (AFSM) [81, 110, 40, 38, 39, 102, 76]. Burst-mode is a commonly-used approach to designing Mealy-like asynchronous controllers. This step will be explained in detail below (Section 5.4). Each constraint (arc) is translated into a single wire (channel). Each CDFG node (e.g., RTL statement) is translated into a series of micro-operations in the controller, where the controller interacts with and sequences the datapath: setting of input muxes, performing operations, writing results, etc.

Using the above approach, however, the resulting implementation may be quite poor. Therefore, this thesis introduces optimizing transformations, both global (at the CDFG level, Step 1) and local (on the extracted AFSMs, Step 3), to further improve the design. Global and local transformations are introduced below (Sections 5.3 and 5.5). The global transformations (controller-controller) have two goals: reducing the number of communication channels between controllers, and improving performance (increasing concurrency, reducing critical path delays). These transformations are defined in such a way that they preserve the precedence order of the original CDFG.

After the global transformations, one controller per functional unit can then be extracted, as described above (Step 2; see Figure 5.15 for a fragment of the burst-mode controller for ALU1). Finally, local transformations (Step 3) improve the controller-datapath protocol for both speed and area: they remove or share wires, increase parallelism of operations, or reshuffle operations to initiate them earlier.

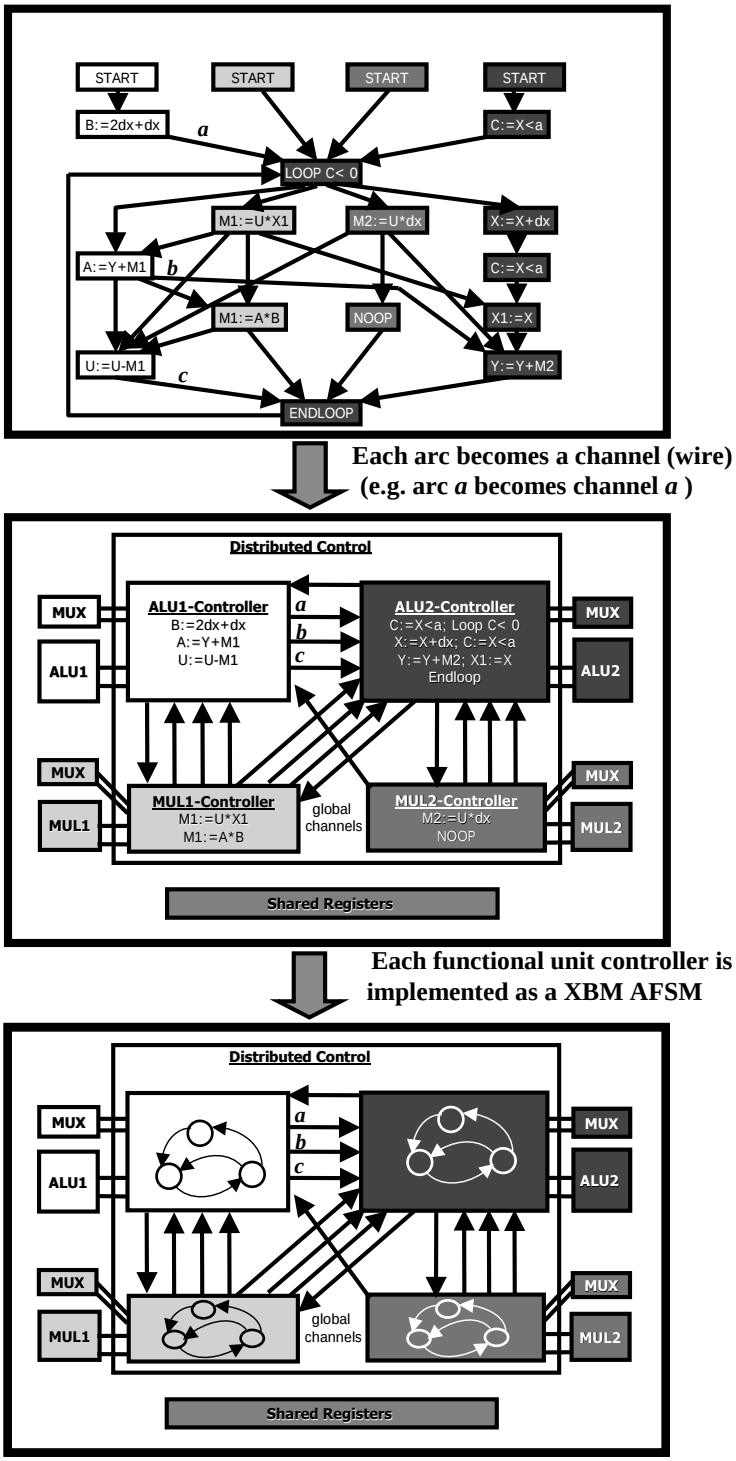


Figure 5.4: Unoptimized Synthesis Approach: from CDFG to Architecture to Distributed Controller Implementation

Note that the goal of this thesis is to introduce the new set of transformations, which can be used to optimize a system (much like the transforms of SIS for multi-level logic synthesis). Each of the individual transforms has been automated. In the future, this approach will be extended by creating *scripts* that automatically apply (or derive) a *sequence* of transforms to find an optimal implementation. Also, note that the assumed architecture (e.g. one controller per functional unit) somewhat limits the design space; this restriction will be relaxed in the future (e.g. multiple controllers per functional unit, or one controller for several functional units).

The following three sections now describe each of the three steps of our approach in detail.

5.3 Global Transformations: Controller-Controller

5.3.1 Overview

In this section, the set of global transformations to optimize controller-controller communication is described.

There are five global transformations: *loop parallelism* (GT1), *removal of dominated constraints* (GT2), *relative timing optimization* (GT3), *merging of CDFG nodes* (GT4), and *communication channel elimination* (GT5). All five transformations re-structure the CDFG.

Starting from the initial CDFG, the first four transformations manipulate constraint arcs in the CDFG and can be applied in any order. GT1 and GT4 modify constraint arcs in the CDFG to allow certain RTL statements to execute in parallel. The result is improved concurrency of the system. In contrast, GT2 and GT3

remove constraints in the CDFG that are not necessary for the correct operation of the distributed control. Since each constraint arc will become a channel, these two transforms therefore reduce the total number of communication channels. Reducing the number of channels may also simplify global system wiring and routing.

After the first four transformations have been applied, each remaining constraint arc is assigned to a distinct communication channel. Each communication channel connects the two functional unit controllers that correspond to the two CDFG nodes that the arc connects.

The final and powerful transformation then aims at eliminating as many communication channels between controllers as possible. The first and basic technique is *channel multiplexing* (GT5.1). This technique eliminates channels by sharing channels when there are multiple channels connecting the same set of units. The second and third techniques *concurrency reduction* (GT5.2) and *channel symmetrization* (GT5.3) both start from configurations where channel multiplexing is not directly applicable, and re-structure the communication channels so that channel multiplexing can then be applied.

5.3.2 GT1: Loop Parallelism

The goal of the "loop parallelism" transform is to improve concurrency of the distributed control. The transform re-structures the CDFG to allow more parallelism between *successive* iterations of a loop. More parallelism is achieved by loosening the synchronization through the ENDLOOP node and replacing it by more localized synchronization constraints.

As an example, compare Figures 5.5 and 5.6. In Figure 5.5, all four functional

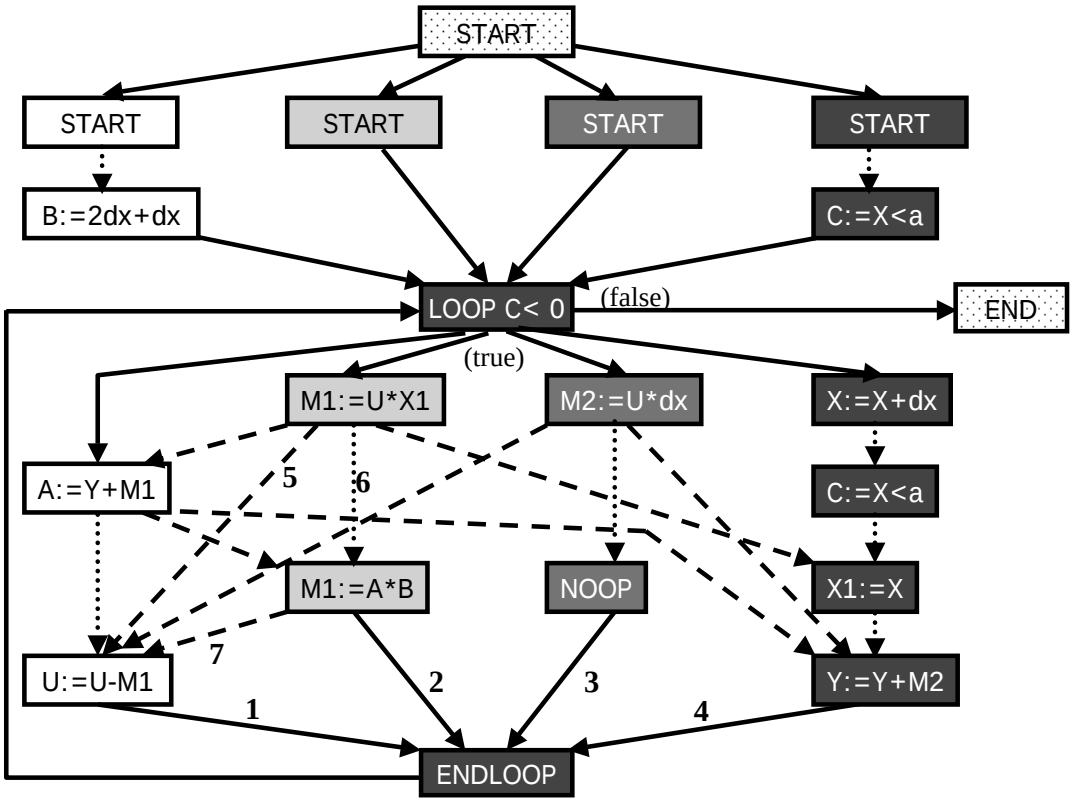


Figure 5.5: CDFG for DIFFEQ (repeat of Figure 5.1)

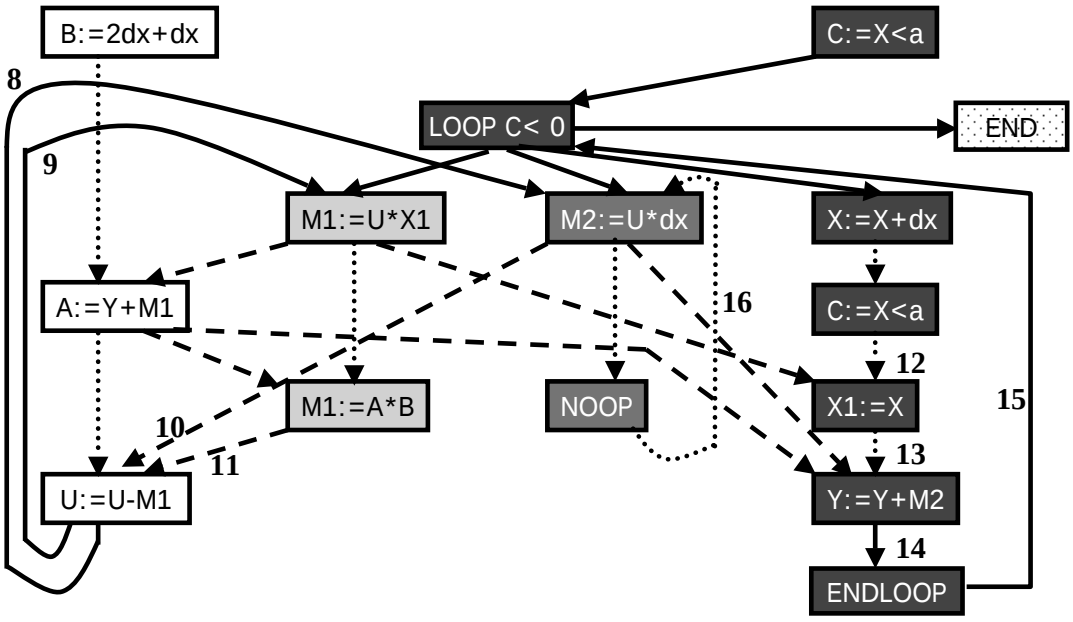


Figure 5.6: DIFFEQ CDFG after the Application of (GT1) Loop Parallelism and (GT2) Dominated Constraints. (*Note:* in order to clarify the presentation, the figure omits the START and END nodes.)

unit controllers are synchronized with an ENDLOOP node — by the arcs labeled 1 through 3. In contrast, in Figure 5.6 these arcs to the ENDLOOP-node are removed, and replaced by more localized synchronization constraints: the three *backward arcs* 8, 9, and 16. As a consequence, greater loop-level parallelism is achieved. Note that backward arcs are special arcs in the sense that they are ignored during the first execution of a loop body. Effectively, a backward arc is a pre-enabled constraint for the first iteration of a loop.

The "loop parallelism" transform consists of four steps in sequence.

Parallelize_Loop(LOOP,ENDLOOP):

- A. *Remove synchronization at ENDLOOP.* The goal is to allow the overlap of successive loop body executions. The solution is to remove all arcs in

the CDFG that are pointing to ENDLOOP; only the FU scheduling arc that connects ENDLOOP to its predecessor node in the FU schedule — of the functional unit ENDLOOP is allocated to — remains.

Compare again Figures 5.5 and 5.6. In step A the three arcs labeled 1, 2, and 3 are removed. The FU scheduling arc 4 remains.

- B. *Add backward arcs: loop body variables and functional units.* First, in the unoptimized scheme, the loop body includes data and register dependency constraints to avoid early reads and writes of registers. The goal of this step is to add constraints to extend these constraints across the loop boundary. For each variable in the loop body, backward arcs from all last instances (one write or multiple parallel reads) of the variable to the first instances (one write or multiple parallel reads) are added. Second, and similarly, a backward arc is added for each functional unit, from its last instance in the loop to the first instance in the loop.

  In the example, step B adds the three *backward arcs* 8, 9, and 16.

- C. *Add arcs: loop variable.* In the unoptimized scheme, the synchronization at ENDLOOP guarantees that the loop variable is updated before the LOOP-node examines it. In the optimized scheme, this requirement must be enforced explicitly. Thus, an arc from the last write of the loop variable in the loop body to the ENDLOOP-node is added. In the DIFFEQ example, step C does not need to add any constraint. The candidate arc from the node $C := X < a$ to the ENDLOOP-node is implied by the path of constraint arcs 12, 13, and 14. Thus, the candidate arc $(C := X < a, ENDLOOP)$ is a *dominated* constraint, and therefore not added.

- D. *Limit parallelism.* In the unoptimized scheme, global controller-controller communication is implemented by transition signaling without explicit acknowledgments. Effectively, there is always a chain of other events that provides an acknowledgment. After removing the arcs pointing to ENDLOOP in step A, this requirement may no longer hold for arcs from the LOOP-node to the first node of a functional unit, so multiple requests may be queued on the same wire, in the loop. The requirement is reinstated by adding arcs from the first use of each functional unit in the loop to the ENDLOOP-node. In effect, these arcs restrict parallelism to two consecutive iterations of a loop: thus, the next loop iteration can only be started after all functional units have completed the first operation in the current loop.

  In the example, step D, like step C, does not add any constraints. The first CDFG nodes of each functional unit — ALU1: $A := Y + M1$, MUL1: $M1 := U * X1$, MUL2: $M2 := U * dx$, ALU2: $X := X + dx$ — is already connected to ENDLOOP through a path of constraints.

There is one timing assumption that must hold if this transform is to be *safely* applied. This case concerns the final exiting from the loop. After applying the loop transform, the final execution of a LOOP-node examines the loop variable *while* other functional units may still be executing statements of the previous iteration. Hence, the LOOP-node "exits" possibly *before* the last iteration of the loop is finished. In this scenario, the transform is safe as long as a system timing constraint is satisfied: all loop components complete their operation before needed.

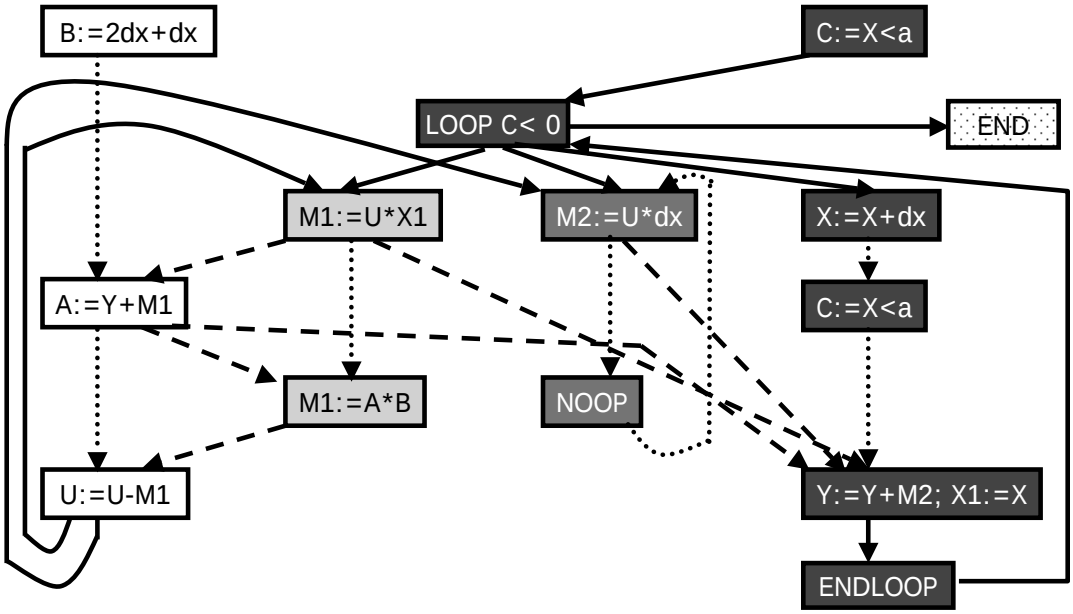


Figure 5.7: DIFFEQ CDFG after the Application of (GT3) Relative-Timing Optimization and (GT4) Merging of Assignment Nodes

5.3.3 GT2: Removal of Dominated Constraints

The goal of the transformation is to remove constraints that are *implied* by other constraints. A constraint arc from node a to node b is implied if there is a path of other constraints starting at node a and ending at node b. More formally, the constraint is removed if it is contained in the transitive closure of all other constraints.

As an example, consider constraint arc 5 in Figure 5.5. This constraint is implied by the path consisting of the two constraints 6 and 7. Thus, arc 5 can be removed, and therefore simplifying the global inter-controller communication — because arcs become channels, and so no channel is needed.

5.3.4 GT3: Relative-Timing Optimization

In asynchronous design, "relative timing" refers to the exploitation of knowledge about the relative occurrence of events in order to simplify design. Relative timing assumptions have been effective [96, 22]. However, existing approaches have mainly been limited to single controllers.

GT3 extends the use of relative-timing information to optimize *interacting* controllers. In the unoptimized case, a controller must wait for a set of "ready" ("request") signals from other controllers before it starts executing the current RTL statement. However, in some cases, the same controller always signals last. In such cases, the "ready" signals from "faster" controllers can be removed, and the controller only waits for the "slowest" one.

Both GT2 and GT3 remove constraint arcs in the CDFG that are not necessary for the correct operation of the system. However, while GT2 removes arcs based on the actual ordering of events enforced by the CDFG itself, GT3 removes arcs based on analyzing user-supplied timing information.

As an example, consider Figure 5.6. There are two constraint arcs from other controllers to $U := U - M1$, labeled 10 and 11. The former gets enabled after one computation — $M2 := U * dx$ — while the latter gets enabled after three computations — $M1 := U * X1$, $A := Y + M1$, $M1 := A * B$. Thus, the latter constraint arc (11) is "slower" under most assumptions. Hence, the former arc (10) is deleted in Figure 5.7 . A detailed timing analysis must be performed to determine where this transformation can be applied: it must be verified that the removed constraint arc is under no execution path the last to occur.

5.3.5 GT4: Merging of Assignment Nodes

Merging of assignment nodes is aimed at improving the speed of a functional unit controller. In the unoptimized scheme, each CDFG node is assigned to a functional unit. However, assignment nodes, i.e. $R_i := R_j$, simply examine and write registers, and thus do not use the functional unit. Such nodes can therefore be executed in parallel with the preceding or succeeding RTL operation assigned to the same functional unit.

As an example, compare Figures 5.6 and 5.7. In Figure 5.6, the two nodes $Y := Y + M2$ and $X1 := X$ are both assigned to the ALU2 functional unit. Since the node $X1 := X$ does not use the ALU2 functional unit, the assignment can be executed *in parallel* with executing the RTL-node $Y := Y + M2$. Thus, the two nodes are merged into one node $Y := Y + M2; X1 := X$ in Figure 5.7.

Before merging an assignment node with an adjacent node, a check is required to ensure no deadlock. In particular, there must not be a path of constraints between the two nodes to be merged — except for the trivial FU scheduling arc that connects both nodes. When merging assignment nodes, the set of incoming and outgoing arcs of the resulting node is the union of the sets of incoming and outgoing arcs of the two original nodes to be merged. If such a path existed, one of the outgoing arcs of the new node would be a predecessor to one of the incoming arcs, and thus a prerequisite for the node to "fire" could never be asserted, resulting in deadlock. However, in practice, it is rare that an assignment node cannot be merged with at least one of its adjacent nodes.

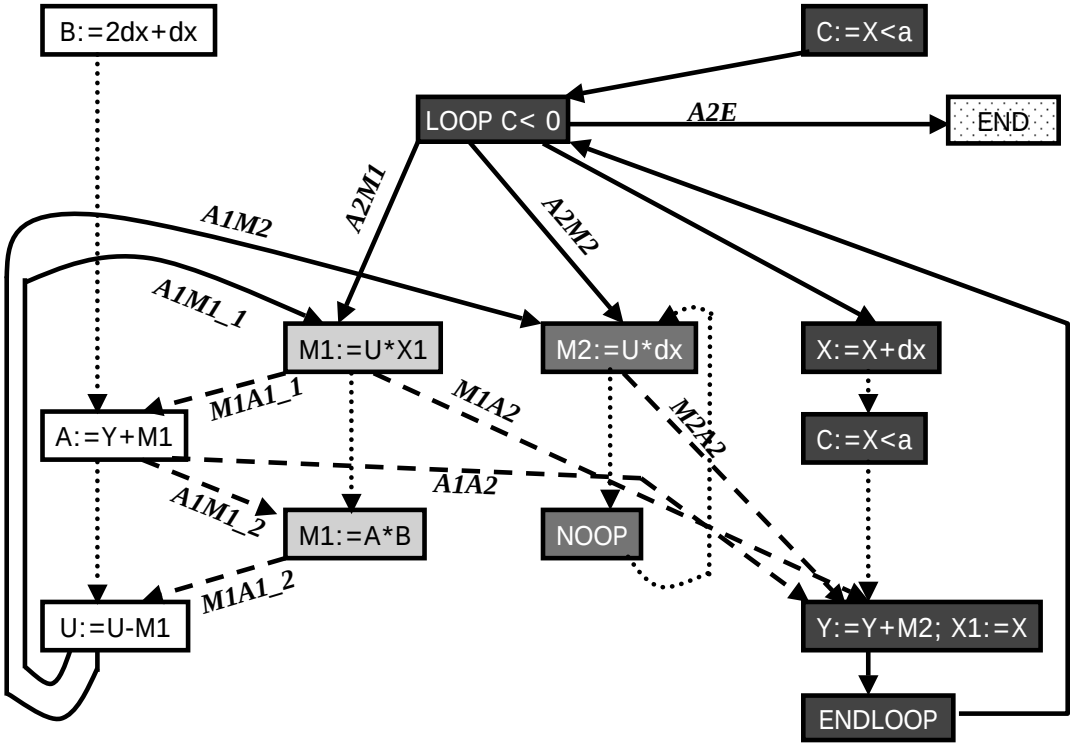


Figure 5.8: Communication Channels for DIFFEQ CDFG after GT1 - GT4.

5.3.6 GT5: Communication Channel Elimination

After the first four transformations GT1 through GT4 have been applied to optimize at the graph level (i.e. CDFG), each remaining constraint arc is assigned to a distinct communication channel. Each communication channel connects the two functional controllers that correspond to the two CDFG nodes that the arc connects (see Figure 5.8).

The goal of "communication channel elimination" is to delete as many communication channels between controllers as possible. The first and most basic technique is (GT5.1) *channel multiplexing*. This transform eliminates channels by sharing channels when there are multiple channels connecting the same set of units. After multiplexing has been applied, the two different events on the two

channels become different phases on the shared channel. The second and third techniques (GT5.2) *concurrency reduction* and (GT5.3) *channel symmetrization* both start from configurations where channel multiplexing is not directly applicable, and re-structure the communication channels so that multiplexing can be applied, by adding and replacing arcs.

Figure 5.9 gives a summary of the significant impact of the three GT5 transforms on simplifying communication. On the left side the communication channels before the application of GT5 transforms is shown, and on the right side the communication channels after several GT5 transforms — multiplexing, concurrency reduction, symmetrization — have been applied. In the example, GT5 transforms reduce the number of channels from ten to five, including two multi-way channels. The result is much simpler inter-controller communication. The CDFG corresponding to the left side of the figure is shown in Figure 5.8, and the CDFG corresponding to the right side is in Figure 5.10.

Each of the three transforms GT5.1 through GT5.3 is now explained in detail.

- **GT 5.1: Channel Multiplexing**

  The idea of "channel multiplexing" is to share communication channels to reduce the number of channels. Multiplexing can be applied to two channels that connect the same functional units and that are never concurrently active. After multiplexing, the two different events on the two channels typically become different phases on the shared channel.

  As an example, consider Figure 5.11, where a CDFG fragment is shown on the left side and the corresponding controller structure is shown on the upper right side. The CDFG fragment contains two nodes bound to ALU1 and

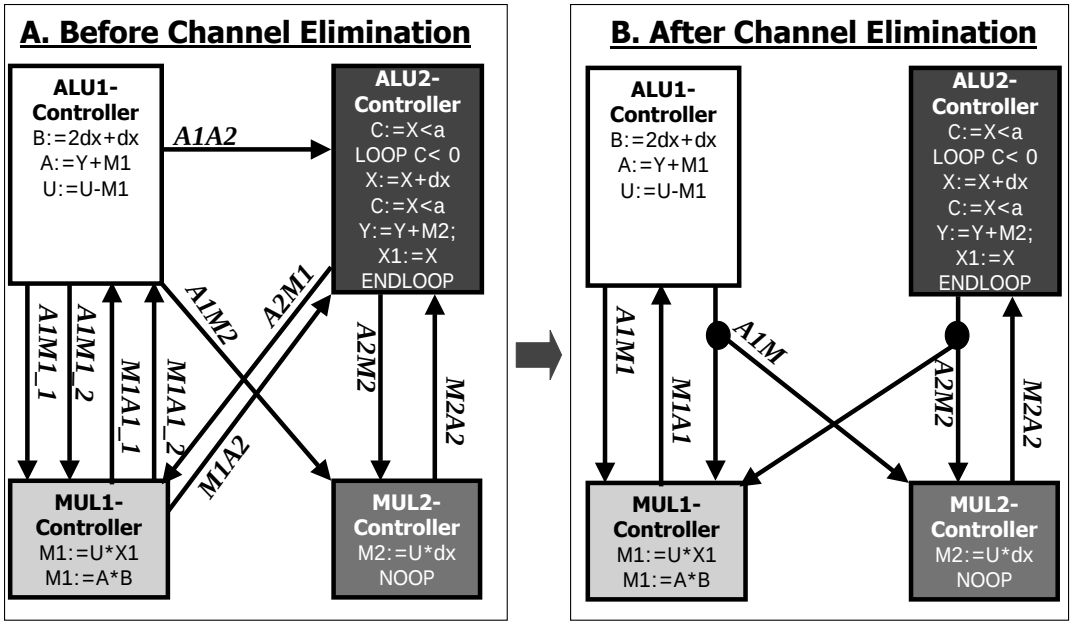


Figure 5.9: GT5: Channel Elimination for DIFFEQ Example.

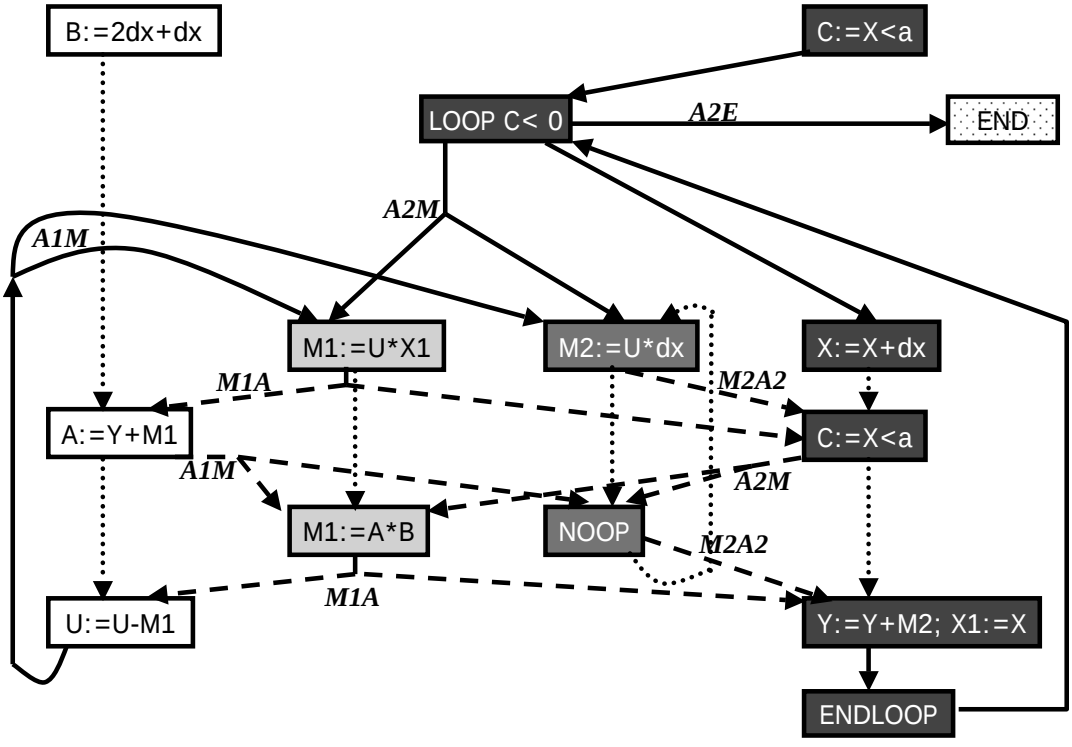


Figure 5.10: DIFFEQ CDFG After Channel Elimination.

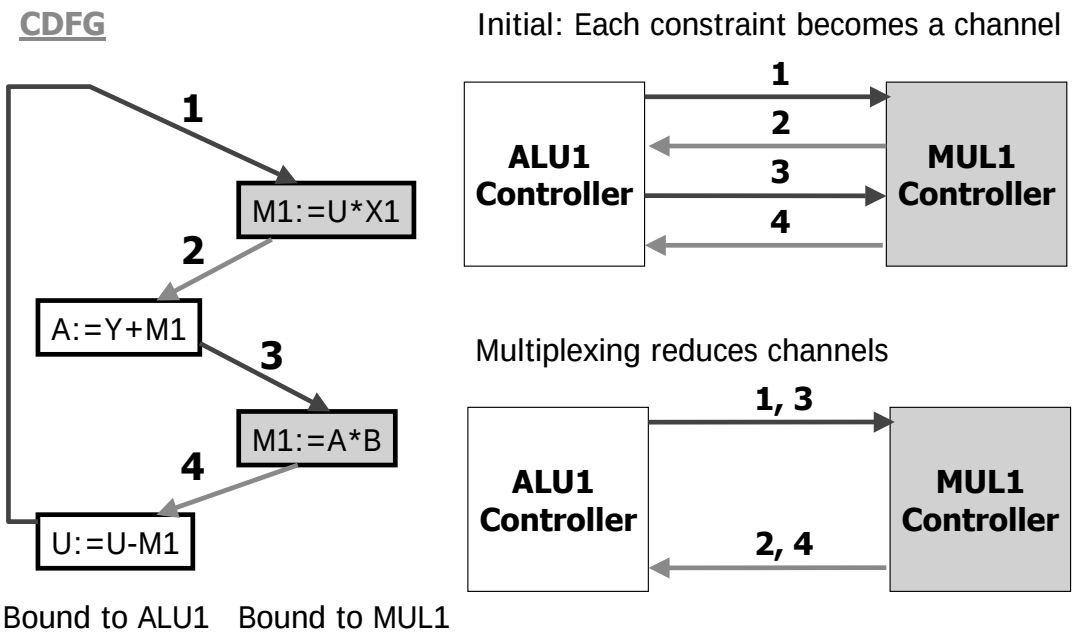


Figure 5.11: GT5.1: Multiplexing Constraints on Communication Channels

two nodes bound to MUL1. There are four arcs between the two functional units, and initially each one is implemented by a separate communication channel. Thus, there are two channels from ALU1 to MUL1, and two from MUL1 to ALU1. "Multiplexing" the two channels from ALU1 to MUL1 leads to sharing one communication channel (and thus a wire, since each channel becomes a wire) from ALU1 to MUL1 (bottom of right side). Similarly, the two channels from MUL1 to ALU1 are multiplexed. As a consequence, the number of channels is reduced from four to two.

- **GT 5.2: Concurrency Reduction**

  "Concurrency reduction" starts from a configuration where channel multiplexing is not directly applicable, and re-structures constraints so that mul-

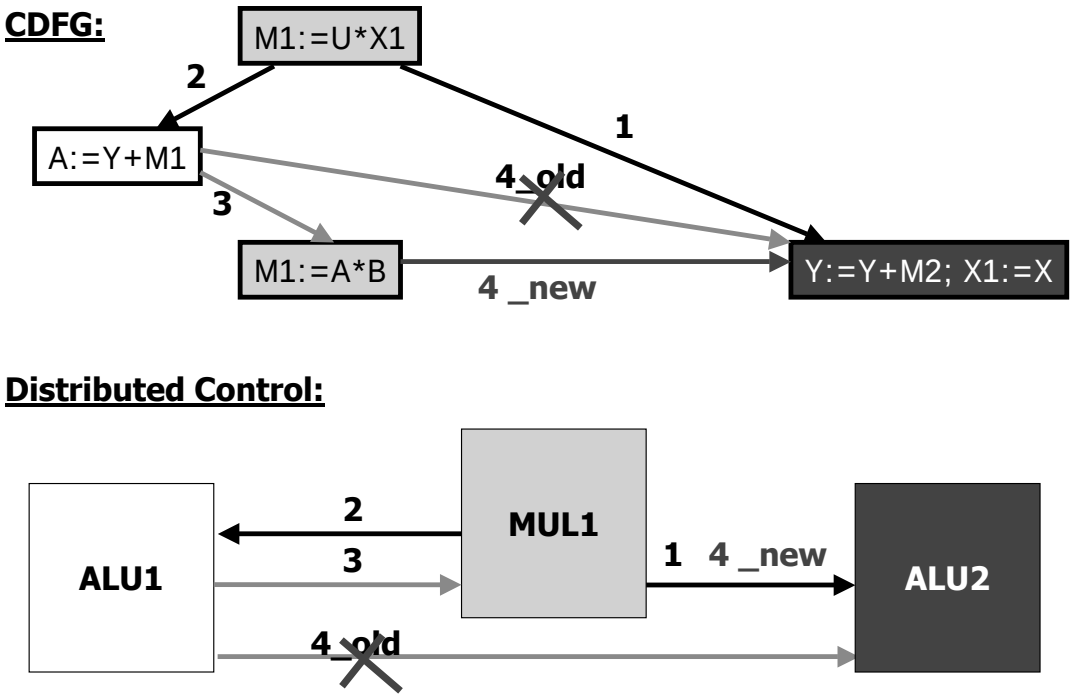


Figure 5.12: GT5.2: Concurrency Reduction

tiplexing can be applied. The transform replaces a *simple* constraint from a node a to a node c by *a chain* of *two other constraints*: a constraint from a to b, and a constraint from b to c. Thus, the additional hub may reduce the concurrency of the system (it possibly delays the start of executing node c), but it eliminates a channel by re-using an existing channel.

The goal of the transform is to apply it to non-critical constraints, and to replace a constraint in such a way with a chain that the resulting two constraints can be multiplexed with existing constraints. (If any of the two constraints already exists in the CDFG, then it is not added.)

Consider the CDFG in Figure 5.12. The constraint arc 4_{old} is replaced by the

existing arc 3 and a new arc 4_{new}. The new arc can be multiplexed with the arc 1 since both arcs connect the same functional units. Hence, the number of communication channels is reduced, and the overall communication structure is simplified: the direct communication channel between the leftmost (ALU1) and rightmost (ALU2) controllers has been eliminated, as shown in Figure 5.12 (bottom).

- **GT 5.3: Channel Symmetrization**

  Like GT5.2, "symmetrization" starts from a configuration where channel multiplexing is not directly applicable. Constraints are added to the CDFG so that multiplexing becomes possible.

  Unlike other transforms, the goal of symmetrization is to create *multi-way* channels. A multi-way channel connects a single CDFG source node to multiple CDFG destination nodes. Each node must correspond to a distinct functional unit. Events sent by the "sender" are seen by all receiving functional units. Given two sets of channels that have the same sending functional unit, but have overlapping but *not identical sets* of receiving functional units, the idea of the transform is to first make the receiving sets symmetric, by "safe addition" of arcs in the CDFG. Next, each set is transformed to a multi-way channel. Finally, the pair of multi-way channels is multiplexed.

  Figure 5.13 visualizes the symmetrization transform. Consider the three constraints 1, 2, and 3 in the CDFG, where 1 and 2 form a set of channels, and 3 is a singleton set. The first set connects ALU1 to MUL1 and MUL2, while the second set connects ALU1 to MUL1. The transform makes the two sets

symmetric by adding constraint 4_{added} to the CDFG, and also to the singleton set. The two sets become two multi-way channels connecting ALU1 to MUL1/MUL2 (see bottom of the figure), which are then multiplexed.

When symmetrizing channels, arcs must be added in a "safe" way to ensure no deadlock. Deadlock can arise when an added constraint forms a circular dependency which prevent units from executing. To detect deadlock, path analysis is performed on the CDFG for illegal cycles. Consider the addition of an arc from node a to b. This arc must lie within a single block due to the block-structuring requirement (Section 5.2.1). Within this block, a and b are partially ordered (in the initial CDFG): either a precedes b, a follows b, or a and b are unordered. If a follows b, no regular arc can be added, since a deadlock would result.[2] If deadlock occurs, symmetrization cannot be applied.

There is one small technical issue that must be considered: only arcs of the same "type" (regular vs. backward) can be combined into a multi-way channel. Recall that backward arcs can be thought of as "pre-enabled" (to set up the first iteration), while regular arcs are not. If the two types are combined in one multi-way channel, the backward arcs cannot be uniquely pre-enabled.

5.4 Individual Controller Extraction

After the global transformations have been applied, an asynchronous finite state machine (AFSM) is extracted for each functional unit controller. The extraction

[2]Backward arcs, which enforce data dependencies *between* loop iterations, will not cause deadlock.

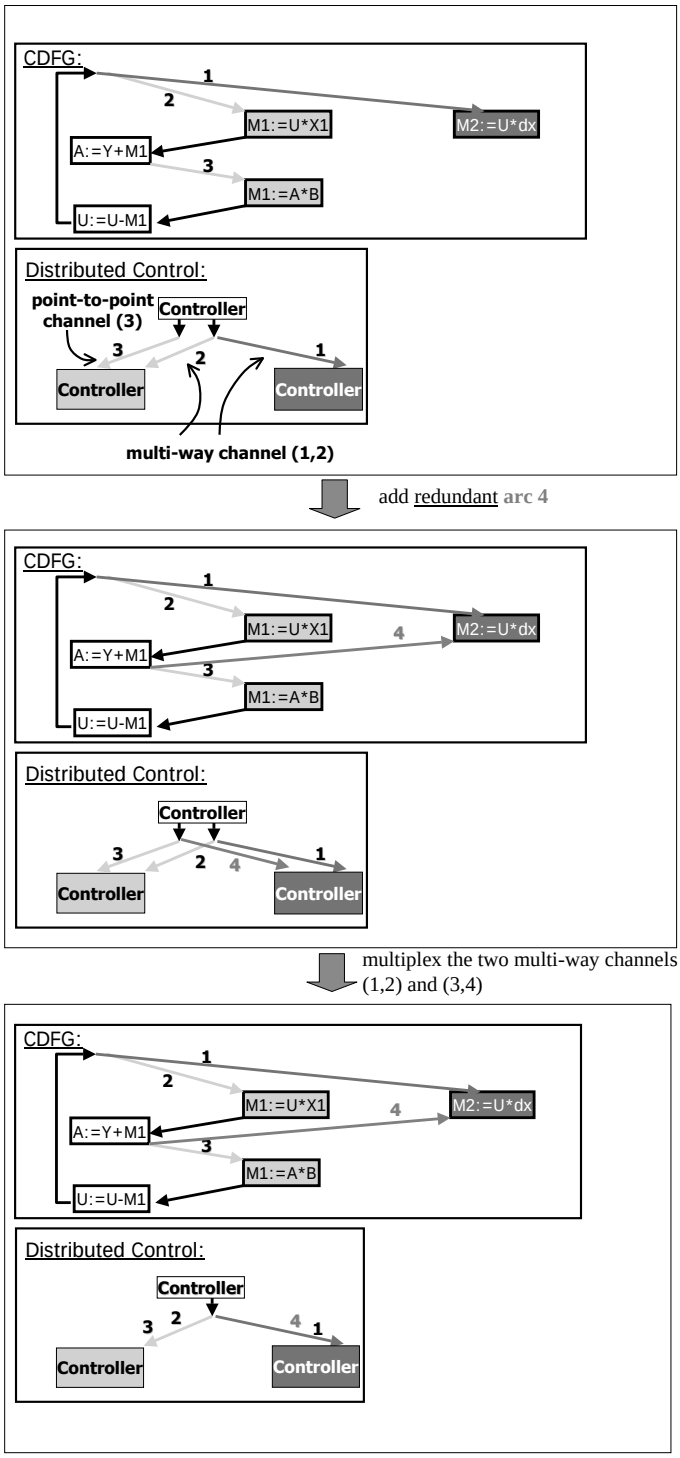


Figure 5.13: GT5.3: Channel Symmetrization – Before Symmetrization; After Adding 1 Constraint; Final Multiplexing of Symmetrized Channels

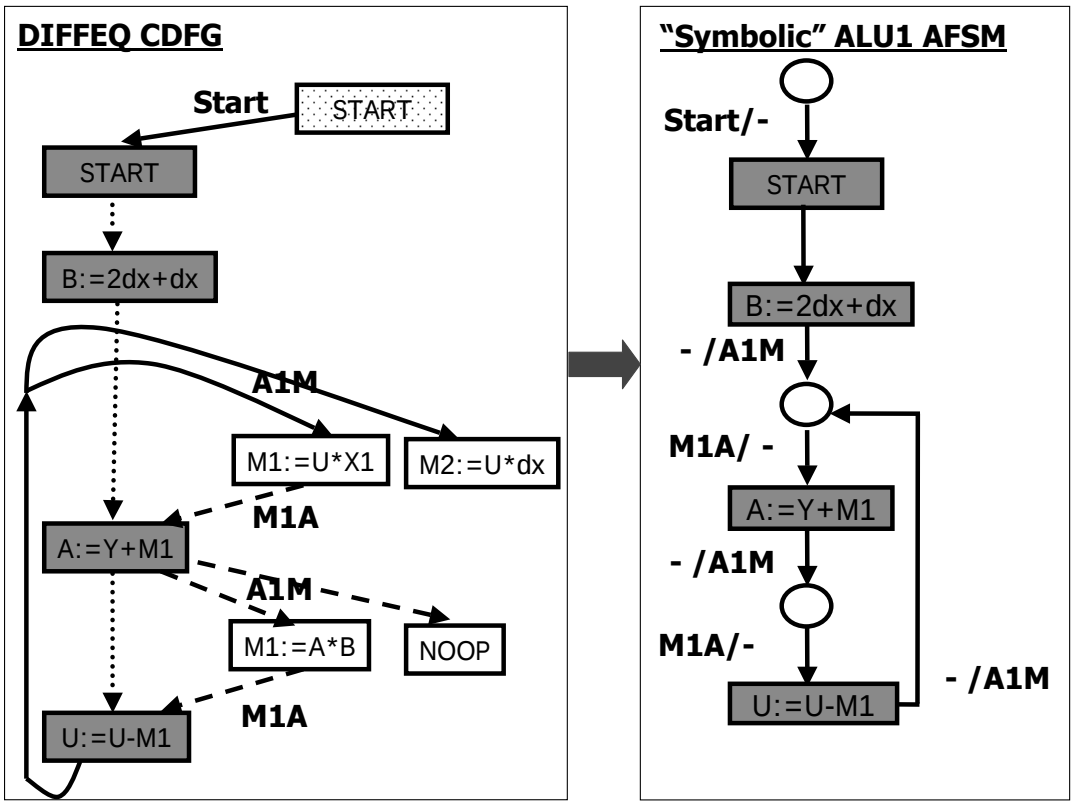


Figure 5.14: Burst-Mode Extraction: Creating an Initial Symbolic Burst-Mode Machine

algorithm is a direct deterministic translation from the CDFG (see Figure 5.10) into asynchronous Burst-Mode Controllers [81, 110, 40, 38, 39, 102, 76].

5.4.1 Overview

The proposed extraction method is based on a direct translation scheme for each CDFG node. As an example, consider Figure 5.14, which illustrates the extraction of the ALU1 controller. On the left side of the figure a partial CDFG that includes all nodes bound to ALU1 is shown, and on the right side is the corresponding "symbolic" AFSM. This AFSM includes one symbolic node for each CDFG node.

Each symbolic node is then expanded into a Burst-Mode fragment, implementing the operation.

The basic idea of translating a CDFG node — *expanding the corresponding symbolic node* — into a Burst-Mode fragment can be illustrated by using an RTL-type node as an example. In Figure 5.15, the left side shows the ALU1 controller, and its communication with other controllers and its datapath. The controller executes three RTL statements ($B := 2dx + dx$, $A := Y + M1$, and $U := U - M1$), and it is now waiting to execute the RTL node, $A := Y + M1$. The right side of the figure shows the Burst-Mode fragment corresponding to the RTL node $A := Y + M1$. This fragment will be explained in detail below.

A BM fragment for an RTL-type node implements the basic protocol: (a) wait for a set of ready signals ("requests") from other controllers, (b) perform the datapath operation, and finally (c) send "ready" signals ("dones") to other controllers to indicate that it has finished executing the RTL statement. "Ready" signals are single transitions on wires. In contrast, local communication with the datapath is based on a 4-phase protocol: $req+, ack+, req-, ack-$.

The given BM fragment in Figure 5.15 consists of a series of six state transitions to implement the micro-operations: (i) *wait for request and set input muxes*, (ii) *do operation*, (iii) *set register mux*, (iv) *write register*, (v) *reset local signals*, and (vi) *send done signals*. Each of the micro-operations (i) through (iv) is done by a $req+$ and $ack+$ pair, the first half of a 4-phase handshake. In the figure the micro-operation labels are placed between the corresponding $req+$ and $ack+$ pair. In (v) all req/ack-pairs are then re-set to 0 in parallel ($req-$, $ack-$). Micro-operation (ii) "do operation" includes two tasks: (a) selecting the operation to be performed by

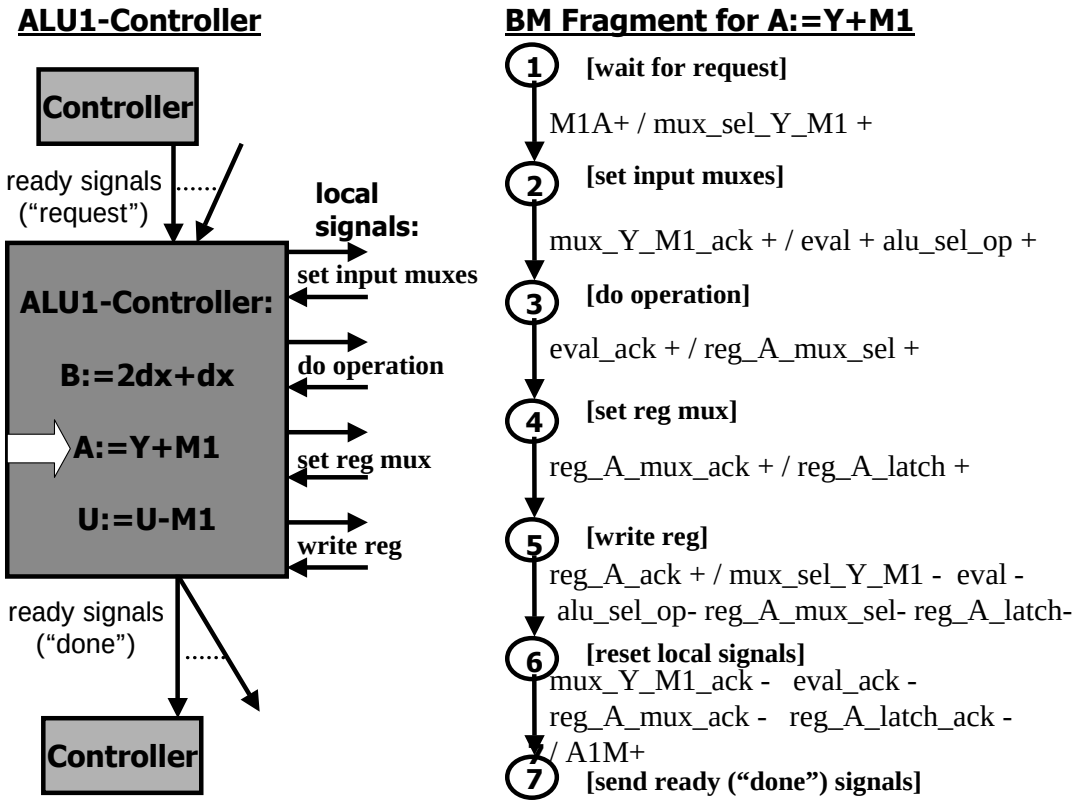


Figure 5.15: BM Expansion of RTL-node A:=Y+M1

the FU — FUs such as ALUs can execute multiple operations, and (b) initiating the execution in the functional unit.

The BM extraction method is implemented by an algorithm containing four steps. First, each CDFG node is directly translated into a BM fragment[3]. Second, BM fragments are stitched together to obtain a near-complete specification for the controller. Third, signal phases to global communication signals are assigned. There is one more final step to ensure that the BM specification may accept early requests. Each basic stitched template assumes all "ready" ("request") signals arrive just when needed. However, in reality, the system may be more concurrent

[3]Steps 1 and 4 may introduce features only available in *extended* Burst-Mode

and thus the fourth step modifies the BM specification to "back-annotate" the early arrival of requests.

As an example, Figure 5.15 shows the translation of a single CDFG node (for $A := Y + M1$) into a BM fragment with global signal phases assigned (e.g. *M1A+, A1M+*). The generated Burst-Mode controller for ALU1 is shown in Figure 5.20.

The four steps of the BM extraction algorithm are now introduced in turn.

5.4.2 Step 1: Translation of CDFG Nodes into BM Fragments

In Step 1, each CDFG node is translated into a BM fragment by using translation templates. There is one translation template for each type of CDFG node: regular RTL-node, IF-node, ENDIF-node, LOOP-node, ENDLOOP-node, START-node. Before explaining the different templates, the common issue of generating global ready signals is presented.

Global "Ready" Signals

All BM fragments include global ready signals to implement the basic protocol (cf. Section 5.2) for a CDFG node. A *simplified view* is as follows: (i) wait for "request" ready signals, (ii) interact with local datapath (if necessary), and (iii) send "done" ready signals to waiting controllers. In addition, the BM fragments must include a state transition [after (iii)] to the BM fragment corresponding to the CDFG node that is to be executed next by the functional unit. In fact, the next CDFG node may not be unique: the next CDFG node is determined by *which* "request"-signals arrive from other controllers.

The solution is to break with the simplified view and to generate BM fragments in a reversed way: (a) interact with datapath (corresponds to ii), (b) send "done" ready signals (iii), and (c) wait for ready signals to *decide* which is the next CDFG node to be executed. In (c) state transitions to all possible next CDFG nodes, i.e. their corresponding BM fragments, are generated (see Figure 5.16).

Templates

There is one translation template for each of the different CDFG nodes.

The template for RTL-nodes is shown in Figure 5.16. The local communication for an RTL node was already explained in 5.4.1. The global communication in the template is reversed: the global "request" ready signals are at the bottom of the template, as explained in the previous section. As a further optimization, when BM fragments are generated, only mux select signals that are necessary are included in the fragment. For example, if a register is only written by one functional unit, no mux is necessary, and thus no mux signal is generated.

Control-flow CDFG nodes (IF, ENDIF, LOOP, ENDLOOP, START) are translated by modified templates. There are two major differences. First, no local signals need to be generated. Second, in the case of LOOP and IF-nodes transitions for the two possible values of the level-sensitive decision variable need to be generated. Refer to Figure 5.17 which shows the Burst-Mode template for the LOOP node: Once the ready signals have arrived, the loop variable is inspected. If the value is 1, the "true" ready-signals are sent, and otherwise the "false" ready-signals.

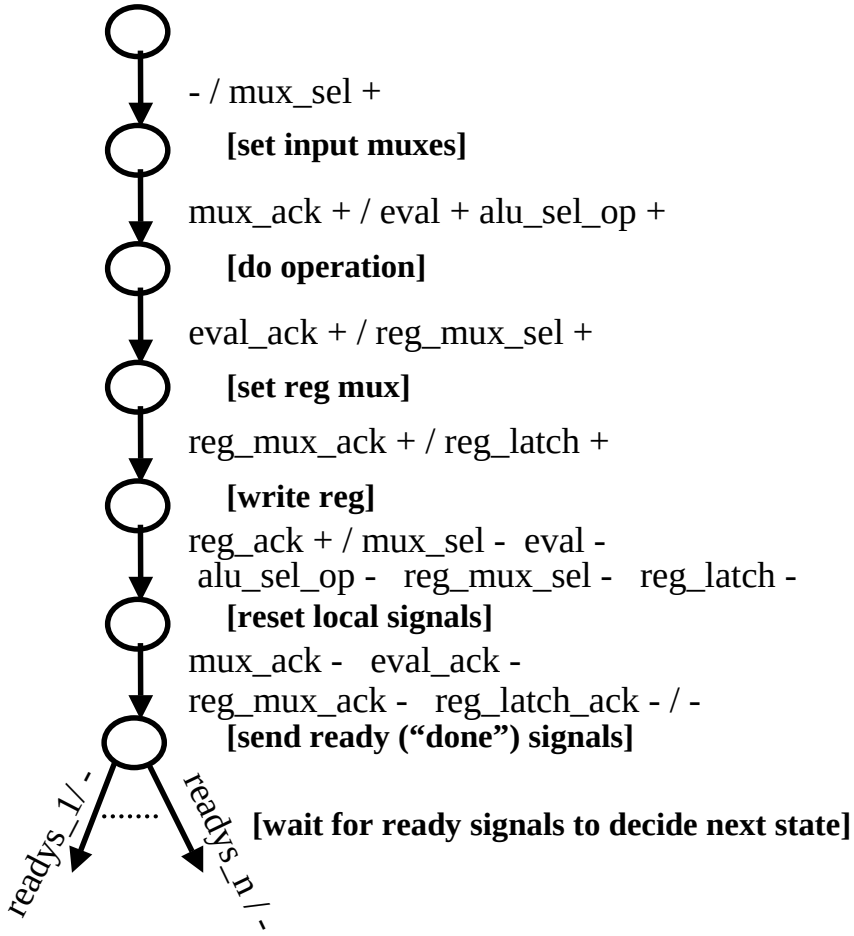


Figure 5.16: BM Template for RTL-node

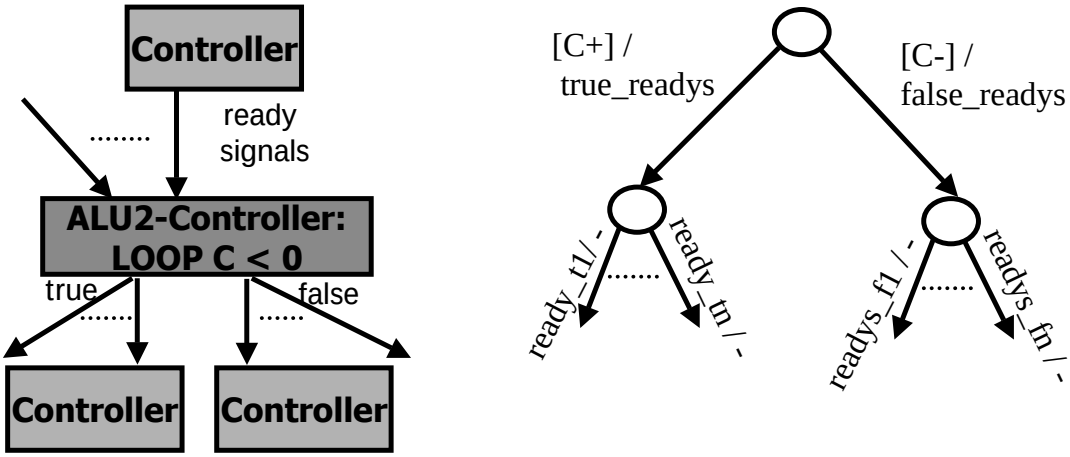


Figure 5.17: BM Template for LOOP-node

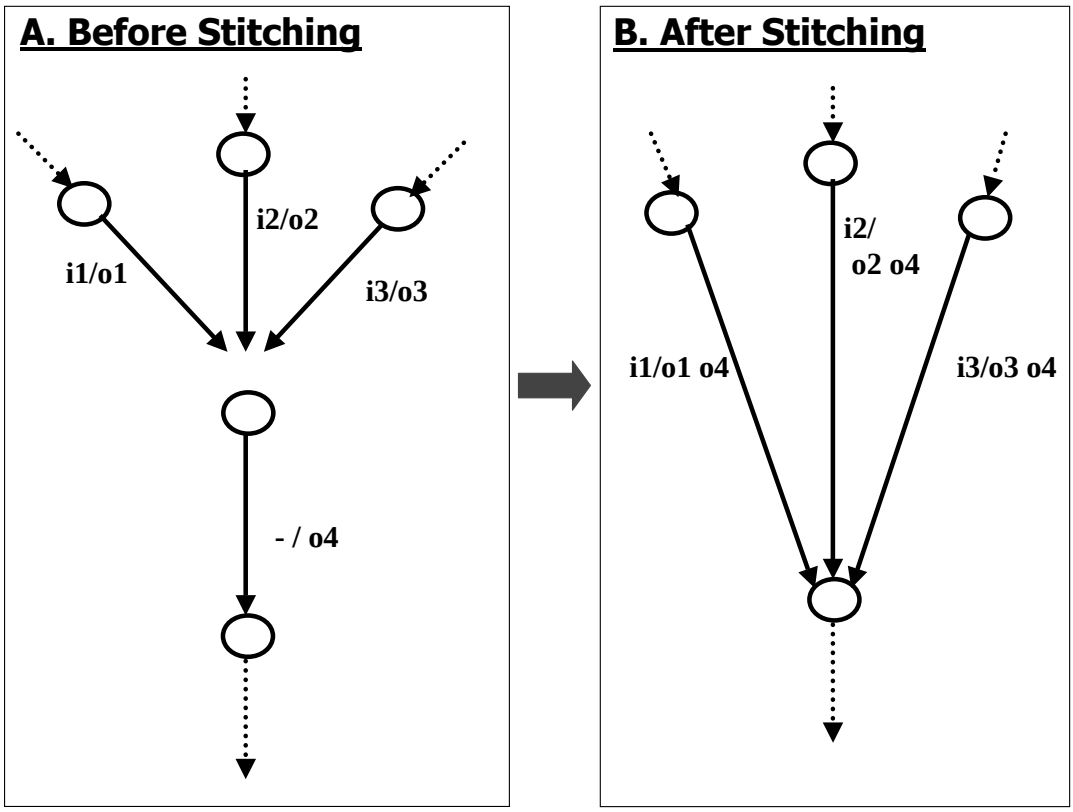


Figure 5.18: Stitching an RTL-node BM Fragment with Preceding BM Fragments.

5.4.3 Step 2: Stitching of BM Fragments

After Step 1 is completed, the BM fragments are stitched together. If BM fragments were simply "appended", this would violate the class of BM/XBM specifications, because the first transitions of the templates (see Figures 5.16 and 5.17) have input bursts, which are empty or do not include a compulsory signal (cf. Section 2.2.2). The solution is to merge the first transition of a fragment with the last transition of all previous fragments. There are two similar subcases, one for RTL BM fragments, and one for IF/LOOP BM fragments.

Stitching a BM fragment for a regular RTL node merges the empty-input-burst of its first transition with all immediately preceding transitions. This process

is visualized in Figure 5.18. In part A (bottom) the first transition of an RTL node fragment (labeled $-/o4$) is shown. Suppose there are three preceding fragments, corresponding to CDFG nodes that immediately precede the RTL node in some execution path. A valid specification is obtained by merging $-/o4$ with each of the transitions $i1/o1$, $i2/o2$, $i3/o3$. The result is shown in Figure 5.18B.

Stitching LOOP and IF-node BM fragments with preceding BM fragments merges the two transitions with conditionals (see Figure 5.17) with all preceding transitions. The LOOP and IF-node BM fragments generate input bursts that only include conditionals, i.e. these bursts do not include a compulsory signal. These input bursts would represent an XBM violation (cf. Section 2.2.2). The solution is to merge each of the two transitions with conditionals with all preceding transitions. This process is visualized in Figure 5.19.

5.4.4 Step 3: Assignment of Signal-Phases to Global Ready Signals

Steps 1 and 2 assign signal phases to local signals but leave global communication signals — "ready"-signals — untouched. As an example, consider Figures 5.16 and 5.17 which include "symbolic" ready-signals without signal phases. The goal of Step 3 is to assign signal phases to the "ready"-signals. That is, each symbolic "ready"-signal *ready* will become *ready*+ or *ready*−.

The assignment of signal phases to "ready-signals" is a straightforward encoding process and may involve unrolling the AFSM, i.e. replicating parts of the AFSM. Consider Figure 5.20, which shows the final Burst-Mode controller after phase assignment. There are two $M1A$ transitions: State 10 waits for an $M1A+$

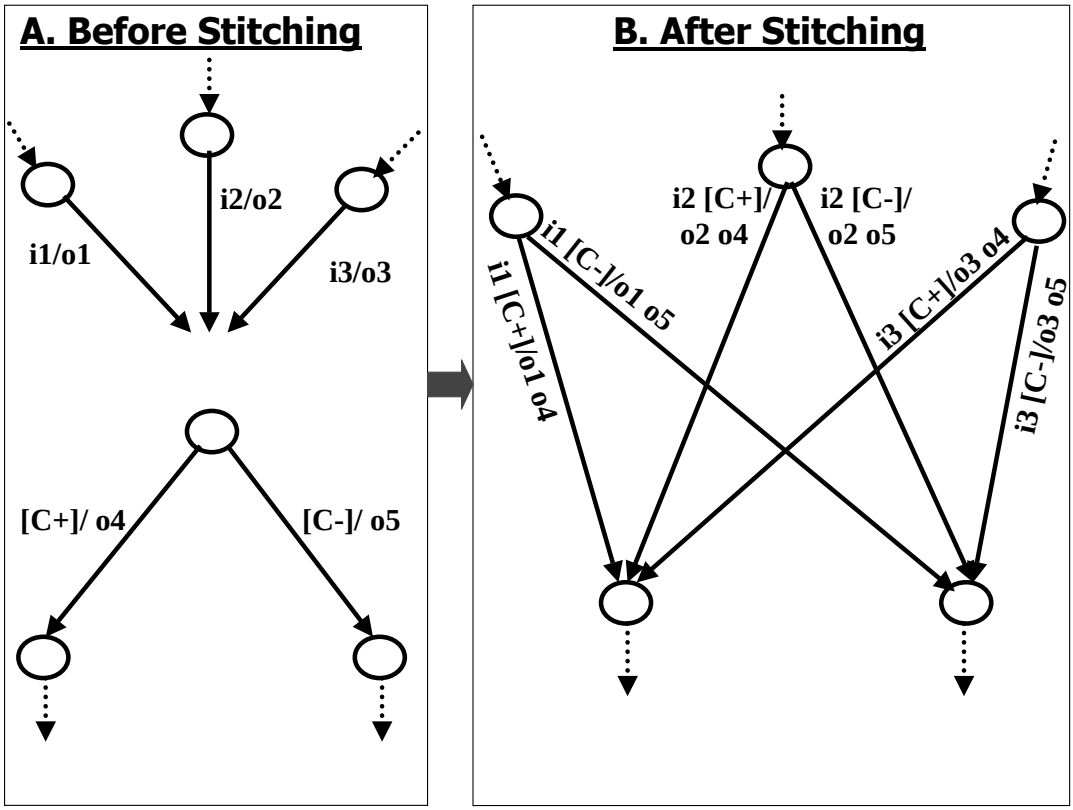


Figure 5.19: Stitching an IF/LOOP BM Fragment with Preceding BM Fragments

transition whereas state 18 waits for an $M1A-$ transition. In this example, signal phase assignment did not involve any replication.

5.4.5 Step 4: Back-Annotation To Allow Early Requests

In some cases, the AFSMs generated by Steps 1 to 3 are oversimplified: they ignore some of the possible concurrency of the system. In particular, each basic stitched template assumes all global "request" ready-signals arrive *just when needed* (in the first input burst). In reality, the system may be more concurrent: some requests may arrive during earlier operations. The goal of Step 4 is to modify the burst-mode controller specifications to "back-annotate" the early arrival of requests.

A Burst-Mode AFSM is back-annotated by augmenting transitions with *directed don't care signals*. Directed don't care signals are an important feature of extended Burst-Mode specifications (cf. Section 2.2.2). The following approach is applied to make the BM robust to early arriving global ready ("request") signals: For each transition in the BM specification, ready signals that may arrive early, i.e. during the transition but which are not part of the input burst of the transition, are added as directed don't cares to the input burst of the transition.

A reachability analysis is used to determine which ready signals may arrive early. This step is performed only once for the entire system. First, only the global ready-signals between BM controllers are considered. At any time, the values of these signals indicate the global state of the system. Second, a state space exploration of the system is performed to determine which global states are reachable. Finally, the resulting reachability information is then used to backannotate each transition in each BM specification with directed don't cares (to indicate potential

early arrival of ready signals).

5.5 Local Transformations: Controller-Datapath

5.5.1 Overview

The outcome of controller extraction (Section 5.4) is a BM specification for each functional unit controller. The global interaction between these controllers (via "ready signals") is now fixed.

Local transformations can now be applied to each of the individual controllers. The transformations aim at optimizing the handshake protocol between a functional unit controller and both its dedicated and shared datapath. In the unoptimized approach, communication between the controller and its datapath uses a series of 4-phase standard handshakes with req/ack wires: $req+, ack+, req-, ack-$. The goal is to reduce both the critical path delay and area of each controller.

There are five local transformations. *Move-up* (LT1) issues output signals earlier to shorten the critical path when it is safe to do so. *Move-down* (LT2) delays generation of output signals to provide opportunities for applying further local transforms. *Mux-preselection* (LT3) removes the selection of source input muxes or register muxes from the critical path. *Remove acknowledgments* (LT4) removes the ack wire that runs from a datapath unit to its functional unit controller when safe. *Signal sharing* (LT5) merges two distinct output wires into one forked output wire.

LT1 and LT4 improve the protocol based on *local timing-based optimization.* Local timing-based optimizations are similar to the global timing-based optimiza-

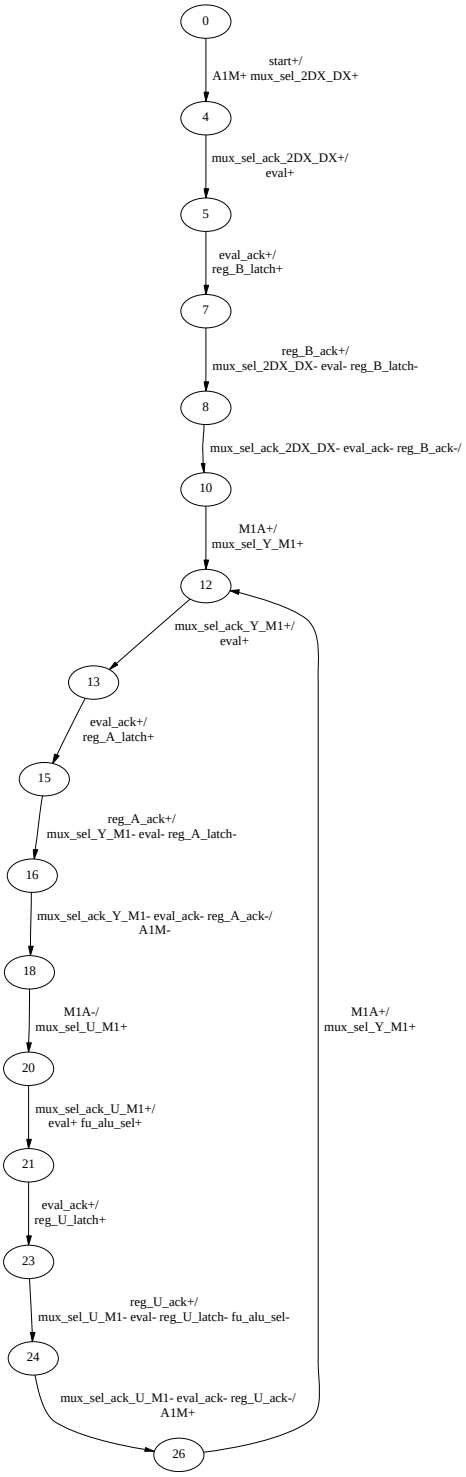


Figure 5.20: Unoptimized ALU1 BM Controller with Datapath Signals (from DIFFEQ).

tions. In fact, in the local case, more drastic optimizations due to the locality of the wiring may be possible. In contrast, LT2 and LT3 are not based on timing considerations. These transforms explore alternatives to the strict ordering of events in the protocol between the functional unit and its datapath, introduced by the template-based translation (cf. Section 5.4).

A quick overview of a typical series of local transforms is given in Figures 5.21 and 5.22. The series of local transforms is visualized at the protocol level. The top part in Figure 5.21 shows the unoptimized protocol. The bottom part of the figure visualizes the idea of removing acknowledgment wires. Next, the top part of Figure 5.22 improves the protocol even further by applying a "move-up" transform. Finally, Figure 5.22 bottom shows the idea of "mux-preselection", i.e. to remove the setting of muxes from the critical path.

Each local transform is now explained in detail.

5.5.2 LT1: Move-Up

The transformation "move-up" safely moves an output signal of the Burst-Mode controller to an earlier burst. The output can be either a local signal (triggering a micro-operation) or a global ready ("done") signal. The primary aim of the transform is to reduce the critical path delay to start the execution of an operation. A secondary aim of the transform is to provide further opportunities for applying other transforms. When "move-up" is applied to a global "done" signal, it effectively shortens the execution time of the current RTL operation.

As an example, consider the BM specification for the ALU1 controller in Figure 5.20, where the "move-up" transform can be applied to the final *A1M+*

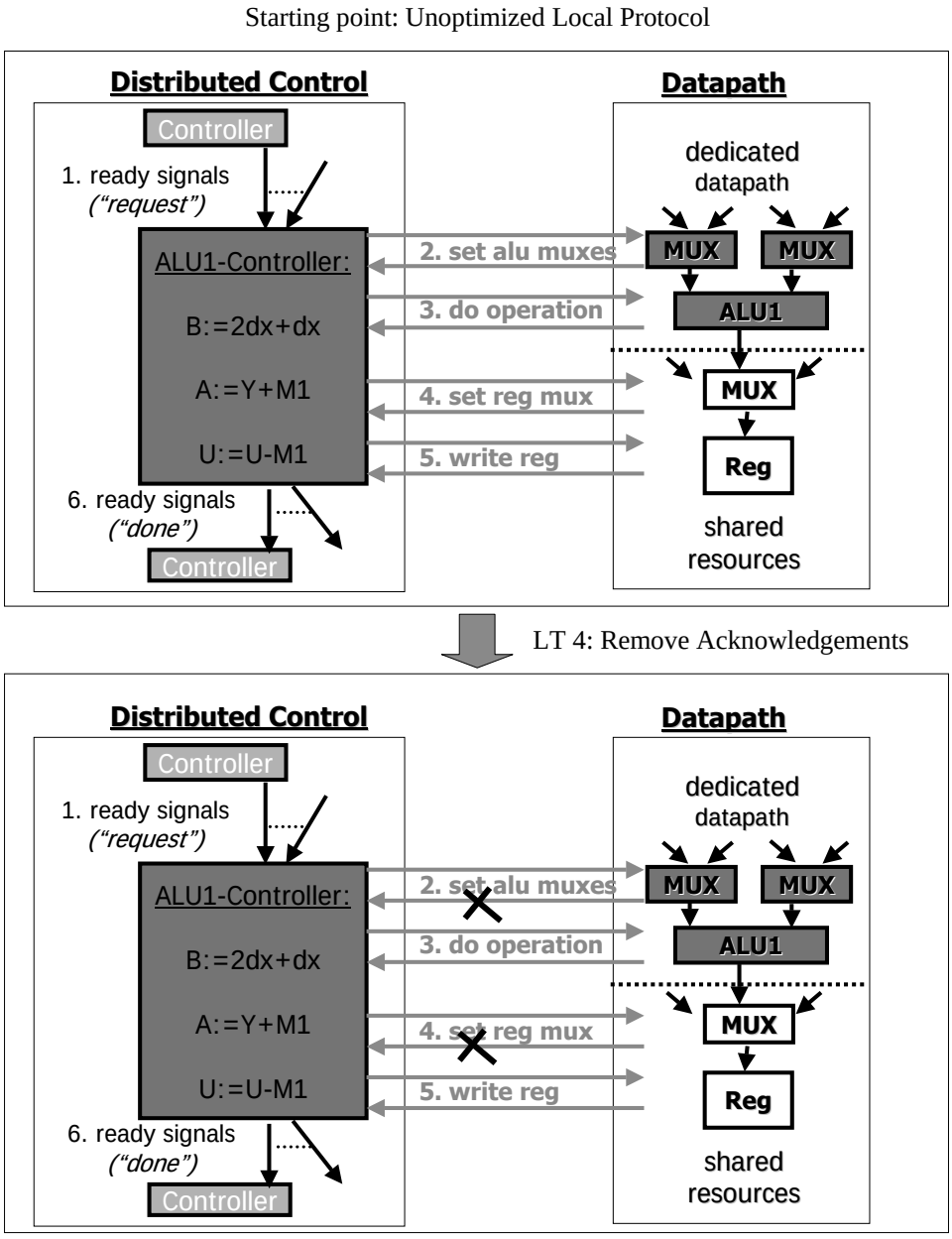


Figure 5.21: Local Transforms Example (Part A): (LT4) Remove Acknowledgments

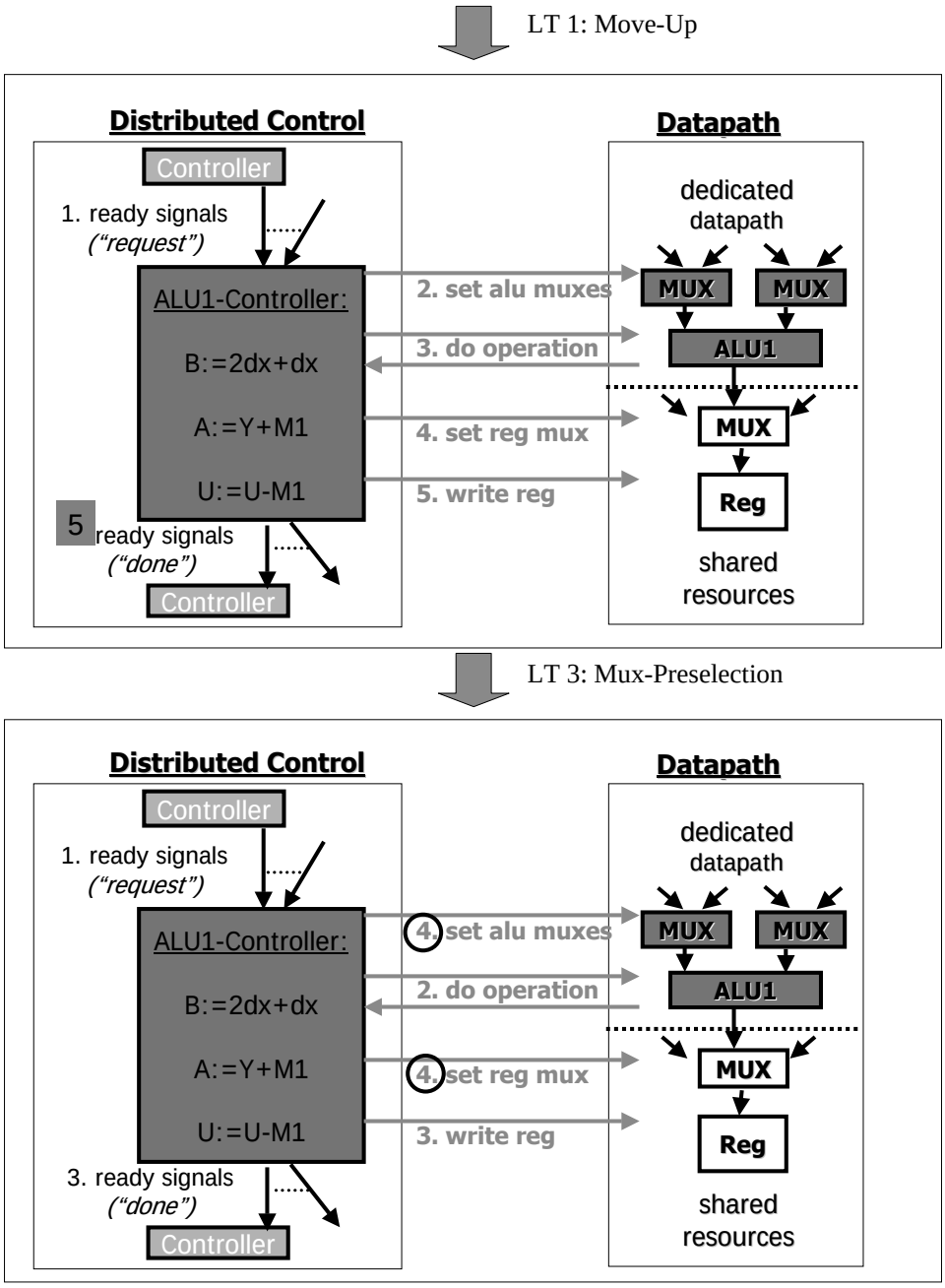


Figure 5.22: Local Transforms Example (Part B): (LT1) Move-Up, and (LT3) Mux-Preselection

ready ("done") signal in the transition from state 24 to state 26. Signal $A1M+$ indicates the completion of the current RTL operation, and is issued after latching B has been acknowledged (*reg_U_ack*), and even re-set. Under most local timing requirements, it is safe to "move-up" the global "done" signal $A1M+$. Figure 5.23 shows the modified controller after the move-up transform (see state transition from state 21 to state 24), i.e. latching the result (*reg_U_latch*) and sending a "done" signal to other controllers ($A1M+$) are now performed in parallel.

Move-up modifies the control specification as follows. The transform is given an output signal to move and the "destination burst" — indicated by giving some input signal contained within this burst. This input will serve as the new "trigger event" which will enable the output signal. In the actual algorithm, the output signal is moved step-wise, i.e. one transition up at a time until the specified input signal is found. If, during this process, a state with more than one in or outgoing transition is encountered, the signal is replicated and moved along each of these transitions.

The transform can only be safely applied if certain timing requirements are satisfied. In particular, it must be analyzed and verified that the moved signal does not enable an operation (either local or global) before it is safe: an early-enabled global RTL operation must have correctly updated operands in shared registers, and early-enabled local micro-operations must be safe under relative-timing assumptions.

5.5.3 LT2: Move-Down

The "move-down" transformation moves output signals that are not on the critical path to a later burst. The motivation is that moving signals to later bursts provides opportunities for the application of the "signal sharing" transform (LT5). "Move-down" is typically applied to the reset phases of local signals ($req-$, or $ack-$). The transformation is implemented analogously to "move-up"; instead of moving a signal to an earlier burst, the signal is moved to a later burst.

5.5.4 LT3: Mux-Preselection

Mux selection is often on the critical path for a system. Traditionally, muxes are selected on demand, that is, at the time they are needed. The idea of "mux-preselection" is to break with that concept and pre-select muxes early.

For a functional unit executing the current RTL operation, it is typically deterministic which RTL operation is next, so its controller can start *pre-selecting* the muxes for the next operation at the end of the *current* RTL operation's execution.

Consider Figure 5.20, where mux selection *mux_sel_Y_M1+* is performed after M1A+ is received, in the transition from state 10 to 12. In contrast in Figure 5.23, the selection is made at the *end* of the *previous RTL statement* (see transition from state 5 to 8), i.e. before M1A+ is received, thus reducing the latency for the RTL execution.

"Mux-preselection" is an important special case of "move-up" (LT1), but highlights a new aspect. The main idea of "move-up" was to move signals to earlier bursts within the state sequence corresponding to the datapath operations of the *current* RTL operation. "Mux-preselection" goes one step further. It "moves-up"

the mux select signal, effectively merging it with the datapath operations of the the *previous* RTL operation.

5.5.5 LT4: Remove Acknowledgments

The transform LT4 removes local acknowledgment wires that are not essential for the correct behavior of the controller. In the unoptimized approach, communication between the controller and its datapath uses a full 4-phase handshake protocol: $req+, ack+, req-, ack-$. The transform replaces the req/ack wire pair by just a req-wire whenever possible, with a simplified pulse-mode protocol: $req+, req-$.

User-supplied timing information is used to verify that the controller operates correctly once the acknowledgment wire has been deleted. In the simple case, the transformation leaves events in order, but simply deletes acknowledgments. It must be verified that removing the acknowledgment does not change the arrival order of events at any datapath or control unit. This scenario can be applied after "mux-preselection" (LT3), to delete unnecessary acknowledgments after pre-selecting and resetting the mux.

In a more aggressive scenario (see below), the transform may result in sequenced operations being merged into one step. In this case, more careful timing analysis is required to ensure that the micro-operations can complete correctly.

The LT4 transform modifies the BM control specification by simply removing the ack signal from each input bursts in which it appears. If this operation leaves an input burst empty, which is not allowed in BM specifications, a post-processing step is applied which merges such a transition with its preceding transition(s). This step is identical to the one explained in Section 5.4.3.

As an example, compare Figures 5.20 and Figure 5.23. In the latter, acknowledgments from registers and muxes have been removed.

5.5.6 LT5: Signal Sharing

Finally, "signal sharing" aims at reducing the number of outputs of a controller. Eliminating outputs is achieved by merging distinct control wires into a single forked wire. The forked wire then activates several datapath operations concurrently.

This transformation can be applied to two output wires that carry the same signal value at all times, i.e., if their corresponding signals appear in precisely the same set of output bursts in a BM specification across all RTL statements executed by the controller.

As an example, consider Figure 5.23. Here, the two signals *reg_U_latch* and *mux_sel_Y_M1* are always asserted and de-asserted in the same transitions. Therefore, the two local communication wires for latching and selecting can be collapsed into one, and a single forked wire used to control the two components: register U, and the input muxes to the functional unit that computes $Y + M1$. The result is indicated in Figure 5.24, where *reg_USSmux_sel_Y_M1* now corresponds to a single forked wire.

5.6 Related Work

In this section, the new distributed control synthesis method is briefly compared with other approaches. The three recent approaches that are most related to our approach are the works by Kim et. al [49], Cortadella and Badia [21], and the ACK

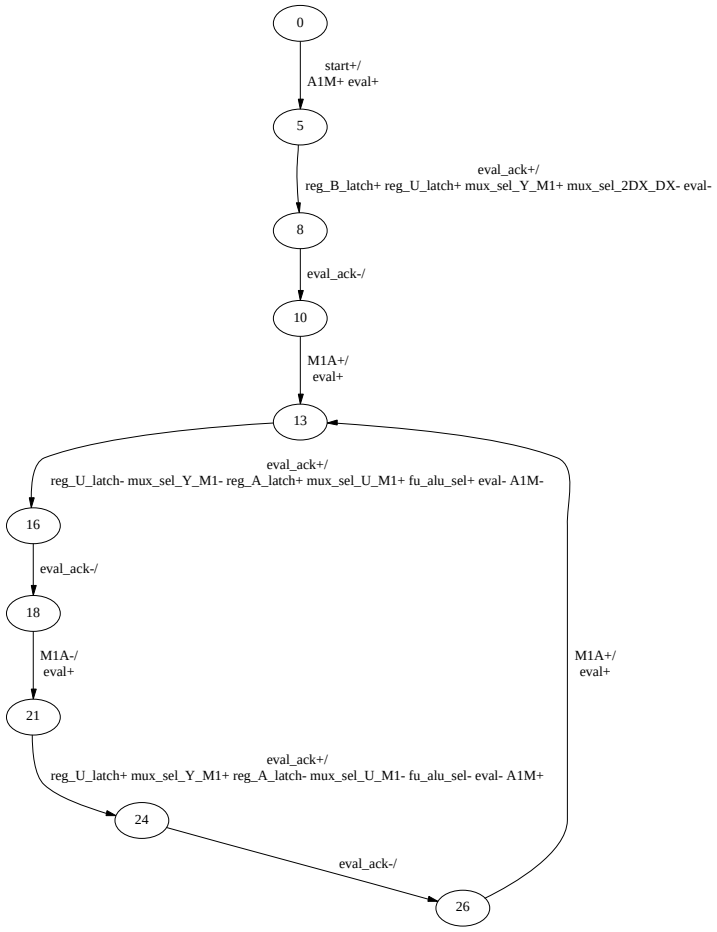


Figure 5.23: ALU1 Controller for DIFFEQ after LT1 through LT4.

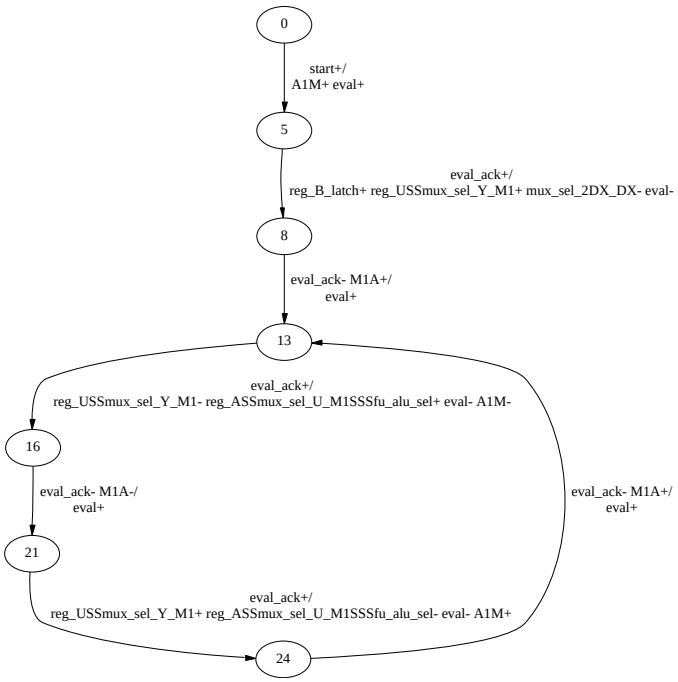


Figure 5.24: ALU1 Controller for DIFFEQ after LT5. (*Note:* Figures do not include the re-set transitions, in this case from state 24 to 0 and from state 13 to 0; including re-sets, there are 7 states and 9 transitions in total.)

system [54].

A critical distinction is that none of these approaches facilitates design-space exploration: in each approach, only a single deterministic implementation is generated for a given specification. In contrast, our approach provides a wide-ranging set of transforms which can be used to generate multiple implementations. Furthermore, unlike these earlier approaches, many of these are powerful restructuring optimizations.

Another fundamental point is that, of the related approaches, only Kim et. al's [49] method is based on Control-Data Flow Graphs, like our method. Their method was developed concurrently with ours and, as indicated below, has some important drawbacks compared to ours. Of the remaining approaches, Cortadella and Badia only consider Data Flow Graphs, and the ACK system is based on Petri Nets.

5.6.1 Kim et al.

Kim et al. [49] have proposed a synthesis method for asynchronous systems from Control-Data Flow Graphs. Their approach also produces distributed control, but subdivides sub-controllers for each datapath block further than our approach, assigning a sub-sub-controller to each of the processes (or CDFG nodes) bound to a functional unit.

In particular, the approach is based on the idea of having different types of controllers for (a) executing a single CDFG node and (b) controlling the order of execution of several CDFG nodes. The generated controllers form a hierarchy, which includes four different types of controllers:

1. one controller for each node in a DFG: to control execution of the node;
2. one controller for each DFG: to control the order of execution of nodes in a DFG;
3. one controller for each control construct such as IF and WHILE;
4. one controller that controls the order of execution of a number of controllers corresponding to DFGs, and IF/WHILE.

Our approach has several advantages over Kim et al.'s approach, beyond the fact that we facilitate design-space exploration. First, some of our aggressive transforms, such as Loop Parallelism, are not included in their approach and would seem very hard, if not impossible, to incorporate in the context of their work. A second major limitation of Kim's approach is that only one Data-Flow subgraph can be active at any time in a given CDFG specification; such a restriction does not hold for our input specifications, and thus more concurrent specifications can be handled.

5.6.2 Cortadella and Badia

Cortadella and Badia [21] have proposed a synthesis method that starts from Data Flow Graphs and synthesizes the control unit in such a way that each datapath block is controlled by a dedicated sub-controller. Thus, unlike our approach, the registers have their own associated sub-controllers.

An important advantage of our approach over Cortadella/Badia's is that they consider only Data Flow Graphs, i.e. no control structures, while our approach can handle conditional statements and loops.

5.6.3 ACK

Finally, synthesis methods have been proposed for large-scale systems based on Petri Nets, such as ACK [54]. These approaches derive a circuit implementation by first decomposing the concurrent Petri net specification into synthesizable parts, and then re-use the synthesis methods for single asynchronous controllers presented in Chapter 2.

Beyond the fact that we facilitate design-space exploration, our approach has two important advantages over ACK. First, ACK can only handle a limited form of concurrency in Petri net specifications: it assumes fork-join pairs. Second, ACK often needs manual intervention in the partitioning process, i.e. it is not completely automated.

5.7 Experimental Results

A prototype version of the presented method has been implemented in C++. As a case study, the method is applied to the well-known *differential equation solver* synthesis benchmark. A highly-optimized implementation was manually designed by Yun et al. [109]. Their circuits used many aggressive optimizations which have been inaccessible to existing asynchronous CAD tools.

Our automated tool has been applied to this example in three experiments, as shown in Figure 5.25. The *unoptimized* controllers were generated directly from the original CDFG with no global or local transformations applied; the *optimized-GT* controllers only after the application of global transformations, and the *optimized-GT-and-LT* controllers after application of both sets of transformations. The final

	#comm. channels	ALU1		ALU2		MUL1		MUL2	
		#states	#trans	#states	#trans	#states	#trans	#states	#trans
unoptimized	17	26	29	45	52	21	24	12	14
opt.-GT	5	16	18	26	32	12	14	8	10
opt.-GT-and-LT	5	7	9	11	13	6	6	4	5
YUN (manual)	5	7	9	14	16	4	4	3	3

Figure 5.25: State Machine Comparison: New Approach vs. Yun (manual)

specification of our ALU1 controller for the *optimized-GT-and-LT* case is shown in Figure 5.24.

Column 1 of Figure 5.25 compares the number of communication channels. For the DIFFEQ example, the number of channels was reduced from 17 (*unoptimized*) to 5 (*optimized-GT*), thus showing the impact of the global transformations.

Columns 2 through 9 focus on the state machines of the four functional unit controllers: ALU1, ALU2, MUL2, and MUL2. For each of the four controllers, the impact of the transformations is immense. For example, for ALU2, the number of states and transitions are reduced from 45 to 11 and 52 to 13, respectively. In comparing the final *optimized-GT-and-LT* controller specifications to Yun's, on average the specifications are comparable in terms of number of states and transitions. In particular, while our controllers are slightly worse for the small designs (MUL1 and MUL2), they are at least as good as Yun's for the larger ones (ALU1 and ALU2). In ALU2, our controller is actually smaller than Yun's (11 vs. 14 states and 13 vs. 16 transitions), which is striking, considering the amount of work that was put into designing Yun's controllers.

Figure 5.26 compares the gate-level implementations of our best experiment (*optimized-GT-and-LT*) with Yun. All functions are implemented by two-level logic. For each of the methods, the number of products and literals are listed. For ALU1

	Yun (manual)		New Method	
	#prod	#lits	#prod	#lits
ALU1	18	110	14	83
ALU2	46	141	40	113
MUL1	19	41	11	30
MUL2	10	15	8	18
total	93	307	73	244

Figure 5.26: Gate-Level Comparison: New Approach vs. Yun (manual)

MINIMALIST [38, 39] was used, but for the other controllers we used 3D [110] to synthesize the burst-mode specification, since only ALU1 is a "pure" burst-mode specification; the other ones are extended burst-mode specifications, and currently cannot be handled by MINIMALIST. Unfortunately, 3D does single-output logic minimization only, and thus does not share products among functions as MINIMALIST does.

Figure 5.26 clearly shows that our approach of applying systematic transforms leads to very efficient implementations: the total number of literals is reduced by almost 30% when compared to Yun's controllers.

5.8 Conclusions

A key bottleneck in the synthesis of large-scale asynchronous systems has been the lack of high-quality CAD tools which allow design-space exploration. The proposed method is the first asynchronous approach to introduce and automate a wide-ranging and powerful set of optimizing transformations, which allow systematic design space exploration.

The transforms implement techniques such as the exploitation of global relative timing assumptions and loop parallelism, channel multiplexing and sym-

metrization, and the pre-selection of muxes. They include aggressive timing- and area-oriented optimizations, several of which have not been previously supported by existing asynchronous CAD tools. We have shown that this set of transformations is powerful enough to derive controllers that are similar to or even better — up to 30% less area — than controllers that have undergone a labor-intensive manual design to make them highly-optimized.

Algorithmic heuristics and scripts based on the set of transformations presented in the thesis are forthcoming. We also plan to broaden the targeted architecture to allow multiple controllers per functional unit, as well as one controller for several functional units.

Chapter 6

Conclusions

In this thesis, several new efficient CAD techniques for the design of asynchronous control circuits are proposed. The contributions include *(i)* two efficient algorithms for hazard-free two-level logic minimization, both heuristic and exact, and *(ii)* a new synthesis and optimization method for distributed control from Control-Data-Flow-Graphs. The first contribution is targeted to optimizing logic for individual controllers. The second contribution is targeted to optimizing large-scale asynchronous systems.

Algorithms for Hazard-Free Two-Level Logic Minimization

Two new minimization methods for multi-output 2-level hazard-free logic minimization have been presented: Espresso-HF, a heuristic method based on Espresso-II, and Impymin, an exact method based on implicit data structures.

Although Espresso-HF is a heuristic minimizer, it almost always obtains an exactly minimum-size cover. Espresso-HF also employs a new fast method to check for the existence of a hazard-free solution, which does not need to generate

all prime implicants.

IMPYMIN performs exact hazard-free logic minimization nearly as efficiently as synchronous logic minimization by incorporating state-of-the-art techniques for implicit prime generation and implicit set covering solving. IMPYMIN is based on the new idea of incorporating hazard-freedom constraints within a synchronous function by adding extra inputs. The proposed technique may very well be applicable to other hazard-free optimization problems, too.

Both tools can solve all examples that are available. These include several large examples that could not be minimized by previous methods[1]. In particular both tools can solve examples that cannot be solved by the currently fastest minimizer HFMIN. On the more difficult examples that can be solved by HFMIN, ESPRESSO-HF and IMPYMIN are typically orders of magnitude faster.

Synthesis and Optimization for Distributed Control from Control-Data-Flow-Graphs

This thesis contributes a new approach for the automated synthesis and optimization of large-scale asynchronous systems. Unlike existing approaches for asynchronous system synthesis which are strictly deterministic and "template-based", the proposed approach is the first to introduce, formalize and automate a wide-ranging and powerful set of transformations, which can be used for design-space exploration.

The new method starts with a given scheduled and resource-bounded Control-Data Flow Graph (CDFG) [66]. Global transforms are first applied to the entire

[1] In publications on the 3D method (see e.g. [111, 108]), note that several of these examples appear but only *single-output* minimization is performed.

CDFG, unoptimized controllers are then extracted, and, finally, local transforms are then applied to the individual controllers. The result is a highly-optimized set of interacting distributed controllers.

As a detailed case study, the transformations are applied to the well-known *differential equation solver* high-level synthesis benchmark [109, 66]. A highly-optimized asynchronous implementation by Yun et al. [109] was manually designed, using a number of aggressive timing- and area-based optimizations. Such an implementation cannot be obtained using existing CAD tools. It has been demonstrated that a very similar optimized design can be simply and automatically derived through systematic application of the new transformations.

In the future, this approach will be extended by creating *scripts* that automatically apply (or derive) a *sequence* of transforms to find an optimal implementation. Also, note that the assumed architecture (e.g. one controller per functional unit) somewhat limits the design space; this restriction will be relaxed in the future (e.g. multiple controllers per functional unit, or one controller for several functional units).

Bibliography

[1] Brandon M. Bachman, Hao Zheng, and Chris J. Myers. Architectural synthesis of timed asynchronous systems. In *Proc. International Conf. Computer Design (ICCD)*, 1999.

[2] Rosa M. Badia and Jordi Cortadella. High-level synthesis of asynchronous systems: Scheduling and process synchronization. In *Proc. European Conference on Design Automation (EDAC)*, pages 70–74. IEEE Computer Society Press, February 1993.

[3] P.A. Beerel and T. Meng. Automatic gate-level synthesis of speed-independent circuits. In *ICCAD-1992*.

[4] Peter A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits*. PhD thesis, Stanford University, 1994.

[5] J. Beister. A unified approach to combinational hazards. *IEEE Transactions on Computers*, C-23(6), 1974.

[6] M. Benes, S.M. Nowick, and A. Wolfe. A fast asynchronous huffman decoder for compressed-code embedded processors. In *Proc. International Sympo-*

sium on Advanced Research in Asynchronous Circuits and Systems. IEEE Computer Society Press, March 1998.

[7] C. H. (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.

[8] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[9] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag, 1995.

[10] 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*, 2000.

[11] J.G. Bredeson. Synthesis of multiple input-change hazard-free combinational switching circuits without feedback. *Int. J. Electronics*, 39(6):615–624, 1975.

[12] J.G. Bredeson and P.T. Hulina. Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits. *Information and Control*, 20:114–224, 1972.

[13] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[14] Erik Brunvand, Hans Jacobson, and Ganesh Gopalakrishnan. High-level asynchronous system design using ack. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000.

[15] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[16] F. C. Cheng, S. H. Unger, and M. Theobald. Self-timed carry-lookahead adders. *IEEE Transactions on Computers*, 49(7):659–672, 2000.

[17] F. C. Cheng, S. H. Unger, M. Theobald, and W.-C. Cho. Delay-insensitive carry-lookahead adders. In *Proc. International Conference on VLSI Design*, pages 322–328, 1997.

[18] Chou, Beerel, Ginosar, Kol, Myers, Rotem, Stevens, and Yun. Optimizing average-case delay in the technology mapping of domino dual-rail circuits: A case study of an asynchronous instruction length decoding pla. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1998.

[19] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[20] Bill Coates, Jo Ebergen, Jon Lexau, Scott Fairbanks, Ian Jones, Alex Ridgway, David Harris, and Ivan Sutherland. A counterflow pipeline experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 161–172, April 1999.

[21] J. Cortadella and R. M. Badia. An asynchronous architecture model for behavioral synthesis. In *Proc. European Conference on Design Automation (EDAC)*, pages 307–311. IEEE Computer Society Press, 1992.

[22] J. Cortadella, M. Kishinevsky, S.M. Burns, and K. Stevens. Synthesis of asynchronous control circuits with automatically generated relative timing assumption. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1999.

[23] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete state encoding based on the theory of regions. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[24] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E80-D(3):315–325, March 1997.

[25] O. Coudert. Two-level logic minimization: an overview. *Integration, the VLSI journal*, 17:97–140, 1994.

[26] O. Coudert. Doing two-level logic minimization 100 times faster. In *ACM-SIAM Symposium on Discrete Algorithms*, 1995.

[27] O. Coudert. On solving covering problems. In *Proceedings of the 33rd Design Automation Conference*. ACM, 1996.

[28] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *Proceedings of the 32nd Design Automation Conference*. ACM, 1995.

[29] A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact asynchronous control circuits. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 193–207. Elsevier Science Publishers, 1993.

[30] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 38-Sup23, pages 231–286. Marcel Dekker, Inc., 1998.

[31] Mark Dean, Ted Williams, and David Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In Carlo H. Séquin, editor, *Advanced Research in VLSI*, pages 55–70. MIT Press, 1991.

[32] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. *Logic Synthesis*. McGraw-Hill, 1994.

[33] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski. Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. In *Proc. ACM/IEEE Design Automation Conference*, pages 415–419, 1994.

[34] E.B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM J. Res. Develop.*, 9(2):90–99, 1965.

[35] R.K. Brayton et al. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[36] J. Frackowiak. Methoden der analyse und synthese von hasardarmen schaltnetzen mit minimalen kosten I. *Elektronische Informationsverarbeitung und Kybernetik*, 10(2/3):149–187, 1974.

[37] R.M. Fuhrer, B. Lin, and S.M. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *1995 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1995.

[38] R.M. Fuhrer, S.M. Nowick, M. Theobald, N.K. Jha, and L. Plana. MINIMALIST: An environment for the synthesis and verification of burst-mode asynchronous machines. In *IEEE International Workshop on Logic Synthesis*, 1998.

[39] R.M. Fuhrer, S.M. Nowick, M. Theobald, N.K. Jha, and L. Plana. MINIMALIST: An environment for the synthesis and verification of burst-mode asynchronous machines. Technical Report CUCS-020-99, Columbia University, 1999. Download site is http://www.cs.columbia.edu/async.

[40] Robert M. Fuhrer. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools*. PhD thesis, Columbia University, 1999.

[41] Robert M. Fuhrer and Steven M. Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools.* Kluwer Academic Publishers, 2001.

[42] S. B. Furber, D. A. Edwards, and J. D. Garside. AMULET3: a 100 MIPS asynchronous embedded processor. In *Proc. International Conf. Computer Design (ICCD)*, 2000.

[43] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver. Amulet2e: An asynchronous embedded controller. In *Async97 Symposium.* ACM, April 1997.

[44] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, 1998.

[45] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1), January 1995.

[46] D. A. Huffman. Design of hazard-free switching circuits. *Journal of the ACM*, 4:47–62, January 1957.

[47] Mark B. Josephs, Steven M. Nowick, and C. H. (Kees) van Berkel. Modeling and design of asynchronous circuits. *Proceedings of the IEEE*, 87(2):234–242, February 1999.

[48] J. Kessels and P. Marston. Design asynchronous standby circuits for a low-power pager. In *Async97 Symposium.* ACM, April 1997.

[49] Euiseok Kim, Jeong-Gun Lee, and Dong-Ik Lee. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000.

[50] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.

[51] Tilman Kolks, Steven Vercauteren, and Bill Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996.

[52] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independendent circuits. In *DAC-1994*.

[53] Alex Kondratyev, Michael Kishinevsky, and Alex Yakovlev. Hazard-free implementation of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 17(9):749–771, September 1998.

[54] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proceedings of the 33rd Design Automation Conference*. ACM, 1996.

[55] D.S. Kung. Hazard-non-increasing gate-level optimization algorithms. In *ICCAD-1992*.

[56] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *DAC-91*.

[57] L. Lavagno and S.M. Nowick. Asynchronous control circuits. In S. Hassoun and T. Sasao, editors, *Logic Synthesis and Verification*. Kluwer Academic Publishers, 2001.

[58] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[59] 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*, 2000.

[60] A. Marshall, B. Coates, and P. Siegel. The design of an asynchronous communications chip. *Design and Test*, June 1994.

[61] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The design of an asynchronous microprocessor. In *1989 Caltech Conference on Very Large Scale Integration*, 1989.

[62] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[63] Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.

[64] E.J. McCluskey. *Introduction to the Theory of Switching Circuits*. McGraw-Hill, 1965.

[65] E.J. McCluskey. *Logic Design Principles*. Prentice-Hall, 1986.

[66] Giovanni De Micheli. *Synthesis And Optimization Of Digital Circuits*. McGraw-Hill, 1994.

[67] S. Minato. Zero-Suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th Design Automation Conference*. ACM, 1993.

[68] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), April 1989.

[69] C. Myers and T. Meng. Synthesis of timed asynchronous circuits. In *ICCD-1992*.

[70] Chris Myers and Hans Jacobson. Efficient exact two-level hazard-free logic minimization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 64–73. IEEE Computer Society Press, March 2001.

[71] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: design of a quasi-delay-insensitive microprocessor. *IEEE Design and Test*, 11(2):50–63, Summer 1994.

[72] L.S. Nielsen and J. Sparso. A low-power asynchronous data path for a fir filter bank. In *Proceedings of the International Symposium on Advanced Research*

in Asynchronous Circuits and Systems (Async96), pages 197–207. IEEE Computer Society Press, November 1996.

[73] M. A. Peña and J. Cortadella. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[74] S. M. Nowick. Design of a low-latency asynchronous adder using speculative completion. *IEE Proceedings, Computers and Digital Techniques*, 143(5):301–307, September 1996.

[75] S. M. Nowick and M. Theobald. Synthesis of low-power asynchronous circuits in a specified environment. In *Proceedings of 1997 International Symposium on Low Power Electronics and Design*, pages 92–95, 1997.

[76] S.M. Nowick and B. Coates. UCLOCK: automated design of high-performance unclocked state machines. In *IEEE International Conference on Computer Design*, pages 434–441, October 1994.

[77] S.M. Nowick, M.E. Dean, D.L. Dill, and M. Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 419–427. IEEE Computer Society Press, January 1993.

[78] S.M. Nowick and D.L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *ICCAD-1991*.

[79] S.M. Nowick and D.L. Dill. Synthesis of asynchronous state machines using a local clock. In *IEEE International Conference on Computer Design*, pages 192–197. IEEE Computer Society Press, October 1991.

[80] S.M. Nowick and M. Theobald. Synthesis of low-power asynchronous circuits in a specified environment. Technical Report CUCS-020-97, Columbia University, 1997.

[81] Steven M. Nowick. Automatic synthesis of burst-mode asynchronous controllers. Technical report, Stanford University, 1993. Ph.D. Thesis.

[82] Steven M. Nowick and David L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on CAD*, CAD-14(8):986–997, August 1995.

[83] Steven M. Nowick, Kenneth Y. Yun, and Peter A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 210–223. IEEE Computer Society Press, April 1997.

[84] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low-noise configurable self-timed DSP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 32–42, 1998.

[85] Ad M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.

[86] Luis Angel Plana. *Contributions to the Design of Asynchronous Macromodular Systems*. PhD thesis, Department of Computer Science, Columbia University, 1998. Technical report CUCS-001-99.

[87] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yu n, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: an asynchronous instruction-length decoder. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70. IEEE Computer Society Press, April 1999.

[88] R. Rudell and A. Sangiovanni Vincentelli. Multiple valued minimization for PLA optimization. *IEEE Transactions on CAD*, CAD-6(5):727–750, September 1987.

[89] Richard Rudell. Logic synthesis for VLSI design. Technical Report UCB/ERL M89/49, Berkeley, 1989.

[90] J.W.J.M. Rutten and M.R.C.M. Berkelaar. Improved state assignments for burst mode finite state machines. In *Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997.

[91] J.W.J.M. Rutten, M.R.C.M. Berkelaar, C.A.J. van Eijk, and M.A.J. Kolsteren. An efficient divide and conquer algorithm for exact hazard free logic minimization. In *Proc. Design, Automation and Test in Europe (DATE)*. IEEE Computer Society Press, February 1998.

[92] J.W.J.M. Rutten and M.A.J. Kolsteren. A divide and conquer strategy for hazard free 2-level logic synthesis. In *International Workshop on Logic Synthesis*, 1997.

[93] T. Sasao. An application of multiple-valued logic to a design of programmable logic arrays. In *Proceedings of Int. Symposium on Multiple-Valued Logic*, 1978.

[94] Montek Singh, Jose Tierno, Alexander Rylyakov, Sergey Rylov, and Steven Nowick. An adaptively-pipelined mixed synchronous-asynchronous digital fir filter chip operating at 1.3 gigahertz. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2002.

[95] R.F. Sproull, I.E. Sutherland, and C.E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, 1994.

[96] Ken Stevens, Ran Ginosar, and Shai Rotem. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, April 1999.

[97] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[98] M. Theobald and S. M. Nowick. Transformations for the synthesis of asynchronous distributed control. In *IEEE International Workshop on Logic Synthesis*, 2000.

[99] M. Theobald and S. M. Nowick. Transformations for the synthesis and optimization of asynchronous distributed control. In *Proc. ACM/IEEE Design Automation Conference*, June 2001.

[100] M. Theobald and S.M. Nowick. An implicit method for hazard-free two-level logic minimization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1998.

[101] M. Theobald, S.M. Nowick, and T. Wu. Espresso-HF: A heuristic hazard-free minimizer for two-level logic. In *Proceedings of the 33rd Design Automation Conference*. ACM, 1996.

[102] Michael Theobald and Steven M. Nowick. Fast heuristic and exact algorithms for two-level hazard-free logic minimization. *IEEE Transactions on Computer-Aided Design*, November 1998.

[103] S.H. Unger. *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, 1969.

[104] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design and Test of Computers*, 11(2):22–32, Summer 1994.

[105] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.

[106] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.

[107] Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.

[108] K. Yun and D.L. Dill. A high-performance asynchronous SCSI controller. In *IEEE International Conference on Computer Design*. IEEE Computer Society Press, October 1995.

[109] Kenneth Y. Yun, Ayoob E. Dooply, Julio Arceo, Peter A. Beerel, and Vida Vakilotojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[110] K.Y. Yun and D.L. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1992.

[111] K.Y. Yun, D.L. Dill, and S.M. Nowick. Synthesis of 3D asynchronous state machines. In *IEEE International Conference on Computer Design*, pages 346–350. IEEE Computer Society Press, October 1992.