Synthesis of asynchronous burst-mode finite state machines

Synthesis of Asynchronous Burst–Mode Finite State Machines

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 19 april 2000 om 16.00 uur

door

Josephus Waltherus Johannes Maria Rutten

geboren te Budel

Dit proefschrift is goedgekeurd door de promotoren:

prof.Dr.–Ing. J.A.G. Jess

en

prof. S.M. Nowick

Copromotor:
dr.ir. M.R.C.M. Berkelaar

Druk: Universiteitsdrukkerij Technische Universiteit Eindhoven

CIP–DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Rutten, Josephus W.J.M.

Synthesis of asynchronous burst–mode finite state machines / by Josephus W.J.M. Rutten. – Eindhoven : Technische Universiteit Eindhoven, 2000.
Proefschrift. – ISBN 90–386–1730–5
NUGI 832
Trefw.: asynchrone logische schakelingen / grote geintegreerde schakelingen ; CAD / logische schakelingen ; CAD / logische geintegreerde schakelingen.
Subject headings: asynchronous circuits / logic CAD / digital integrated circuits / integrated logic circuits.

Summary

In this thesis we discuss the synthesis of asynchronous burst–mode machines (BMMs). These machines are a constrained form of asynchronous Huffman machines. Because of this, they can be synthesized fully automatically and they allow for a modular design. Compared to competing design methods, the BMM–based design method has the important property that the existence of an implementation is guaranteed.

BMMs are synthesized with the same synthesis steps that can be found in the synthesis of synchronous finite state machines. These steps are:

- State reduction
- State assignment
- Logic synthesis
- Technology mapping

All these steps, however, need to be modified in order to guarantee that the implementation is free from hazards and critical races. In this thesis, each step is described in detail. The contributions of this thesis can mainly be found in the steps state assignment and logic synthesis.

In the state assignment step, we improve the symbolic minimization method proposed by Fuhrer. We show that the encoding constraints introduced by Fuhrer can be relaxed. Furthermore, we propose to constrain the allowed intersections between so–called symbolic implicants. By doing so, we can obtain a state encoding that results in smaller implementations of BMMs.

Our contributions to logic synthesis are twofold. In this thesis we propose a new algorithm, called the threeway method. We show that the threeway method, and a heuristic method based on the threeway method are capable of dramatically reducing the runtimes necessary to find an efficient PLA implementation of a BMM. In this thesis we also show that the initial PLA implementation can be converted to an efficient multi–level implementation, where we use a set of so–called hazard–non–increasing transformations and a set of local don't cares.

In this thesis, we also show that we are able to verify that the combinational logic of a BMM does function correctly without hazards. This is done with the help of a 5–valued algebra, which is based on the work by Kung. Verification takes place by simulating each transition. We show that with one minor modification the 5–valued algebra can also verify that each transition is implemented such that the so–called Burst–Mode condition is met.

Samenvatting

In dit proefschrift beschrijven we het implementatietraject van asynchrone burst–mode machines (BMMs). Deze machines zijn asynchrone Huffman machines die aan een aantal extra voorwaarden dienen te voldoen. Hierdoor kunnen ze automatisch door een computer worden geïmplementeerd en staan ze toe dat de ontwerper zijn of haar ontwerp modulair opdeelt. Vergeleken met concurrerende implementatiemethoden, heeft de methode gebaseerd op BMMs de belangrijke eigenschap dat het bestaan van een implementatie gegarandeerd kan worden.

BMMs kunnen worden geïmplementeerd gebruikmakend van dezelfde stappen die al bekend zijn voor het implementeren van synchrone machines. Deze stappen zijn:

- Toestandsreductie
- Toestandscodering
- Logische synthese
- Afbeelding op een gegeven technologie

Al deze stappen moeten echter wel aangepast worden om te kunnen garanderen dat de implementatie vrij is van hazards en critical races. In dit proefschrift wordt elke stap in detail beschreven. De bijdragen van dit proefschrift zijn vooral te vinden in de stappen toestandscodering en logische synthese.

Onze bijdrage in de toestandscoderingsstap is een verbetering van de symbolische minimalisatie methode voorgesteld door Fuhrer. We laten zien dat de coderingsvoorwaarden, geïntroduceerd door Fuhrer, kunnen worden afgezwakt. Verder stellen we voor om de toe te laten doorsnijdingen tussen zo–genoemde symbolische implicanten te beperken. Hierdoor kunnen we een toestandscodering vinden die resulteert in kleinere implementaties van BMMs.

We hebben twee bijdragen geleverd op het gebied van de logische synthese. In dit proefschrift introduceren we een nieuw algoritme genaamd de threeway–methode. We laten zien dat de threeway–methode en een heuristische methode gebaseerd op deze threeway–methode in staat zijn om de rekentijd, nodig om een efficiënte PLA–implementatie van een BMM te vinden, drastisch te reduceren. We laten verder zien dat de PLA–implementatie kan worden geconverteerd naar een efficiënte multi–level beschrijving, waarbij we gebruik maken van transformaties die hazard–non–increasing zijn en van lokale don't cares.

We laten in dit proefschrift ook zien dat we in staat zijn om te verifiëren dat het combinatorische gedeelte van een BMM correct functioneert zonder hazards. Hierbij maken we gebruik van een 5–waardige algebra, welke is

gebaseerd op het werk van Kung. Het verificatieproces vindt plaats door elke transitie te simuleren. We laten zien dat met een kleine wijziging de 5–waardige algebra ook gebruikt kan worden om te controleren dat elke transitie wordt geïmplementeerd zodanig dat voldaan wordt aan de zogenaamde Burst–Mode conditie.

Contents

Preface

The thesis you are reading now, and which you will hopefully continue to read, represents my humble contributions to the exciting world of asynchronous design. My little trip in that world started back in 1994 and without the guidance and good advice of many people I probably would have stranded somewhere.

I had the opportunity to work in a group with a collaborative spirit, where everybody was interested in each other's work. Over the years I received guidance and good advice from many people in our group. I am especially grateful to Michel Berkelaar, my co–promotor, for his patience and his constructive criticism. I am also grateful to Koen van Eijk for the many discussions we had. I consider him as my second co–promotor.

Because you can't work all the time, not even at the university, life would not be fun without a room mate to share your mental state with (with or without their approval). I would therefore like to thank Hans Fleurkens and Etienne Jacobs for enduring me. I would also like to thank Etienne for being the victim who read the first version of my thesis.

I would like to thank all members of the reading committee for their valuable comments with respect to this thesis. I would also like to thank Professor Jess for giving me the opportunity to perform my Ph.D. in his group, of which I am now a member. Last, but not least, I would like to thank my family and my friends for their support.

Chapter 1 Introduction

1.1 Introduction

The importance of automated design is well understood in the EDA (Electronic Design Automation) community. With current chip designs exceeding ten million transistors [Sema97], a hand–crafted design is only feasible for very small parts of digital integrated circuits with very high volumes, like the popular x86–microprocessors.

The majority of all digital systems designed today are synchronous systems. The main reason for the popularity of synchronous systems is that major parts of a synchronous system can be designed automatically. This is because clock–based designs are modular. A design can therefore be partitioned into smaller blocks, each of which can be designed independently thus reducing the complexity of the design process. Today, designers can specify a synchronous system in a high–level hardware description language. Commercial tools together form a complete design flow that transforms the description eventually into layout. Asynchronous design styles have been lagging behind in this respect. Asynchronous designs often still are handcrafted designs.

In the last decade, there has been a renewed interest in the development of asynchronous design methods. Initially this renewed interest was a reaction to the belief in that time that synchronous design styles were doomed. In future technologies, delay models would break down. Wire delays would become dominant compared to gate delays. People would have problems controlling the clock skew. Asynchronous design styles, with their lack of any clock, and their tolerance to timing deviations seemed a promising alternative.

Today, most digital systems are still designed as synchronous systems. New solutions have emerged that solved most of the problems, at least for the time being. This fact, however, does not nullify the possible advantages of having an asynchronous design style. In the last decade synthesis systems have been developed that show that asynchronous designs, just like synchronous designs, can be designed in a modular fashion.

Furthermore, asynchronous designs can be faster than synchronous designs. In synchronous designs, the clock must be adjusted to the slowest element in the design, whereas in asynchronous designs, elements can operate at their own native speed. Until now, the promise of increased performance has only been shown for very specific hand–crafted circuits, like the self–timed divider of Williams [Will91].

The last possible advantage that we will mention is the application of asynchronous circuits in the area of low–power circuits, especially in static–CMOS design. In synchronous circuits, the latches needed consume power every clock cycle, even when the circuit is in idle mode, i.e. when it is doing nothing useful. In asynchronous designs, circuits in idle mode consume almost nothing at all. It has been demonstrated that asynchronous circuits are indeed well–fit for applications that must have long stand–by times, for example hand–held applications [Kess97]. However, synchronous design methodologies are also being improved for low–power applications with, for example, clock–gating techniques [Thee96], although it seems that asynchronous designs have the leading edge, for the time being.

1.2 Asynchronous circuit synthesis

This thesis is about the synthesis of asynchronous circuits. In the synthesis of synchronous circuits, three important problems are addressed, namely:

- Avoiding hazards
- Avoiding critical races
- Modularity problems

Hazards, or glitches, are spurious changes in output signals. In synchronous systems, hazards are removed by making sure that all logic is stable at the time of clocking. This is guaranteed by choosing a cycle time that is long enough.

Critical races can occur when two or more variables, usually state variables, change concurrently. In practice, two or more variables will never change exactly simultaneously. So, a circuit might observe a certain order in which the variables change. If the behavior of the circuit depends on the order in which it observes these changes, a critical race is said to exist within the circuit. This is not acceptable. In synchronous systems, critical races are avoided by explicitly synchronizing concurrently changing signals. The flipflops/latches make sure that concurrent changes are perceived by the circuit as occurring simultaneously.

Modularity is the capability to partition a design into parts that can be designed separately. Modularity is related to synchronization: in a modular

design, all parts that communicate with each other have some sort of synchronization scheme. Problems with these synchronization schemes are related to critical races, since these problems emerge at the interface between the different parts. At the interface, we can also have multiple signals that change concurrently. Again, the behavior of the implementation of a part might be influenced by the order in which it observes signal changes. In a synchronous system this problem does not occur, again because of the explicit synchronization of concurrently changing signals.

Without a clear solution for each of the above mentioned problems, an asynchronous system cannot function correctly. In the past, asynchronous circuits were therefore hand–crafted. This meant that designing an asynchronous system took a lot of time. For high–performance circuits such as microprocessors, this can be tolerated, and asynchronous processors have indeed been designed, mostly by hand [Furb97].

Due to the renewed interest in asynchronous design methods, a number of asynchronous design systems have been developed, that address the three problems mentioned.

In the area of high–level asynchronous system synthesis, several approaches have been proposed [Akel92], [Ber92a]. One of the most successful approaches in this category is the DICY–system, developed by Philips [Ber92a]. In the DICY–system, designers can specify their circuit in the TANGRAM language, a language similar to the CSP (Communication Sequential Processes) language. The specification is then mapped onto a set of pre–designed asynchronous building blocks that communicate with each other by using a handshake protocol. The DICY–system is a complete design environment, in which also testability issues have been addressed. So, a designer can design a completely asynchronous circuit without needing a detailed knowledge about asynchronous circuit design. The DICY–system has been applied successfully to design a number of chips, ranging from DCC error–correctors [Berk94] to asynchronous 8051–processors [Gage98]. The resulting circuits are better in terms of power compared to their synchronous counterparts.

In the DICY–system, the aspect of modularity has been well addressed. Asynchronous building blocks are synchronized by the introduction of handshaking. The DICY–system uses a fixed number of building blocks which are hand–designed to ensure a hazard–free and critical–race–free behavior. This way, the first two problems, avoiding hazards and critical races, are solved. However, more efficient circuits might be obtained if special building blocks, designed for a specific functionality, would be available. Since almost all building blocks are designed by hand, designing a special

block will take a lot of time, and one cannot pass this responsibility to the designer. This means that the size and complexity of the building blocks is limited, except for special regular building blocks. The DICY–system could possibly profit from asynchronous design styles that focus on an automatic solution of the first two problems.

In the next three sections, we look at two popular approaches that focus on the automatic generation of hazard–free and critical–race–free asynchronous circuits. The two approaches are STG–based (Signal Transition Graph) synthesis and asynchronous FSM–based (Finite State Machine) synthesis. We focus on these two approaches because they are more or less fully automatic. There are other approaches that are based on using transformations to derive circuits. These approaches are in general not fully automatic. Martin [Mart90], for example, introduces the notion of production rules to guide an experienced designer in a hand–based design. In the discussion in the next three sections, we assume that the reader is familiar with most basic concepts in digital and asynchronous design. Many of these concepts are introduced in chapter 2.

1.3 STGs

Signal Transition Graphs (STGs) are interpreted *Petri–nets*. Petri–nets are bi–partite directed graphs. Bi–partite graphs consist of two types of nodes. Nodes of the same type are not connected by edges. In Petri–nets, the two types of nodes are called *places* and *transitions*. An example is given in figure 1.1.

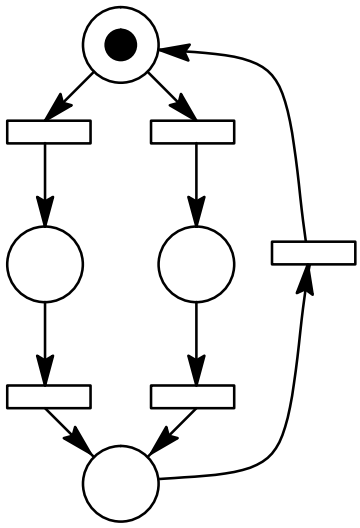


FIGURE 1.1. Example of a Petri–net

Places are represented by circles. Each place can contain one or more *tokens*. These tokens are visible in figure 1.1 as black dots (only one token is visible). A possible configuration of tokens over all places is called a *marking*. Each Petri–net is initialized with an initial marking. Transitions are represented by boxes. Petri–nets can be used to model, analyze, and simulate processes. Simulation is performed by the so–called *token–game*. The token–game describes how a Petri–net is simulated, or *executed*. In a Petri–net, each transition can *fire* if each fanin place contains at least one token. In that case, the transition is said to be *enabled*. If a transition fires, a token is removed from each fanin place, and a new token is put in each fanout place. The two top transitions in figure 1.1 are both enabled. Only one of these transitions can fire. By doing so, it will disable the other transition. The Petri–net is free to choose which of these transitions will fire. If a transition is enabled, and it is not disabled by the firing of another transition, then it will eventually fire.

In an STG, each transition is labeled with a signal transition. Usually, places are not shown visibly in STGs. This is because places in STGs usually have one fanin and one fanout transition. An initial token configuration is represented by black dots put on the appropriate edges. An example of an STG is shown in figure 1.2. In figure 1.2, $\mathrm{Re}_{\mathrm{in}}$ and $\mathrm{Ack}_{\mathrm{in}}$ are input signals; and $\mathrm{Re}_{\mathrm{out}}$ and $\mathrm{Ack}_{\mathrm{out}}$ are output signals. Sometimes places *are* shown to express choice. An example of this is given in figure 1.3a.

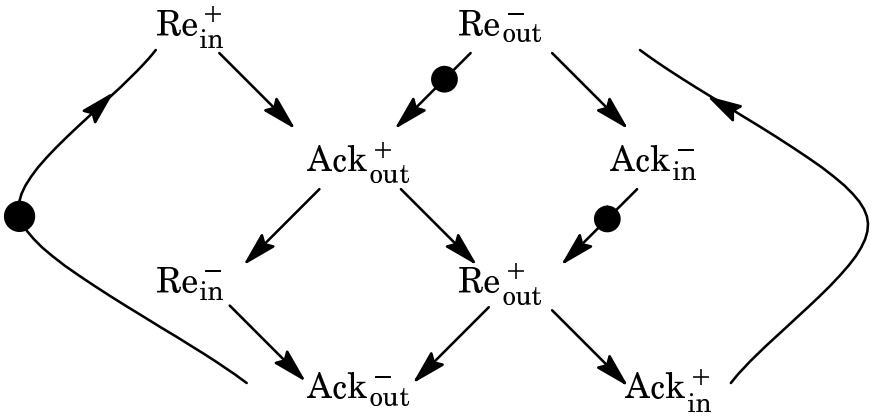


FIGURE 1.2. Example of an STG

In figure 1.3a the STG can choose between two transitions a+ and b+. Figure 1.3a is an example of a *free–choice STG*. A free–choice STG is an STG in which each place, that has more than one fanout transition, is the only fanin place of these fanout transitions. An example of a non free–choice STG is shown in figure 1.3b. Most STG–based synthesis methods assume that the specification is a free–choice STG.

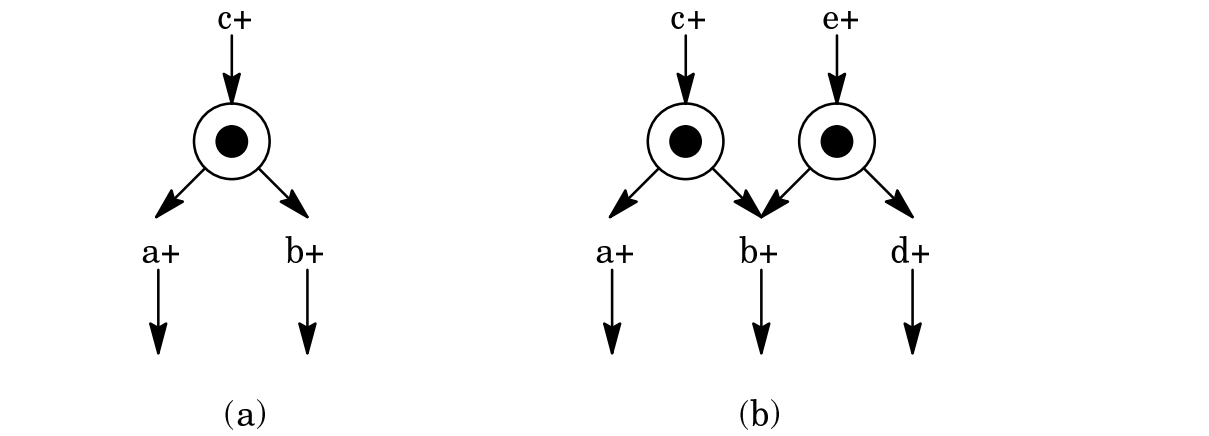


FIGURE 1.3. Choice in STGs

An STG without any restriction whatsoever can specify exotic behavior that cannot be implemented by (hazard–free) hardware. All STG–based synthesis methods therefore impose constraints on the STG specification. The constraints that are required in most, if not all methods, are: 1–boundedness, also called safeness, and liveness. The 1–boundedness constraint guarantees that the number of tokens in each place does not exceed one, for all the possible markings reachable from the initial marking. Liveness guarantees that an STG is safe, that rising and falling signal changes alternate, and that a sequence of transitions exists, such that each transition in the STG can be enabled from each reachable marking.

In many STG–based synthesis methods, an STG is first converted to its *state graph* representation. A state graph is a directed graph, where vertices correspond with states. Each state corresponds with a reachable marking of the STG. A state is connected by a directed edge to another state if the marking of the second state is reachable from the marking of the first state by changing one variable in value. The state graph of the STG in figure 1.2 is represented in figure 1.4.

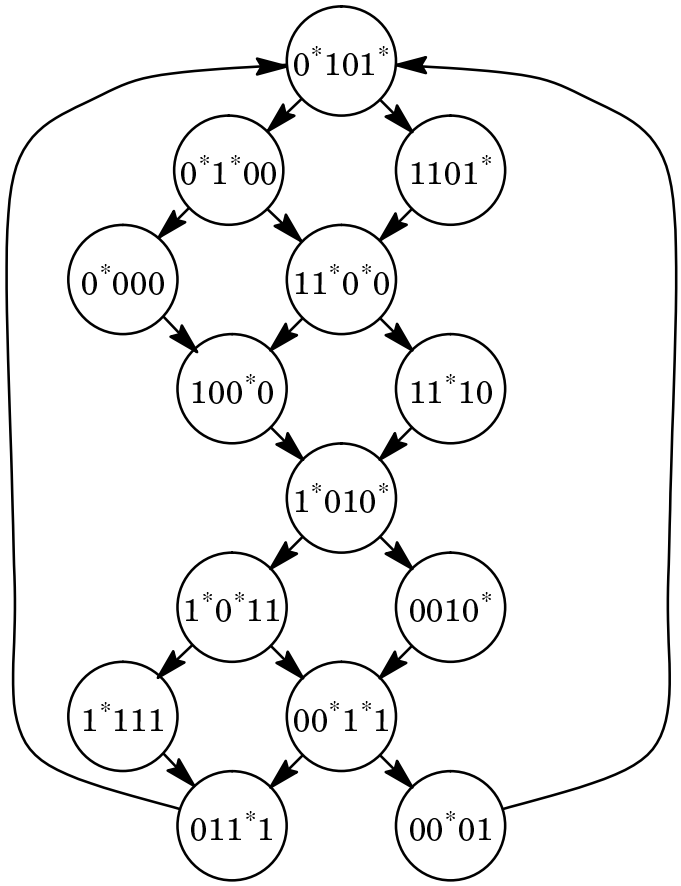


FIGURE 1.4. State graph representation of the STG in figure 1.2

Each state is labeled with the values of all signals that occur in the STG. In the state graph in figure 1.4 the order of the signals is: Re_{in}, Ack_{in}, Ack_{out} and Re_{out}. A star denotes that the corresponding signal is excited in that state, meaning that it might change.

In the next section we will discuss some of the STG–based synthesis approaches that have been proposed and implemented in the last few years. This survey is not complete. We refer the interested reader to the more thorough discussions provided in the excellent overview papers [Hauc95], [Yako96], [Davi97].

1.4 STG–based synthesis approaches

1.4.1 Introduction

In this section we will look at a number of STG–based synthesis methods. These methods can, roughly, be divided into *speed–independent* and non

speed–independent approaches. Speed–independent methods assume an unbounded gate delay model, where wire delays are neglected.

1.4.2 Non speed–independent approaches

Chu [Chu87], and later Meng [Meng91] consider STGs that satisfy persistency constraints. Persistency constraints require that a signal, that is enabled in an STG, cannot be disabled by the firing of another signal. According to Chu, these constraints are necessary and sufficient, and guarantee a correct hazard–free implementation of an asynchronous circuit. However, as is noted by [Moon92], persistency constraints are only capable of avoiding *function hazards*. It is impossible to guarantee the absence of *logic hazards* by evaluating a state graph, or an STG specification. A discussion about the possible forms of hazards is provided in section 2.7.

[Lava92] proved that in general the persistency constraints formulated by Chu are not even necessary. He showed that it is necessary for an STG to obey the Complete State–Coding property (CSC property). This property states that two or more states, in the state graph representation of an STG, may have the same label, if the set of enabled signals that are different between two of such states consists entirely of input signals. The CSC constraint is necessary to avoid all function hazards. The CSC constraint, however, is also not sufficient in order to suppress logic hazards, since the possible occurrence of these hazards is determined by the actual implementation.

Moon [Moon92] targets a two–level sum of cubes implementation (explained in section 2.2) based on an unbounded gate delay model. Moon also assumes that the environment is well–behaved, meaning that it obeys the transition relations specified by the STG, and that it only applies new input changes when all logic, having these inputs in their fanin set, has stabilized. Moon introduces transition cubes (A cube is basically an AND–gate, for a better definition see chapter 2) to avoid static 1–hazards. He also tries to remove all dynamic hazards by applying factorization. Moon's approach, however, cannot guarantee a circuit implementation for every STG specification that obeys the above mentioned constraints. Furthermore, he requires the environment to be well–behaved, which imposes timing constraints on the environment. These timing constraints may be hard to satisfy, since the methodology does not incorporate any timing information.

An interesting approach was proposed by Lavagno [Lava92]. Lavagno initially targets a two–level sum of products implementation with an unbounded wire delay model. He shows that he can avoid static hazards resulting from concurrent (input) transitions by introducing a set of transition cubes that cover all the possible combinations of concurrent

transitions. These cubes should all be covered by the cubes in the solution. This notion is similar to the concept of *required cubes* introduced in chapter 4. The transition cubes avoid possible static hazards as long as transitions occur within one set of concurrent changes. However, as different sets of concurrent transitions are entered, some transition cubes will turn off, while other transition cubes turn on. Some of the transition cubes being off must turn on before the transition cubes supposed to turn off do so. A failure to meet this constraint will result in a so–called *essential hazard* [Ung69]. Essential hazards can only be avoided by the introduction of at least one delay element. According to Lavagno, these hazards occur if a circuit, implementing the behavior of an output/state signal, observes a sequence of transitions out of order from the intended sequence as specified by the STG. His idea is to add delay elements such that he can guarantee that the circuit is forced to observe the intended sequence of transitions.

It is not clear how Lavagno deals with conflicting constraints/requirements. One constraint might require that a certain transition is delayed with respect to another transition in order to force the circuit to observe the second transition before the first transition. Another constraint might specify the opposite. It is not possible to solve both constraints by just adding delay elements. By introducing multiple copies of transitions cubes with different delay paddings one might be able to solve both constraints simultaneously. It is necessary to resolve conflicting constraints, one way or another, in order to avoid static hazards.

Lavagno also claims that his approach removes all dynamic hazards as well. He states that dynamic 1–to–0 hazards will not occur since "*We are sure that every onset cube that we can enter has time to be turned on before we proceed.*" However, not all dynamic hazards are caused by the circuit observing a certain intended signal sequence out of order. Therefore, Lavagno's approach is not capable of detecting all dynamic hazards. It also seems that his method is computationally expensive. His delay padding algorithm is based on a linear programming method. However, even for small circuits many constraints can be generated resulting in large runtimes.

1.4.3 Speed–independent approaches

The most popular approaches to synthesize STGs in use today are completely speed–independent approaches. In order to have truly speed–independent circuits, the switching activity of each gate should be acknowledged by at least one other gate. Many papers have been devoted to synthesis methods based on speed–independent approaches.

Most speed–independent approaches assume an architecture based on a standard Mueller C–implementation [Beer92], [Kond94]. The general architecture, called a *signal network*, is depicted in figure 1.5.

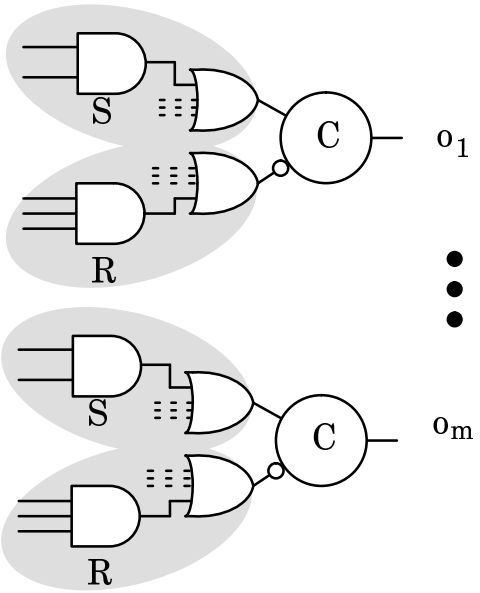


FIGURE 1.5. General signal network to implement speed–independent STGs

Each of the networks in figure 1.5 is used to implement the behavior of an output signal o_k. Each network contains a so–called Mueller C–element (or C–element).

A C–element, like a Set–Reset flipflop, is a memory element. We assume that initially the output of the C–element is equal to 0. A C–element will turn on when both inputs are equal to 1. A C–element will turn off when both inputs return to 0. For the remaining input combinations the C–element will retain its state. This behavior can be described by the Boolean formula: $C := (a + b)C + ab$. A *time diagram* of the typical behavior of a C–element is given in figure 1.6.

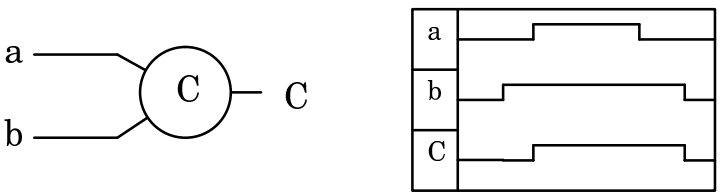


FIGURE 1.6. Typical behavior of a C–element

With the help of the two–level network denoted by S, an output, o_k, is set to 1. The two–level network denoted by R is used to reset output o_k, i.e. set it to 0. To implement the STG in figure 1.2, we need two of the signal networks in figure 1.5. The C–element is used to maintain the value of output o_k. In order to change output o_k from 0 to 1, an AND–gate belonging to the S–network should turn on. In order to guarantee a correct speed–independent behavior, only one of all the AND–gates is allowed to turn on. The change of this AND–gate is acknowledged by the C–element by changing the value of output o_k from 0 to 1. This is also the reason why only one AND–gate is allowed to turn on: The C–element can acknowledge only one AND–gate of the S–network. To change the output from 1 to 0, only one AND–gate belonging to the R–network is allowed to turn on, again because the C–element can only acknowledge one AND–gate of the R–network. The AND–gate belonging to the S–network that caused the 0–to–1 change of the output must also turn off before the output can make the change from 1 to 0. The C–element can thus be used to acknowledge the changes of two AND–gates simultaneously: one belonging to the S–network, and one belonging to the R–network.

In order to guarantee that an STG can be mapped onto a signal network, it should at least obey the earlier mentioned constraints of liveness, safeness, and the STG should also obey the CSC–property. All these constraints are necessary, but they are in general not sufficient.

To generate a circuit implementing the behavior specified by an STG, most speed–independent methods transform the STG into a state graph. As was said before, the nodes of a state graph are formed by the set of reachable markings of the STG specification. An implementation of the STG is derived by extracting, for each signal that has to be implemented, the so–called *excitation* and *quiescent* regions of that signal: states where the output is excited or remains stable, respectively. For each excitation region, an AND–gate is introduced. The quiescent regions can be used as don't cares. Application of this method to the STG in figure 1.2 results in the implementation in figure 1.7.

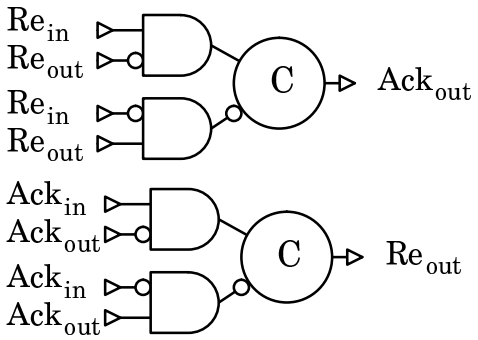


FIGURE 1.7. Implementation of the STG in figure 1.2

Although synthesis approaches targeting signal networks are attractive in their ingenuity, and simplicity, there are some major problems.

The first problem is that these approaches assume that an AND–gate can invert a set of inputs for free. That is, they assume that the contribution to the delay of the inverters, required to invert the inputs, is negligible. This assumption is clearly invalid. To avoid this problem, S/R–flipflops can be used instead of C–elements. S/R–flipflops have inverted outputs for free, that is, the assumption is made that both phases of an output are generated at the same time. However, this does not solve the cases where we need inverted primary inputs. Special circuitry is required to provide the circuit with both phases of these primary inputs. Furthermore the use of S/R–flipflops might introduce critical–race problems in case the set and reset inputs of the S/R–flipflop switch nearly simultaneously.

The second problem is the assumption of the delay model itself. In most speed–independent methods the delay model does assume wire delays to be negligible. Although this assumption is still valid for the blocks that can be handled by synchronous logic synthesis tools today [Sylv98], neglecting wire delays in asynchronous circuits can lead to a hazardous behavior. Even if wire delays can be neglected compared to gate delays, then we still need the notion of wire delay to model the fact that gates can also have input delays. Some input changes might be observed by the gate before other input changes, due to e.g. variations in threshold voltages. These delay differences may not be negligible [Ber92b].

The third problem is the specification of the STG itself. In order to guarantee that in each S or R–network in the general signal network implementation only one AND–gate can, and will, turn on, the STG has to obey some additional constraints. Kondratyev et al [Kond94] prove that an STG not obeying these additional constraints can be transformed into one that does,

by inserting extra signals. By doing so, the extra signals might have to be made observable to the environment. This means a modification of the environment, which is not always acceptable.

The last problem we mention is technology mapping. The number of inputs that can be connected to a complex gate is bounded. For larger number of inputs, the circuit must be decomposed to be implementable. Since the speed–independent behavior must be preserved, common technology–mapping algorithms cannot be used. In [Cort97], special decomposition and technology–mapping algorithms are described. However, the experiments in this paper show that the algorithms can fail for even relative small circuits. Furthermore, these experiments also show that the mapped circuit can become quite large in terms of area.

Patches do exist for all of the above mentioned problems. In case of timing problems, one could introduce (inertial) delay elements, or special hazard–filtering flipflops [Sawa95]. These flipflops might degrade the circuit in terms of area and speed. In case the methods fail, the designer can try to adapt his specification manually. This is not always acceptable. One can try to formulate new additional constraints to be satisfied by each STG specification. However, by introducing new constraints, the specification process might become troublesome for a designer. With so many constraints necessary, it might be better to adopt a more inherently constrained model instead of the STG–model. In the next section, we describe such a model.

1.5 Synchronous and Asynchronous Finite State Machines

Finite State Machines (FSMs) are digital circuits with memory. Synchronous FSMs have explicit memory elements, flipflops or latches that are clocked each clock cycle. In each clock cycle, the machine can change its state. In this thesis, we consider asynchronous FSMs that are not clocked. Instead of memory elements, delay elements are used. These delay elements are necessary to avoid the existence of essential hazards. As was said before, these hazards can only be avoided by inserting at least one delay element in the circuit. The above discussed architectures for both synchronous and asynchronous finite state machines are shown in figure 1.8.

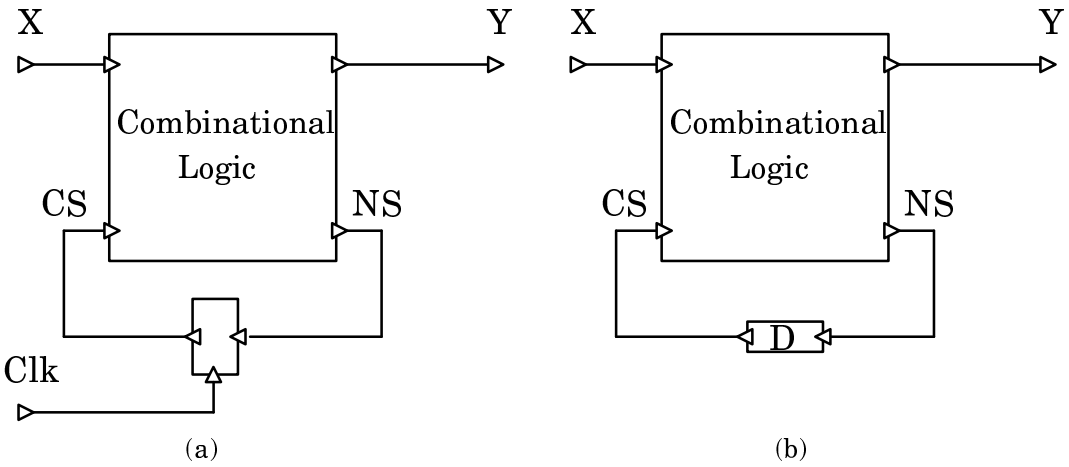
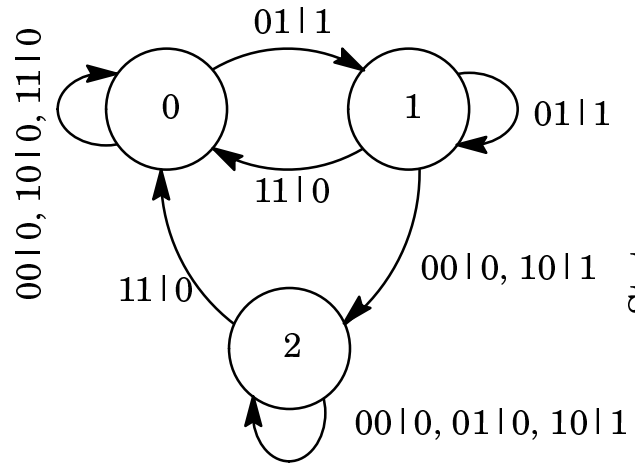


FIGURE 1.8. Synchronous (a) and asynchronous (b) FSM architectures

In figure 1.8, X represents the set of primary inputs, Y represents the set of primary outputs, CS represents the set of current state variables and NS represents the set of next state variables.

Both synchronous and asynchronous FSMs can be specified by using a state–transition graph, or a flow table. Examples of both a state–transition graph and a flow table are given in figure 1.9.



State \ Input	00	01	10	11
0	0,0	1,1	0,0	0,0
1	2,0	1,1	2,1	0,0
2	2,0	2,0	2,1	0,0

(b)

FIGURE 1.9. Example of a state–transition graph and a flow table

In figure 1.9a, a state–transition graph is depicted. A state–transition graph is a directed graph, where the nodes correspond one to one with a state, and

the edges between nodes represent the next–state and output functions. Each edge represents a possible transition between two states. Each edge is annotated with one or more conditions, separated by commas. Each condition consists of an input part followed by an output part. These two parts are clearly distinguished by the symbol '|'. The input part of a condition specifies the input vector necessary to enable the state transition represented by the edge. If a certain edge is 'taken', the outputs of the machine are set to the output part of the condition that is valid.

In figure 1.9b, a flow table representation of the state graph in figure 1.9a is depicted. The rows in a flow table correspond with the states of the FSM. The columns correspond with all possible input vectors. The entries in the flow table specify the next state and output of the machine as a function of the current state (the row) and the input vector (the column). Empty entries correspond with don't cares. The example in figure 1.9 has one output. This example has no entries with don't cares.

In synchronous machines, the logic block in figure 1.8 will calculate, in each clock cycle, the next state and the new values assumed by the outputs. In asynchronous machines, however, there is no clock to tell the machine when to evaluate its inputs and current state. Because of the unlatched feedback, asynchronous machines can be considered as machines that continuously calculate a new state. However, in order to be able to apply new inputs, which can consist of multiple changes of input variables, the circuit has to know which input changes belong together, i.e. it has to know the set of input changes that together produce the new next state and output of the machine. In asynchronous finite state machines, known as *Huffman machines*, this problem is solved by introducing two delay values δ_1 and δ_2, where $\delta_1 > \delta_2$. Input changes that happen within a period of duration δ_2 are assumed to have happened simultaneously. Input changes that are at least δ_1 apart are assumed to be input changes belonging to two separate inputs to the machine. An example is shown in figure 1.10.

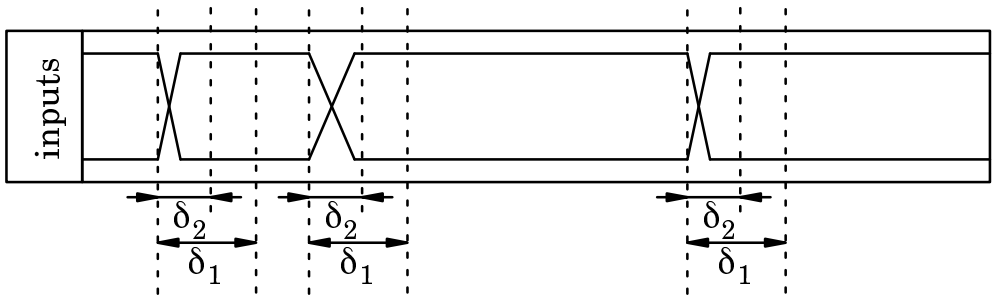


FIGURE 1.10. Timing in a Huffman machine

In figure 1.10, a typical behavior in time of a Huffman machine can be observed. At certain points in time, some input variables change in value. Changing inputs are represented by two crossing lines. After the first observed change of an input variable, interval δ_2 is started. Changes of input variables belonging together should occur within this interval. Input changes outside δ_2 are not interpreted as belonging together. Usually, it is not allowed for input changes to occur after interval δ_2. After the input changes have been observed, the machine needs some time, represented by δ_1 to calculate the next state and the new values of the outputs. As δ_1 expires, the environment is allowed to again change the values of some input variables.

Value δ_1 is determined by the time it takes for the asynchronous machine to become stable again. To estimate the value of δ_1, we must assume a certain delay model. In the synthesis of Huffman machines, it is common to choose a bounded gate delay model. The delay of each gate is bounded by a certain maximum. An asynchronous circuit expecting the environment to wait until it has stabilized, like the Huffman machine, is said to operate in *fundamental mode*.

1.6 Asynchronous FSM–based synthesis approaches

The synthesis of both synchronous and asynchronous FSMs can be partitioned into the following steps:

- State reduction
- State assignment
- Logic synthesis
- Technology mapping

In the first step, the number of states of a machine is reduced. In the second step, the symbolic states are assigned a binary encoding. During the third step, the logic description of the combinational network part in figure 1.8 is reduced. Finally, during the last step, the machine is mapped onto silicon using a specific library in a given technology.

By applying the classical synthesis of asynchronous Huffman machines, as is described in e.g. [Ung69], the specification in figure 1.9, is converted to the circuit depicted in figure 1.11.

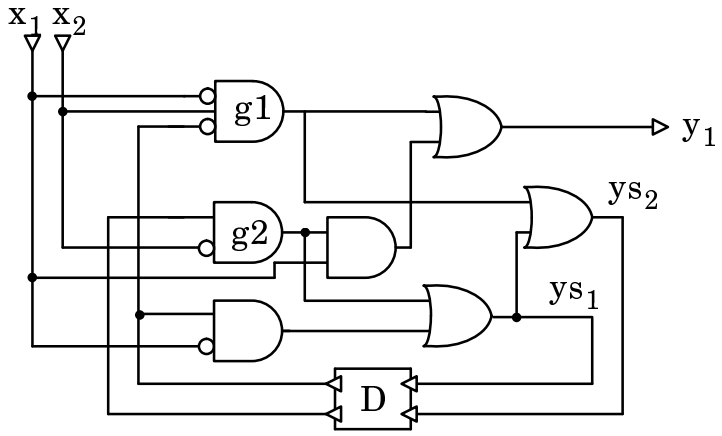


FIGURE 1.11. Huffman implementation of the specification in figure 1.9

In this circuit, the state encoding displayed in table 1.1 was used. Let us analyze the behavior of the circuit for two situations.

TABLE 1.1 State encoding of the flow table in figure 1.9b

State	Encoding (ys_1 ys_2)
0	00
1	01
2	11

Suppose the machine is in state 0 with both inputs equal to 0. Now suppose the environment changes both inputs to 1. According to the flow table in figure 1.9b, the machine is supposed to remain in state 0. However, if we examine the circuit in figure 1.11, we can observe that gate g1 can temporarily switch on if this gate observes the change of input x_2 first, causing a glitch at output y_1. Furthermore, because of this, state variable s_2 might temporarily turn on. If this behavior is observed by the machine, the result could be a false state transition from state 0 to state 1. In this specific case, the behavior of the machine will correct itself, since all next states for input combination 11 are equal to state 0. However, in general this will not be the case. Therefore, the delay box must have an additional property: it must filter glitches. A delay element that is capable of doing so is called an *inertial delay element*. Inertial delay elements can be quite large. If the environment is too slow, then even inertial delay elements are not able to remove all glitches. The environment therefore has to apply all input changes within a minimum time

δ_2, where δ_2 is determined by the circuit topology, and the delay elements of the machine.

Suppose the machine is in state 1, with input x_1 being 0, and input x_2 being 1. If x_2 turns off, gate g1 will turn off, and gate g2 will turn on. This behavior might cause state variable s_2 to temporarily turn off: a so–called off–glitch. This might again lead to a false state transition from state 1 to state 0. Here, the behavior of the machine does not correct itself, since state 0 is a stable state for input combination 00.

From these two observations, we can see that direct synthesis of Huffman machines can be a very tedious exercise. First of all, Huffman machines put severe timing constraints on the environment. Huffman machines are therefore not fit to be used in a modular design. Secondly, synthesis based on Huffman–machines can produce combinational logic that might suffer from hazards. In order to avoid these hazards, large inertial delays might be required, especially to suppress function hazards, which are explained in section 2.7.

To overcome these problems, people have proposed SIC (Single Input Change) Huffman machines. Since there is only one input change, there is no need for a δ_2 period. Also hazard–free synthesis algorithms do exist [Eich65]. However, the applicability of SIC machines is limited, mostly to the domain of counters.

In the 1960s and 1970s, much effort in the research on asynchronous finite state machines was targeted to finding efficient ways to implement flow tables such that the resulting circuits are free from hazards. Besides the architecture shown in figure 1.8, many other different architectures have been proposed. Friedman [Fried68] proposes an architecture without delay elements in the feedback loop of the state variables. The machine does require delayed versions of the inputs. Overviews of more general asynchronous architectures can be found in [Ung69],[Now93],[Mago71].

In [Wu91], a method is proposed that starts with a behavioral description of an asynchronous machine instead of a flow table. The user must partition the inputs into edge–controlled inputs, level–controlled inputs, and data inputs. The edge–controlled inputs are used to signal the network to perform some action, i.e., perform state changes or output changes.

The behavioral description is translated into a flow table which is implemented by a circuit. This approach no longer takes a given flow table as a starting point. By requiring the user to specify the behavior in a higher–level specification, the possible forms of the flow tables are limited.

The flow tables are basically restricted to the class of circuits known as pulse–mode circuits [Hill93].

Another approach no longer commencing with a given flow table is described in [Coat93]. The primary specification is a state–transition graph, where, instead of input vectors, input *changes* are specified. The latter is accomplished by augmenting the set of values to be assumed by a Boolean (input) variable (see section 2.7 for more details). This kind of specification is called a *Burst–Mode* specification, and the synthesis of asynchronous finite state machines that are specified like this, named Burst–Mode Machines, is the topic of this thesis.

Burst–mode specifications allow MIC (Multiple Input Change) specifications. However, a burst–mode machine does not need a δ_2 period, and its logic can be generated free from hazards. This means that burst–mode machines do not impose tight timing constraints on the environment. The designer can design a burst–mode machine without having to worry about the arrival times of signal changes. Therefore, burst–mode machines can be used in a modular design environment. Also, in a burst–mode based synthesis trajectory, the designer does not need to specify a lower/upper bound on gate delays and wire delays [Ung69]. This is because during synthesis of the logic part, an unbounded wire delay model is assumed. This assumption is more restrictive than the delay model applied in e.g. the synthesis of Huffman machines, but it allows the designer to apply any structural decomposition of the logic part, and it allows for any layout placement.

Burst–mode machines, however, like all asynchronous machines, do require delay elements to avoid essential hazards [Ung69], meaning that an estimation of the actual delay of the combinational logic is required. The designer is free to perform the estimation of these delays at any point in the design process. It will not change the logic part of the design. The main advantage of burst–mode synthesis over other automated asynchronous logic synthesis methods is that an implementation is guaranteed to exist. That is, for every valid burst–mode specification, an implementation at the gate level does exist. Existing STG–based synthesis methods, for example, cannot guarantee an implementation at the gate level.

1.7 Thesis outline

This thesis is organized as follows: in chapter 2, we provide the reader with some background information and notation that is used throughout this thesis. In chapter 3, we introduce the synthesis of burst–mode machines, the topic of this thesis. We show which steps are taken in the synthesis trajectory

by applying them to an example. Chapter 3 is concluded with a short overview of the contributions of this thesis. Chapters 4 to 8 cover the steps in more detail.

Chapters 4 and 6 cover logic synthesis. Chapter 4 introduces two–level minimization, chapter 6 discusses multi–level synthesis. Chapter 5 shows how we can verify combinational networks generated by the logic synthesis algorithms described in chapters 4 and 6. Chapter 7 covers state minimization and chapter 8 discusses state assignment. The chapters are out of order compared to the sequence in which these steps are normally applied. The reason for this is that much of the knowledge typically needed for logic synthesis is also necessary to understand the other steps.

Chapter 2 Background

2.1 Introduction

In this chapter we introduce some concepts and notation that will be used throughout this thesis. Most of the concepts are well–known, and have already been in use for a long time, see for example [Hill93],[McCl86]. In section 2.2 we start with the introduction of Boolean functions and expressions. We show how these functions and expressions can be used to model the behavior of a digital circuit with a single output. We introduce a special kind of expression that we will use many times in this thesis: the two–level expression. In section 2.3 we show how we can derive a minimum two–level expression. Minimum two–level expressions can be implemented very efficiently. In section 2.4 we extend the Boolean functions to incorporate variables that can assume more than two values. The new class of functions that are derived this way are known under the name multi–valued functions. In section 2.5, we make the relevance of these functions clear by showing that they can be used to model Boolean functions that describe multiple outputs.

In section 2.6 we introduce the concept of a delay model, which is used to examine the worst–case behavior of an asynchronous circuit after implementation. In section 2.7 we introduce the concepts of transitions and hazards. Our hazard–free synthesis algorithms are based on the hazard–free implementation of a given set of transitions. We show that it is possible to partition hazards into several classes.

2.2 Boolean functions

A fully specified Boolean function can formally be defined as follows:

DEFINITION 2.1
$f : B^n \rightarrow B$

Here, B is the Boolean set $\{0, 1\}$. A Boolean function maps a combination of values of n Boolean input variables onto a Boolean output variable. In this thesis the Boolean input variables will be denoted by x_i, where i is an integer

used to distinguish between individual input variables. Output variables will be denoted by y_i. The part of the input domain for which function f is equal to 1 is called the onset of function f. The part of the input domain for which function f is equal to 0 is called the offset.

DEFINITION 2.2

Let a Boolean function f be defined over B^n. $f|_{x_i}$ is called the positive *cofactor* of function f with respect to input variable x_i. $f|_{x_i}$ is defined as follows:

$$x \cdot f \Rightarrow f|_x \Rightarrow f + \bar{x}$$

A similar definition is possible for $f|_{\bar{x}}$, the negative cofactor of f. Sometimes it is necessary to explicitly calculate the cofactors. In those cases we need a uniquely defined version of the cofactor that still obeys definition 2.2. This version is usually based on the expression of function f as follows:

$$f|_{x_i} = f(x_1, ..., x_i = 1, ..., x_n) \tag{2.1}$$

Consider function $f = x_1\bar{x}_2 + \bar{x}_1x_2$. The two cofactors of function f with respect to input variable x_1 are: $f|_{x_1} = \bar{x}_2$ and $f|_{\bar{x}_1} = x_2$.

DEFINITION 2.3

f is *unate* in variable x_i if $f|_{\bar{x}_i} \Rightarrow f|_{x_i}$ or $f|_{x_i} \Rightarrow f|_{\bar{x}_i}$. In the first case the function is positive unate in x_i, in the latter case the function is negative unate in x_i. A function is unate if it is unate in each input variable.

Consider function $f = x_1x_2 + \bar{x}_2x_3$. Function f is positive unate in both variables x_1 and x_3. For example, function f is unate in variable x_1, because $f|_{\bar{x}_1} = \bar{x}_2x_3 \Rightarrow f|_{x_1} = x_2 + \bar{x}_2x_3$. Consider function $g = x_1x_2 + x_1x_3 + x_2x_3$. Function g is unate, since it is unate in all input variables.

PROPOSITION 2.1

For each function f holds:

$$f = x \cdot f|_x + \bar{x} \cdot f|_{\bar{x}}$$

This is also called a Shannon expansion [Brow90].

Boolean functions can also be defined incompletely. That is, the function is not defined in the entire domain B^n. By augmenting the codomain B, we can give a formal definition of an incompletely–specified Boolean function.

DEFINITION 2.4

An incompletely–defined function can be defined as:

$$f : B^n \rightarrow B_*$$

Here, the codomain is augmented by value $*$. This value is also known as the don't care value. So, $B_* = B \cup \{*\}$. For this type of function a don't care set can be defined. The don't care set is that part of the input space for which function f evaluates to value $*$.

DEFINITION 2.5
For Boolean functions, each possible subset $s \subseteq B$ is called a *literal*.

There are four possible literals, since there are four possible subsets, namely $\{\emptyset\}, \{0\}, \{1\}, \{0, 1\}$.

DEFINITION 2.6
A *cube* s is a non–empty cartesian product $s_1 \times \ldots \times s_n$ of literals, where each literal must be non–empty ($\neq \{\emptyset\}$).

If each input variable x_k assumes a value in literal s_k, then the cube is enabled, i.e., it evaluates to Boolean value 1. If $|s_i| = 1$, the cube is called a minterm. A minterm represents one combination of values assigned to the n input variables. A cube can be seen as a set of minterms. A cube is said to cover another cube if the set of minterms of the latter cube is a subset of the set of minterms of the former cube. If a cube has no intersection with the offset, meaning that there are no minterms for which function f is equal to 0, the cube will be called an *implicant*.

DEFINITION 2.7
A *prime* implicant is an implicant that is not covered by any other implicant.

An other definition would be: a prime implicant is an implicant that does not imply any other implicant. Boolean functions are defined by the designer by means of Boolean expressions. One type of Boolean expression that will be referred to many times in this thesis is the so–called *sum of cubes expression*, also called *two–level expression*. A sum of cubes expression uses a set of cubes to describe the part of B^n for which a function evaluates to 1.

A cube can be described by specifying its literals. The names of the input variables can be used for this purpose. Each cube is specified by specifying the *literal form* of each input variable. This is done as follows: Literal $\{1\}$ is described by the name of the input variable, e.g. x_i. Literal $\{0\}$ is described by putting a bar on top of the name of the input variable, e.g. $\bar{x}_i$. The full literal $\{0, 1\}$ is defined by omitting the input variable. Literal $\{\emptyset\}$ is not represented because of definition 2.6. A cube is invalid if one literal is equal to the empty set. The cube as a whole is then omitted. A concatenation of the literals encoded this way fully specifies a cube.

Suppose a Boolean function is built on four input variables x_1,x_2, x_3 and x_4. A cube $s = x_1\overline{x}_3x_4$ specifies the product $\{1\} \times \{0,1\} \times \{0\} \times \{1\}$.

In chapters 6 and 8 another encoding will be used to specify cubes. This encoding is sensitive in the order in which literal forms of variables are written down. In this encoding, literal {1} is written down as 1, literal {0} is written down as 0, and literal {0, 1} is written down as –. Symbol – is also referred to as a don't care value. With this encoding scheme, the above given cube is written down as: 1–01, where the sequence of the variables is $x_1x_2x_3x_4$.

For an incompletely–specified function, a sum of cubes expression can be used to define e.g. the onset, offset or the don't–care set.

For functions with a small number of input variables, a graphical representation can be used as a representation. These representations are called Karnaugh diagrams [Hill93]. An example of a Karnaugh diagram for a function with four input variables can be observed in figure 2.1.

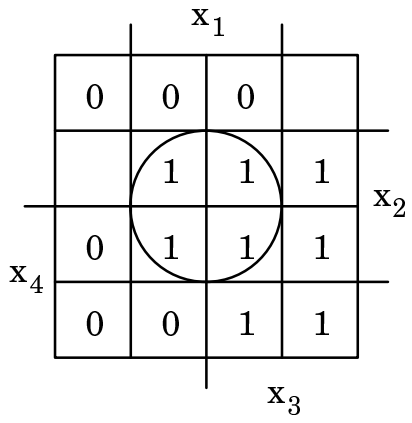


FIGURE 2.1. Example of a Karnaugh diagram

Each entry in a Karnaugh diagram represents a minterm. The empty entries correspond to minterms for which the function is unspecified, or in other words equal to don't care. This is a convention we will use throughout this thesis unless stated otherwise. In this case there are $2^4 = 16$ minterms. Each input variable partitions the Karnaugh diagram in two equal parts. In one partition the input variable is equal to 1 for all minterms, in the other partition the input variable assumes value 0 for all minterms.

The arrangement, and therefore the partitioning of the Karnaugh diagram, is such that two neighboring minterms can be merged into one cube that

covers both minterms. Also, two neighboring cubes can be combined into one bigger cube. The depicted ellipsoid in figure 2.1 corresponds with cube x_1x_2.

Boolean expressions not only define Boolean functions, they can also be implemented by digital circuits. It is possible to implement a sum of cubes expression directly as we will see in section 2.3. Usually, one is interested in an implementation of the onset of a Boolean function. In practice, the cost of such an implementation in terms of area is related to the number of cubes necessary. This cost can be reduced by also taking into account the don't care set in a search for a minimal implementation in the number of cubes.

2.3 Two–level logic minimization

A logic expression consisting of a sum of cubes is also called a two–level expression. Two–level expressions are used to describe the functionality of PLAs (Programmable Logic Arrays). The cost of a PLA in terms of area is directly related to the number of cubes, or implicants, in a two–level expression. The fewer implicants there are, the smaller the PLA is.

DEFINITION 2.8
A set of implicants is irredundant if by removing one implicant the resulting set of implicants no longer describes a given function correctly.

The goal of two–level logic minimization is to find an irredundant set of prime implicants necessary to describe a given function. An exact two–level minimization algorithm generates an irredundant set with the smallest cardinality. Small solutions in the number of cubes relate directly to the area used in PLA–based implementations. However, two–level logic minimization is not only important for PLA–based synthesis. In many multi–level synthesis systems, the result of two–level logic minimization is taken as an initial description. There is a rich literature on two–level minimization. In [Coud94] an excellent overview is presented.

Here, we briefly discuss one exact method, used by e.g. *espresso* [Rud87], [deM94]. The exact method first generates all prime implicants. There are several methods to do this. In espresso the set of prime implicants is generated by using the so–called *unate recursive paradigm*. From the set of prime implicants a minimum selection is made. The resulting set of cubes is minimum in the number of cubes necessary to describe the original function.

The unate recursive paradigm can be described by the following formula:

$$\begin{aligned} \text{Primes}(f) = \; & \text{SCC}(x \odot \text{Primes}(f|_x) \cup \overline{x} \odot \text{Primes}(f|_{\overline{x}}) \cup \\ & \text{CONSENSUS}(\text{Primes}(f|_x), \text{Primes}(f|_{\overline{x}})) \end{aligned} \tag{2.2}$$

Here, operator $s \odot T$ operates on a cube s and a set of cubes T. It calculates the Boolean product of s, with each cube in set T. In formula this is expressed as: $s \odot T = \{s \cdot t \mid t \in T\}$.

SCC is called the Single Cube Containment operation. It removes each cube that is covered by a single other cube. The CONSENSUS operation calculates the Boolean product of all possible combinations of the members of the two given sets of implicants f and g.

Formula 2.2 splits the original problem of calculating the set of all prime implicants into the smaller problems of calculating the set of prime implicants of belonging to $f|_x$ and $f|_{\overline{x}}$. These problems can again be split into smaller problems. If function f is described as a two–level expression, then we must continue this process until the two–level expression of the corresponding function has become unate. An expression is unate if each input variable only occurs in only one of the literal forms x, or $\overline{x}$. One can prove that the set of cubes, describing an irredundant unate two–level expression, is equivalent to the set of all the prime implicants of the represented function. Note that a unate expression and a unate function are not equivalent. Every unate expression describes a unate function. But not every expression representing a unate function is necessarily unate, although a unate expression does exist for each unate function.

An example is shown in figure 2.2. In figure 2.2 a Karnaugh diagram is used to represent a function that can be expressed in its minterms as: $x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 + x_1x_2x_3 + \overline{x}_1x_2x_3$.

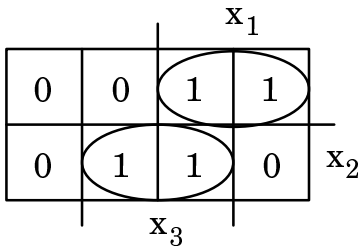


FIGURE 2.2. Example of a completely specified Boolean function

We take input variable x_2, and apply the unate recursive paradigm. The result is:

$f|_{x_2} = x_1x_3 + \overline{x}_1x_3$
$f|_{\overline{x}_2} = x_1\overline{x}_3 + x_1x_3$

Both expressions are not unate, i.e. in $f|_{x_2}$ both x_1 and $\overline{x}_1$ do occur.

Next, we take input variable x_1 to split $f|_{x_2}$ and x_3 to split $f|_{\overline{x}_2}$. This results in:

$f|_{x_2}|_{x_1} = x_3$ and $f|_{x_2}|_{\overline{x}_1} = x_3$

$f|_{\overline{x}_2}|_{x_3} = x_1$ and $f|_{\overline{x}_2}|_{\overline{x}_3} = x_1$

These expressions are all unate. With the help of these expressions we can calculate Primes($f|_{x_2}$) and Primes($f|_{\overline{x}_2}$).

$\text{Primes}(f|_{x_2}) = \text{SCC}(\{x_1x_3, \overline{x}_1x_3, x_3\}) = x_3$

$\text{Primes}(f|_{\overline{x}_2}) = \text{SCC}(\{x_3x_1, \overline{x}_3x_1, x_1\}) = x_1$

Finally:

$\text{Primes}(f) = \{x_2x_3, \overline{x}_2x_1, x_1x_3\}$

From this set of prime implicants we select the minimum number of prime implicants necessary to cover the onset of the function. In this case the minimum solution consists of two prime implicants, namely $x_1\overline{x}_2$ and x_2x_3. They are depicted as ellipsoids the Karnaugh map in figure 2.2.

The set of prime implicants can be quite large. In fact, it can be shown that the number of prime implicants is $\Omega(3^n/n)$, meaning that for a sufficient large number of inputs, there can be as many as $3^n/n$ prime implicants, where n is the number of inputs [Chan78].

2.4 Multi–valued functions

In a multi–valued function, variables can assume more than two values. A multi–valued function can formally be defined as follows.

DEFINITION 2.9

$f : V_1 \times ... \times V_n \rightarrow B$

A variable x_i can assume $|V_i|$ values. A multi–valued function maps a combination of values potentially assumed by n multi–valued variables $x_1...x_n$ onto a Boolean output. Definitions of literals, cubes, and prime implicants can be generalized from the definitions given for Boolean functions.

In multi–valued functions, sets of values, that can be assumed by variables, are denoted by literals. For a variable x_i, a literal is a subset of V_i. Therefore, there are $2^{|V_i|}$ literals for variable x_i.

DEFINITION 2.10
Each literal is uniquely defined by the notation $x_i^{s_i}$, where $s_i \subseteq V_i$. The expression $x_i^{s_i}$ evaluates to 1 if $x_i \in s_i$, else it evaluates to 0.

A literal can be represented with the help of a bitvector. Each bit in this vector represents one value. A bit is set to 1 if the corresponding value is part of the literal, else it is set to 0.

DEFINITION 2.11
A multi–valued cube is a non–empty cartesian product of literals: $x_1^{s_1} \times ... \times x_n^{s_n}$.

Given a multi–valued function f, an implicant is a cube that does not evaluate to one for any assignment yielding f = 0. In other words, an implicant is a cube that implies the multi–valued function.

DEFINITION 2.12
A multi–valued prime implicant is a multi–valued implicant that is not covered by any other multi–valued implicant.

For a more detailed description about properties of multi–valued functions, we refer to [Rud87] and [deM94]. We will make an exception for the cofactoring operation since this operation is used in chapter 4.

DEFINITION 2.13
Cofactoring can be defined as: $x_i^{s_i} \cdot f \Rightarrow f|_{x_i^{s_i}} \Rightarrow f + x_i^{\bar{s}_i}$

Here, $x_i^{\bar{s}_i}$ is the complement of $x_i^{s_i}$. This definition, like the definition for Boolean cofactoring, is not uniquely defined. To explicitly calculate a cofactor one could apply the definition of a multi–valued cofactor proposed by [Rud87].

The cofactor operation defined by Rudell is based on two cubes S and T. The cofactor of S with respect to T is empty if S · T = 0, else it is defined as $x_1^{s_1 \cup \bar{t}_1} \times ... \times x_n^{s_n \cup \bar{t}_n}$. With this definition we can perform a cofactor operation on an expression defined by a set of cubes, by calculating the cofactor of each cube separately.

DEFINITION 2.14
A multi–valued function is said to be weakly unate in a variable x_k if V_k can be partitioned into two sets s and t such that $f|_{x_k^s} \Rightarrow f|_{x_k^t}$. A function is said to be weakly unate if it is weakly unate in all variables.

There are many applications for multi–valued functions. In the next section, and in chapter 4, we show how multi–valued functions can be used to

represent a vector of Boolean functions. In chapter 8 we will show how multi–valued functions can be used to derive an efficient state assignment for a class of asynchronous finite state machines known as Burst–Mode Machines.

2.5 Representations of multiple–output functions

A vector of Boolean functions can be used to represent multiple outputs. Such a vector is formally defined as follows.

DEFINITION 2.15

$f : B^n \rightarrow B_*^m$

A vector of completely specified Boolean functions can be described by one multi–valued (or characteristic) function. This is performed by representing all output variables with one multi–valued variable, x_{n+1}. The number of values that can be assumed by this multi–valued variable is equal to the number of output variables. The relation between x_{n+1} and an output y_i is given by:

$$y_i = (x_{n+1} \equiv i) \tag{2.3}$$

So, in case of a function with 3 outputs, if x_{n+1} assumes value 2, output 2 is enabled. In this multi–valued variable, each value corresponds with the case that a corresponding output is enabled. The multi–valued function F is constructed as is shown in formula 2.4 [Coud94].

$$F(x_1, ..., x_n, x_{n+1}) = \prod_{i=1}^{m} \left(y_i \Rightarrow f_i(x_1, ..., x_n)\right) \tag{2.4}$$

For a vector of incompletely specified Boolean functions, at least two multi–valued functions are required. One can be used to specify the onset. The other one can be used to specify the offset. These two multi–valued functions are given by the following two formulas [Coud94]:

$$F(x_1, ..., x_n, x_{n+1}) = \prod_{i=1}^{m} \left(y_i \Rightarrow (f_i(x_1, ..., x_n) \equiv 1)\right) \tag{2.5}$$

$$R(x_1, ..., x_n, x_{n+1}) = \prod_{i=1}^{m} \left(y_i \Rightarrow (f_i(x_1, ..., x_n) \equiv 0)\right) \tag{2.6}$$

A third multi–valued function can be used to describe the don't–care set. However, this is not necessary since it can be derived from both the onset and

offset. The don't–care set represents a collection of minterms for which specific outputs are left unspecified. The don't–care set D is equal to $D = \overline{F + R}$.

An example of a vector of Boolean functions is given in figure 2.3

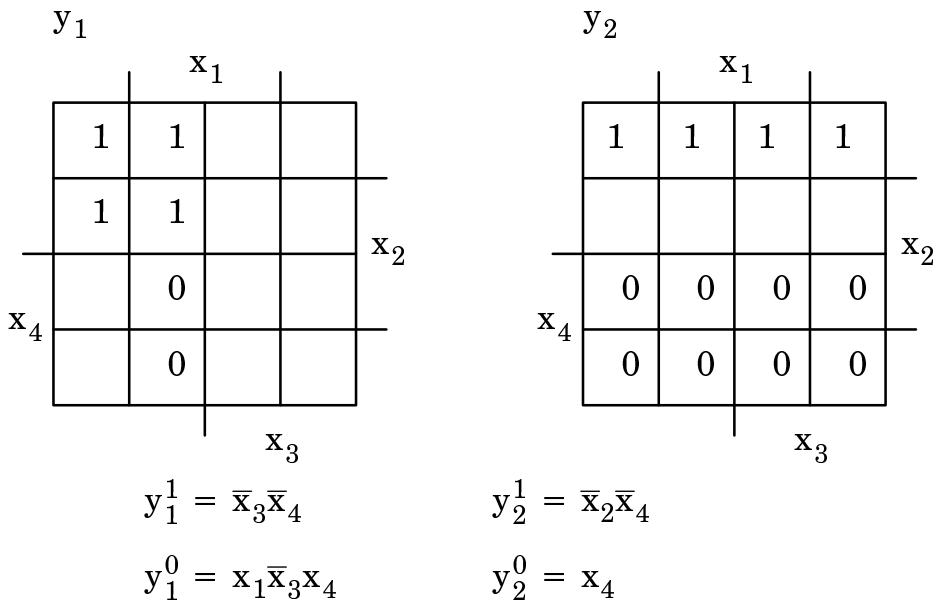


FIGURE 2.3. Example of a vector of Boolean functions

The vector of Boolean functions in figure 2.3 describes two outputs. For each output we can determine the sum of products expressions of both the onset and the offset. These expressions are also depicted in the above given figure. With the help of formulas 2.5 and 2.6, we can derive a sum of products expression for the characteristic function that expresses the onset, and for the characteristic function that expresses the offset.

$$F(x_{1,}x_2, x_3, x_4, x_{5)} = \overline{x}_3\overline{x}_4x_5^{\{1\}} + \overline{x}_2\overline{x}_4x_5^{\{2\}}$$

$$R(x_1, x_2, x_3, x_4, x_5) = x_1\overline{x}_3x_4x_5^{\{1\}} + x_4x_5^{\{2\}}$$

From these functions the characteristic function for the don't–care set can be derived.

$$D(x_1, x_2, x_3, x_4, x_5) = x_2\overline{x}_4x_5^{\{2\}} + x_3x_5^{\{1\}} + \overline{x}_1x_4x_5^{\{1\}}$$

The method that converts vectors of Boolean functions, or multiple output functions, to multi–valued functions seems quite complex. In the applications we are interested in, multiple–output functions are specified as a list of cubes, where each cube implies a number of outputs. Consider the following example:

$\overline{x}_1 x_2 x_3$ 0101
$x_2 x_3$ 1101

In this example, two cubes are shown representing a function with three input variables and four output variables. A bitvector is used to indicate which output variables are implied by each cube. The first cube implies outputs two and four. The second cube implies outputs one, two and four. A zero for an output variable is used to indicate that the cube has no meaning for that output variable. Note that the above given list of cubes can also be noted as:

011 0101
–11 1101

The bitvector that represents the multiple outputs can be considered as a bitvector representing a multi–valued literal. The given specification can therefore be converted trivially to a multi–valued function: $\overline{x}_1 x_2 x_3 x_4^{\{2,4\}} + x_2 x_3 x_4^{\{1,2,4\}}$. This function will evaluate to one if for a certain combination of input variables x_1, x_2 and x_3, there is a value for multi–valued variable x_4 that will enable the function. I.e, the multi–valued function will evaluate to one if there is an output, corresponding to a value of variable x_4, that will evaluate to one for the combination of the input variables x_1 to x_3.

2.6 Delay models

The result of asynchronous synthesis is the implementation of a given circuit specification in a certain technology. The last step in this process performs a mapping of a gate–level description of the circuit in a certain technology. This step can be modeled as an assignment of a certain delay to each wire and gate in the gate–level description of the circuit.

The delay values introduced by the last step can be the cause of a faulty, hazardous behavior of a circuit. Designers therefore would like to have a model that accurately predicts these delay values. Unfortunately, it is extremely hard to obtain a model that is accurate enough. To obtain the desired accuracy, designers often have no other choice than to actually perform the last (computationally expensive) step and to backannotate the delay values obtained this way in the gate–level description.

In order to solve this problem, a number of delay models have been proposed that were thought to be flexible enough in order to guarantee an implementation of a given gate–level description of an asynchronous circuit. The three most important delay models that have been, and still are, in use are:

- Bounded gate–delay model
- Unbounded gate–delay model
- Unbounded wire–delay model

In a bounded gate–delay model, the designer assumes that each gate in the gate–level description of his circuit has an unknown delay that is bounded by a certain upper bound. This model is used e.g. by Unger during the synthesis of general Huffman machines [Ung69]. Under the assumption of an unbounded gate–delay model, a circuit will continue to function, regardless of the delay assigned to any gate in the gate–level description of the circuit. Synthesis methods that fully rely on an unbounded gate–delay model are, as we saw in chapter 1, also called *speed–independent* methods. The last delay model, the unbounded wire–delay model, is the most general model in the sense that a circuit operating correctly under an this model, will operate correctly regardless the delay assignment of the last step. Synthesis methods that completely rely on this delay model are also called *delay–insensitive* methods.

A delay–insensitive method is only usable if the gate–level description of the circuit can take advantage of a rather complex set of gates. The DICY–system of Philips can be considered as a delay–insensitive method relying heavily on a set of pre–fabricated complex gates.

The difference between an unbounded gate–delay model and an unbounded wire–delay model is tricky and can be best explained by means of an example. Consider the following circuit.

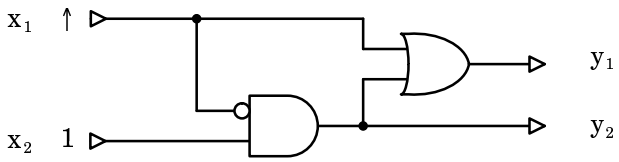


FIGURE 2.4. A circuit showing a different behavior under an unbounded gate–delay and unbounded wire–delay respectively

In the example circuit, input x_1 will change from 0 to 1. If the designer does assume an unbounded gate delay model, then output y_1 will remain on. However, if we use an unbounded wire delay model to model the implementation step, then output y_1 could temporarily turn off. This will happen if the delay assigned to the wire from input x_1 to the OR–gate is longer than the delay assigned to the path from input x_1, through the AND–gate, to the OR–gate.

In the circuit, input x_1 is *forked*. In an unbounded wire–delay model, each branch of the fork can be assigned a different delay value, whereas in an unbounded gate–delay model, both fork branches have an equal delay. Both the fork at input x_1, and the observation that the network implementing output y_1 contains a reconvergent path, are responsible for the difference in behavior. A circuit with no reconvergent paths will show the same behavior for both an unbounded gate–delay and an unbounded wire–delay model.

Some synthesis mehods do assume an unbounded wire–delay model, where delay assumptions are made for some of the forks that occur in the gate–level representation of the circuit. These methods are also known as *quasi delay–insensitive methods*.

2.7 Hazards and transitions in Asynchronous logic

In this thesis the term asynchronous logic will refer to a hazard–free (or glitch–free) logic implementation of a given set of transitions. In this section we formally introduce the concept of transitions, and show how these transitions can help classifying the possible sets of hazards that can occur.

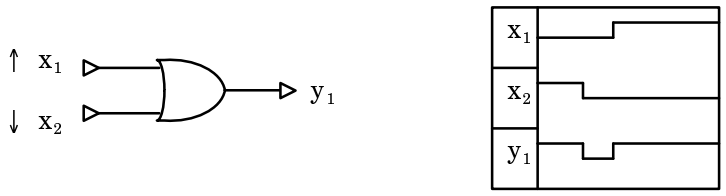
DEFINITION 2.16
Given the set $D = \{0, 1, \uparrow, \downarrow\}$, a transition T is defined as follows: $T \in D^n \times D^m$, where n is the number of inputs, and m is the number of outputs.

A transition maps a specific set of changes that takes place on the input variables onto a specific set of changes on the output variables. The set D represents all changes that are possible for a specific input or output variable. Symbols 0 and 1 represent non–changes, that is, the variable is assumed to maintain its value, which is e.g. 0 in the case that the symbol is 0. The up–arrow represents a change of a variable from 0 to 1. The down–arrow represents the opposite change. The set of changes on the input variables are not ordered in time: they may arrive in any order. Also, the set of changes on the output variables are not ordered in time. The changes on the output variables, however, may only occur when all changes on the input variables have been observed. We will call this the *Burst–Mode condition*.

An asynchronous logic circuit is said to be a hazard–free implementation of a given set of transitions if it implements each transition free from hazards under the chosen delay model. In this thesis, we will use the unbounded wire delay model. At the start of each transition, however, the circuit must be in rest. The latter does put a constraint on the environment. The environment is only allowed to apply a new transition when the circuit is in rest. The

asynchronous logic circuit contains no state and the sequence in which transitions are applied to the circuit can be completely random.

In order to guarantee a hazard–free implementation, we must examine what kind of hazards exist, i.e. we need to discover the classes of hazards. We start by giving an example of a hazard.



(a) (b)

FIGURE 2.5. Example of a function hazard

In figure 2.5a an OR–gate is depicted that is supposed to implement transitions $\uparrow 0 \rightarrow \uparrow$, $\uparrow\downarrow \rightarrow 1$ free from hazards. If we consider the second transition, then it is possible that a change of input variable x_2 is observed before the change of input variable x_1 occurs since the input changes of the input variables are not ordered. This might result in the OR–gate temporarily switching off, as can be observed in the time diagram in figure 2.5b. In an asynchronous circuit, such a behavior could result in other anomalous behavior since other parts of the circuit might respond to the OR–gate temporarily going off. So clearly this circuit isn't a correct hazard–free implementation of the two specified transitions.

Unfortunately, no circuit exists that can guarantee a correct implementation of the two transitions, regardless the delay model assumed. Both transitions specify a different output value for input combination $x_1 = 0$, $x_2 = 0$. According to the first transition, the output should be equal to 0, whereas the second transition specifies that the output should be equal to 1. It is clear that an asynchronous logic circuit cannot comply to both requirements. By choosing to comply one requirement, a hazard is introduced for the transition whose requirement is not met. We call such a hazard a *function hazard* [Ung69].

DEFINITION 2.17

A function hazard is a non–monotonic change of an output signal of an asynchronous logic circuit, which is caused by a contradiction in the specification, consisting of a set of transitions.

Consider the circuit in figure 2.6a as an implementation of the transitions $\uparrow 00 \rightarrow 1$, $1 \uparrow 0 \rightarrow 1$, $11 \uparrow \rightarrow 1$.

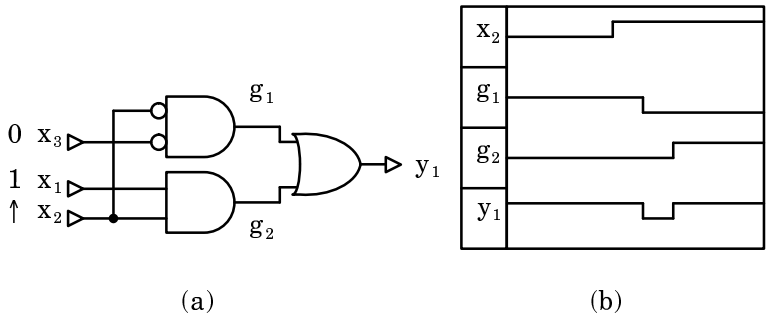


FIGURE 2.6. Example of a logic hazard

In the time diagram in figure 2.6b a possible response of the circuit is depicted for the second transition: $1 \uparrow 0 \rightarrow 1$. The output y_1 is supposed to remain stable at value 1, i.e. the circuit should be insensitive to the change of variable x_2. However, a hazard might occur, due to a so–called *term–takeover*. That is, in the circuit AND–gate g_1 switches off while g_2 switches on. In this case, the hazard is not due to a contradiction in the specification, but due to a timing mismatch: if the OR–gate would have observed the down–transition at gate g_1 after the up–transition at gate g_2, the output would have shown no hazardous behavior. This kind of hazard is called a *logic hazard*.

DEFINITION 2.18
A logic hazard is a non–monotonic change of an output signal, during a transition from a set of function–hazard–free transitions, which is caused by a certain assignment of delays to the gates/wires of the gate–level description of the circuit, in compliance with the delay model selected.

Both function and logic hazards can be split into two separate classes of hazards, namely static hazards and dynamic hazards. A static hazard can occur when a signal is expected to remain at a constant value, either 0 or 1. Both cases in figures 2.5 and 2.6 are examples of static hazards. The static hazard that is depicted in these figures is also called a static–1 hazard, since it does occur in signals that are expected to remain stable at 1. Static hazards occurring in signals that should remain stable at 0 are called static–0 hazards.

A dynamic hazard can occur in signals that are expected to make a monotonous transition from e.g. 1 to 0. An example of a dynamic hazard from the class of logic hazards is shown in figure 2.7. The circuit in this figure is supposed to implement transition $\uparrow 0 \uparrow \rightarrow \downarrow$:

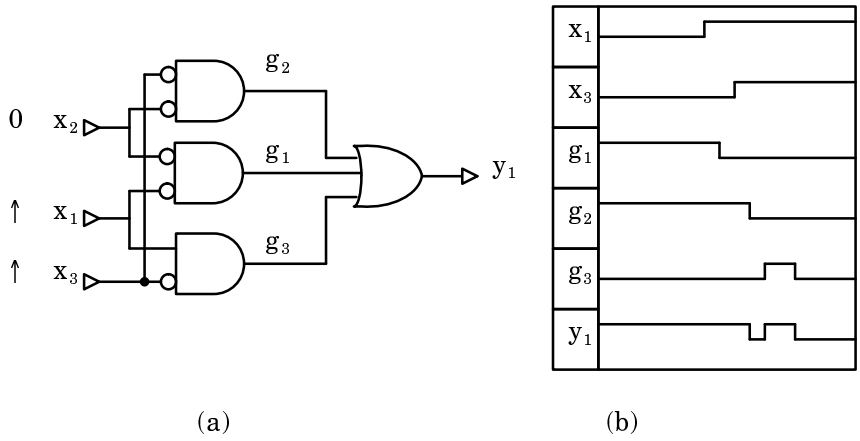


FIGURE 2.7. Example of a dynamic logic–hazard

It can be observed in figure 2.7b that a dynamic hazard occurs at output y_1. This hazard does occur because gate g_3 might temporarily turn on, if the change on input variable x_1 is observed before the change on input variable x_3. Because we assume an unbounded wire delay model, the behavior of gate g_3 might be observed by the OR–gate after it has observed the monotonous transitions on gates g_1 and g_2.

An overview of the possible hazards that can occur in combinational circuits (that is circuits without memory) is represented in figure 2.8.

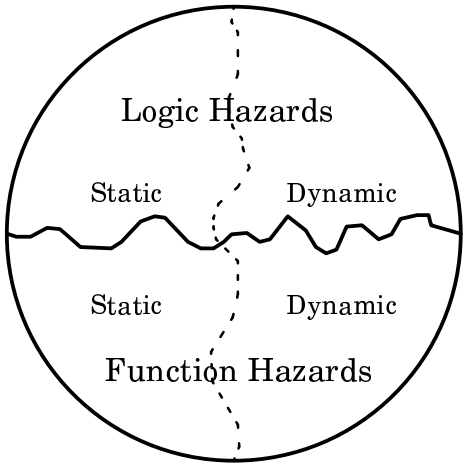


FIGURE 2.8. Classification of combinational hazards

During the synthesis of the combinational part of asynchronous logic, we have to take into account those hazards in figure 2.8 that are classified as

logic hazards. Function hazards represent a contradiction in the specification of the transitions, and can only be removed/avoided by changing the specification. This can be done by the designer, or by an automated synthesis system, by adding state variables, i.e. adding memory to a circuit. By adding memory to circuits, we can use the finite state machine model to describe these circuits.

The original classification of hazards and foundations of hazard–free synthesis are by [Beis74], [Bred72], [Eich65], [Frac74], [Ung69].

Chapter 3 Synthesis of BMMs

3.1 Introduction

In chapter 1 we saw that designing a completely asynchronous system with the help of Huffman machines can be a tedious, mostly manual, exercise. To overcome the main disadvantages of Huffman–machine based synthesis, we introduce a restricted form of Huffman machines in this chapter, called *Burst Mode Machines* (BMM). In section 3.2 we introduce BMMs, a formal model is given in section 3.3. In section 3.4 we introduce the BMM architecture we will use in this thesis. In section 3.5 we discuss some of the advantages/disadvantages of BMM–based synthesis. In this thesis, synthesis of BMMs is accomplished by adopting and modifying the synthesis steps common to synchronous and Huffman based synthesis. In section 3.6 we introduce these steps and show the modifications necessary, by applying these steps to an example specification. In section 3.7 we give a short overview of the history on the development of BMM–based synthesis. We conclude with a list of our contributions in section 3.8.

3.2 BMMs Introduction

A BMM, like a Huffman machine, operates under the fundamental mode condition, allowing multiple input changes. BMMs allow the environment to apply input changes over an unbounded time interval. A BMM can do this because in each state it knows the possible sets of input changes that can be applied by the environment. Therefore, it can determine the end of a sequence (or *input burst*) of input changes. So, compared with Huffman machines, BMMs don't have a fixed evaluation period δ in which the environment must apply all input changes to the machine.

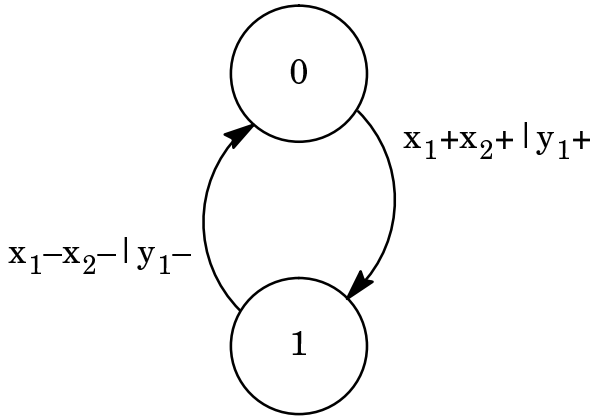


FIGURE 3.1. Example of a Burst Mode Machine

In figure 3.1 an example of a BMM is specified as an ordinary state–transition graph consisting of two states. For the specification of BMMs, we adopt the graph–specification conventions introduced by [Coat93] Coates et. al. The edges are annotated with changes of input and output variables. Changes themselves are represented by '+' and '–' instead of ↑ and ↓. Variables that remain stable are not shown in the annotation. Input and output variables are clearly separated by symbol '|'. In the case that all output variables remain stable, symbol '|' is not depicted. In our example, the machine knows that the environment will change both input variables x_1 and x_2 when it is in state 0. Both variables will change from 0 to 1. When both changes have been observed, the machine will respond by changing its internal state from state 0 to state 1, and by changing output variable y_1 from 0 to 1. It is not allowed to specify an empty input burst.

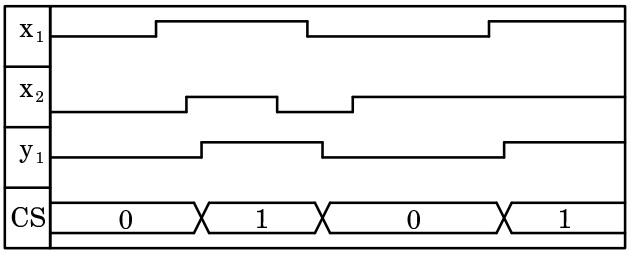


FIGURE 3.2. Typical behavior of the machine specified in figure 3.1

A typical behavior of the specification of the machine in figure 3.1 is shown in figure 3.2. In this figure we can observe that input changes can occur in any order and that they are not bound by any delay interval. After an input burst has been completed, then, after some time which is determined by the actual

implementation, both the the output and the current state of the BMM make the required changes.

In general, the designer is allowed to specify multiple input bursts starting in the same state of the BMM. Nowick [Now91] proposed two restrictions in order to guarantee a hazard–free implementation.

The first restriction states that it is not allowed that an input burst is specified as a subset of another specified input burst, when both input bursts originate from the same state. This restriction is known as the maximal set property [Now93]. An example should clarify this. Consider a part of a specification of a BMM in figure 3.3.

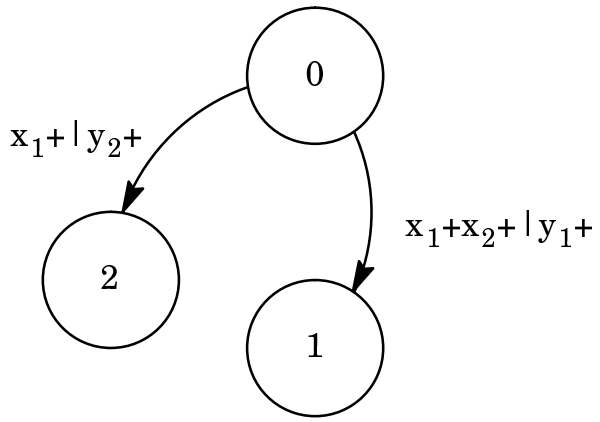


FIGURE 3.3. Two conflicting input bursts, starting in the same state

We can see that two input bursts originate from state 0. Note that input burst x_1+ is a subset of input burst x_1+x_2+. The BMM can therefore no longer detect with certainty that an input burst has completed because after a change of input variable x_1, a change of input variable x_2 might follow. The only way to discern between the two input bursts is by demanding that the changes of variables x_1 and x_2 occur in a fixed time interval δ. Therefore, this interval can no longer be unbounded. So, the above given specification can not be implemented by a BMM, although it could be implemented by a Huffman machine.

After a BMM has observed all input changes belonging to a certain input burst, it will change its state to the next state. The values of the input variables after this input burst uniquely describe an entry point of the next state. A BMM specification in general allows each state to have more than one entry point. The second restriction states that each state must have a unique entry–point, even if it is possible to enter the same state by two or more edges

in the state–transition graph. This restriction is necessary in our synthesis approach in order to avoid specifications that cannot be implemented free from hazards by a two–level implementation. An explanation of this will be given in section 3.6.1. Furthermore, the values of the output variables at the given entry point must be consistent. To clarify this, an example is given in figure 3.4. In this figure a specification is given of a machine with two inputs: x_1 and x_2, and one output: y_1.

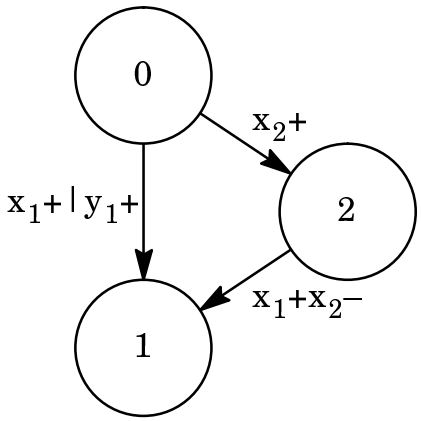


FIGURE 3.4. A conflicting output specification for state 1

In figure 3.4 there are two edges that lead to state 1. Both edges enter state 1 by the same entry point. However, the edge originating from state 0 specifies output y_1 to be equal to 1, whereas the edge originating from state 2 specifies output y_1 to be equal to 0. This means that for the input entry two different output values should be implied *simultaneously*. This is of course impossible.

The two restrictions discussed are sufficient in order to guarantee that an implementation of the machine is possible. Furthermore in contradiction to Huffman–based implementations, it is possible to show that BMMs only need pure delay elements, instead of inertial delay elements. In practice it turns out that many BMMs operate without any delay element at all [Chak97].

3.3 Formal definition of a BMM specification

The following formal definition is by Nowick [Now93].

DEFINITION 3.1
A BMM specification is a directed graph G defined by the tuple $(V, E, I, O, v_0, in, out)$, where:

- V is the set of states
- $E \subseteq V \times V$ represents the edges
- $I = \{x_1, ..., x_n\}$ represents the input signals
- $O = \{y_1, ..., y_m\}$ represents the output signals
- $v_0 \in V$, is the start state
- $in : V \rightarrow B^n$, together with $out : V \rightarrow B^m$ are functions specifying the entry point of each state.

BMMs are accompanied by two labeling functions $trans_i : E \rightarrow P(I)$ and $trans_0 : E \rightarrow P(O)$, where, function P returns the powerset of the set given. Both functions are defined as follows:

$$trans_i : ((u, v)) = \{x_j \mid in_j(u) \neq in_j(v)\} \tag{3.1}$$

$$trans_o : ((u, v)) = \{y_j \mid out_j(u) \neq out_j(v)\} \tag{3.2}$$

Each edge is therefore labeled with a set of changing inputs and outputs. The sign of each changing input/output can be reconstructed by examining the entry points of the two states that are connected by the edge.

By the above definition, each state has a unique entry–point. However, as we saw in the previous section, the maximal set property is also required to make sure that the BMM is *well–formed*, i.e. that we can guarantee an implementation.

DEFINITION 3.2
A BMM specification is said to be well–formed if:

$trans_i((u, v)) \subseteq trans_i((u, w)) \Rightarrow v = w$ and $trans_i((u, v)) \notin \emptyset$.

So, a BMM is well–formed if it has the maximal set property and if an input burst cannot be empty. In this thesis, we will only consider well–formed BMM specifications.

3.4 BMM architecture considerations

Here, in this thesis, we map BMMs on the architecture shown in figure 3.5. Basically, it is the same architecture as the one used for implementing Huffman machines, introduced in chapter 1.

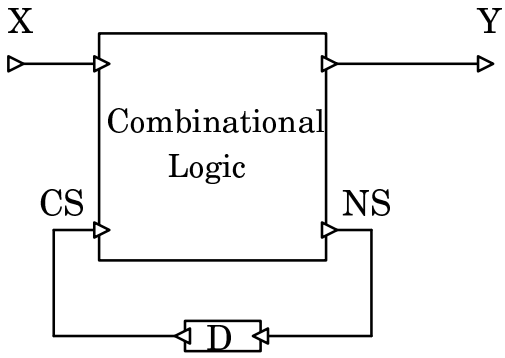


FIGURE 3.5. General architecture of BMMs

When a BMM observes a valid input burst, the combinational part in figure 3.5 responds by, possibly, generating an output burst on Y, and a state transition on NS. We emphasize the fact that the machine *can* respond this way. As we already noticed, an input burst does not always trigger an output burst. Furthermore, an input burst will not always cause a state transition because of a synthesis step known as state reduction, which is introduced in section 3.6.1.

Whatever the response of the combinational part, a certain amount of time is needed for it to become stable again. The delay elements in figure 3.5 are necessary to make sure that the combinational logic does not observe the effect of a state transition before it has stabilized from its evaluation of an input burst. This is necessary in order to avoid essential hazards [Ung69]. Unfortunately, it also does put a constraint on the environment. The environment is not allowed to apply new input changes before the machine has stabilized.

The delay elements in figure 3.5 force the combinational logic to operate on two separate transitions: an *input transition* initiated by the environment, and a *current state transition*, calculated by the combinational logic block itself. More details can be found in section 3.6.3.

Given a BMM specification, that obeys all the necessary constraints described in the previous section, it is possible to prove that a correct implementation of the combinational logic part exists. This means that we can implement the desired behavior without any hazards under the unbounded wire–delay model [Now93], which was introduced in section 2.7. The latter guarantees that no hazards will occur, no matter the layout of the final implementation of the BMM, as long as the delay elements are such that the current state transition is observed after the input transition.

Each BMM has a *unique start state*. When the corresponding circuit is first initialized, additional circuitry is needed to make sure that the current state of the machine is initialized with this start state.

3.5 Advantages/ Disadvantages of BMM based synthesis

There are two major advantages of BMM–based synthesis methods over synthesis methods that are based upon general Huffman machines.

- BMMs don't require inertial delays.
- BMMs allow for a modular design.

In the first place, BMMs don't need inertial delay elements to remove spurious glitches. A designer that specifies a correct BMM can rely on the fact that a correct hazard–free implementation is guaranteed to exist, for any library in any technology. The fact that no inertial delay elements are required implies that BMM machines are smaller in terms of area, and faster than general Huffman machines.

The modular design issue addresses the point that the designer doesn't have to worry about the delays of specific signals in the circuit, when he is designing the circuit with the help of BMMs. Each BMM will wait until a specified input burst has been completed. After that, it will complete the specified input transition, i.e. it will generate a state transition, and an output burst. So during BMM synthesis it is no longer required to take into account the $[\delta_1, \delta_2]$ intervals of other connected BMMs, as is the case in a Huffman machine based design.

BMM–based synthesis methods also have some disadvantages.

- Most synthesis algorithms need some modifications.
- The designer has to specify his design as a BMM specification. A BMM specification is restricted in what it can express.

As we will see in the next section, during BMM synthesis it is no longer possible to apply the synthesis steps used to generate e.g. Huffman machines, or synchronous machines. We will show that some modifications are necessary.

Huffman machines can be used to implement the specification of any synchronous machine. This is not the case for BMMs. BMMs operate on input bursts. Huffman machines and synchronous machines operate on input vectors. The consequence is that not every synchronous finite state machine specification, or Huffman–machine specification can be converted to a BMM specification.

3.6 Synthesis of BMMs

We consider the synthesis steps necessary to map BMMs on the general architecture shown in figure 3.5. Each synthesis step transforms a specification into an intermediate result, which can be considered as the specification of the next step.

Despite the important differences, BMMs are still very similar to basic Huffman and synchronous machines. Therefore it makes sense to try to apply the same synthesis steps, although some modifications are necessary. The modifications are necessary to make sure that the intermediate result produced by each step will not lead to a specification that is infeasible by the next steps. The classical synthesis of synchronous machines, and asynchronous Huffman machines consists of four steps. Each step must be modified to ensure a correct implementation. The four steps, with the modifications necessary, are:

- State reduction. During this step the number of states necessary to describe a BMM is reduced as much as possible. One step in the state reduction process, as will be explained in more detail in chapter 7, consists of determining the set of incompatible state pairs. The states belonging to such a pair cannot be merged into one new state. Nowick [Now94] shows that extra incompatible state pairs are needed to avoid non–removable logic hazards in a two–level implementation.
- State assignment. In this step, each state is assigned a binary code. The code must be chosen carefully in order to avoid the introduction of critical races. Critical races occur when the behavior of the machine depends on the order in which two or more state variables switch during a state transition. Critical races are a form of function hazards.

The first two synthesis steps, state reduction and state assignment, convert a BMM specification into a set of transitions. Both steps must be able to guarantee that this set of transitions is free from function hazards. They also must be able to guarantee that the set of transitions can be implemented free from logic hazards in a two–level implementation.

- Logic synthesis. During this step a logic hazard–free implementation of the above mentioned set of transitions is derived. Conventional logic synthesis algorithms must be modified, since they operate on input vectors instead of input transitions.
- Technology mapping. In this step, the logic description of the BMM is mapped onto a given library in a given technology. In this step care must be taken not to introduce hazards while mapping on this library. Boolean mappers for example are in general not usable. A safe method is to apply only structural mappers that use so–called hazard–non–increasing transformations [Kung92].

The modifications necessary to each of the steps described above, are discussed and demonstrated by means of an example. In figure 3.6 a specification is given of a BMM. We will apply the synthesis steps mentioned above, and discuss some of the problems related with these steps.

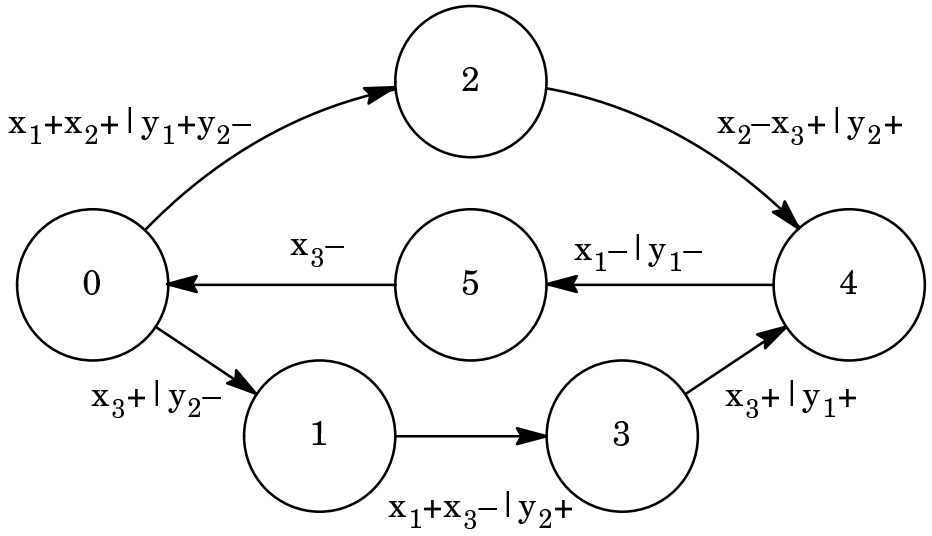


FIGURE 3.6. BMM specification of "*Bad–Merge*" [Now94], a specification with 3 inputs and 2 outputs

To provide a better insight we convert the specification in figure 3.6 into a flow table shown in figure 3.7.

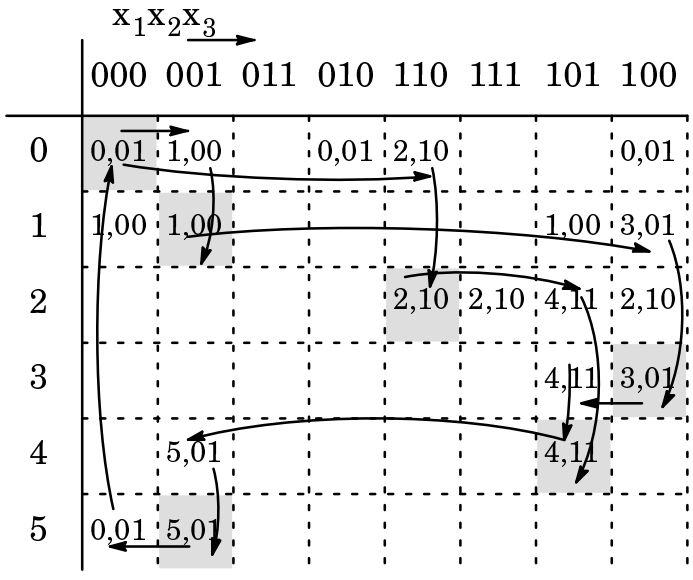


FIGURE 3.7. Flow table representation of "*Bad–Merge*"

Each edge in figure 3.6 is converted to two separate arrows in figure 3.7. This is because, as we saw before, each edge in figure 3.6 actually consists of two transitions: an input transition and a current state transition. During the input transition, the environment changes a set of input variables. If these changes, according to the specification, correspond with a valid input burst, the machine will apply the annotated changes to the output variables, and it will make the necessary changes to the state variables in order to make the required state transition. Note that as long as not all input changes have been observed, the machine must remain in the same state, and maintain the values of the output and state variables. This is why in e.g. state 0 input entries 010 and 100 are specified. During the second transition, the current state transition, the BMM observes the changes to the state variables. The second transition is applied when the machine has stabilized from the first transition, and is represented by a vertical arrow. During the second transition the outputs and the state variables do not change.

The next sections will describe the mentioned synthesis steps, and the modifications necessary to each step in more detail.

3.6.1 State reduction

In the first step we reduce the number of states, needed to describe the BMM, as much as possible. We merge states such that the behavior of the machine remains unchanged. For this purpose, we use the concept of incompatible state pairs. Two states are incompatible if they imply different output values for the same input entry, or if they imply incompatible states. More details can be found in chapter 7. By examining the flow table in figure 3.7 it is clear that it is possible to merge states 3,4, and 5: state pairs $(3, 4)$, $(3, 5)$, and $(4, 5)$ are all compatible. If we use a synchronous state–minimizer, such as stamina, which is part of the SIS–package [Sent92], it will also show that states 0 and 3 can be merged: For entry 100, both states maintain their current state, and both states set output y_2 to 1. In all other entries at least one of the states shows a don't care behavior. Therefore, a synchronous version of the reduced flow table, shown in figure 3.8, consist of three states, where the new states are assigned as follows: $0' = \{0, 3\}$, $1' = \{1\}$ and $2' = \{2, 4, 5\}$.

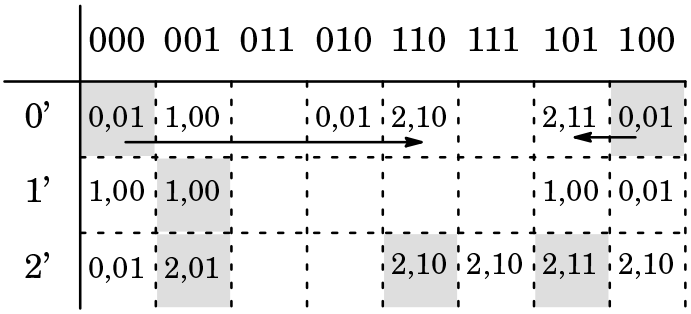


FIGURE 3.8. Minimal flow table of the specification in figure 3.7

Now, let us examine the behavior of the machine that is implemented using the minimized flow table. We focus on the second output variable y_2. Furthermore, we only consider the two input transitions depicted in figure 3.8, so we do not consider the resulting current state transitions. Thus, we seek for a network that correctly implements transitions $\uparrow\uparrow 0 \rightarrow\downarrow$ and $10 \uparrow\rightarrow 1$. A possible two–level network, that implements these transitions, is shown in figure 3.9. Transition $\uparrow\uparrow 0 \rightarrow\downarrow$ is implemented by gates g_1 and g_2. Gate g_2 makes sure that output y_2 stays at one as long as only a change of input variable x_1 has been observed. Similarly, gate g_1 makes sure that output y_2 stays at one in the case that input variable x_2 changes its value first. Transition $10 \uparrow\rightarrow 1$ is implemented by gate g_3. This gate makes sure that during the change of input variable x_3 the output remains stable at one.

If we examine the network consisting of gates g_1, g_2, and g_3 in combination with transition $\uparrow\uparrow 0 \rightarrow\downarrow$, a dynamic hazard is possible, as can be observed in the time diagram in figure 3.9.

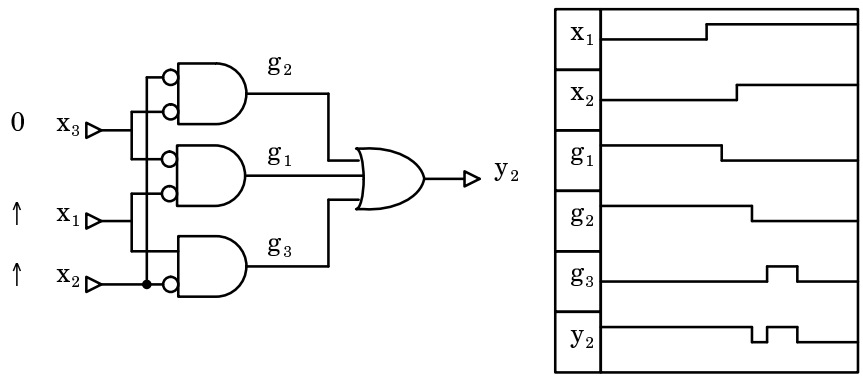


FIGURE 3.9. Hazardous implementation of the two input transitions in figure 3.8

This dynamic hazard does occur if input variable x_1 changes first. In that case gate g_3 can temporarily turn on. Due to our unbounded wire–delay model the behavior of gate g3 might be observed after both gates g_1 and g_2 have turned off.

In a two–level implementation, like the one shown in figure 3.9, it can be shown that this hazard is *unavoidable*. No two–level network, consisting of a sum of cubes, exists that implements the two transitions free from hazards. The dynamic hazard can only be avoided by not merging states 0 and 3 in the original flow table in figure 3.7. Therefore, additional constraints must be added to the state reduction process. In this case a correct minimized flow table consists of 4 states, where the new states are: $0' = \{0\}$, $1' = \{1\}$, $2' = \{2\}$ and $3' = \{3, 4, 5\}$. This flow table is shown in figure 3.10.

	000	001	011	010	110	111	101	100
0'	0,01	1,00		0,01	2,10			0,01
1'	1,00	1,00					1,00	3,01
2'					2,10	2,10	3,11	2,10
3'	0,01	3,01					3,11	3,01

FIGURE 3.10. A reduced flow table of the specification of figure 3.7 that can be implemented without hazards

The example in this section also shows why each state in an original BMM specification must have a unique entry point. If we would let go of this constraint, then the reduced BMM specification, with the corresponding flow table in figure 3.8, would also be a valid BMM specification. As we saw, in our synthesis approach, we cannot implement this BMM specification free from logic hazards in a two–level implementation.

3.6.2 State assignment

The second step in the synthesis of a BMM consists of state assignment. During this step all the states are assigned a binary code. For synchronous machines, state assignment can be straightforward. The only constraint that must be obeyed is that each state must be assigned a unique code. Therefore for N states, at least $^2\log N$ state variables, making up the binary codes of

the states, are needed. This is no longer the case in BMM–based synthesis methods, and for asynchronous machines in general. Suppose we use the state encoding in figure 3.11 for the flow table in figure 3.10.

S	Encoding
0’	00
1’	01
2’	11
3’	10

FIGURE 3.11. State assignment of the flow table in figure 3.10

Let us examine the state transition in this flow table, for input column 100, between states 1’ and 3’. During this transition both state variables change. If the first state variable changes first, state 2’ might be entered. Since the next state in state 2’ for input column 100 is state 2’, the machine might end up in that state. This clearly leads to incorrect behavior. This kind of behavior is also indicated by the term *critical race*. A critical race can occur if at least two state variables change in value during a state transition, and if the behavior of the machine depends on the order of the changes in time of these state variables. In this case both state variables are required to change simultaneously.

Critical races can be avoided by putting extra constraints on the state encoding. This might lead to the introduction of extra state variables. In this case, at least three state variables are needed. In figure 3.12 a correct critical–race–free encoding can be observed.

S	Encoding
0'	000
1'	011
2'	101
3'	110

FIGURE 3.12. Critical–race–free encoding of the flow table in figure 3.10

If we again examine the state transition between states 1' and 3', we see that state 2' is no longer entered. This is because the second state variable is equal to 1 in states 1' and 3', and equal to 0 in state 2'.

3.6.3 Logic synthesis & Technology mapping

The third step in our synthesis trajectory is logic synthesis. During this step the combinational logic part of the machine is instantiated. All the edges of the original BMM state–graph specification are mapped onto a flow table. We saw that each edge in this state graph corresponds to two separate transitions, one input transition, and one current state transition. Because the state reduction step can reduce the flow table in the number of states necessary, the number of current state transitions could also be reduced. The set of all transitions, consisting of all input transitions and all current state transitions, is instantiated with the help of the state assignment found during the state assignment step.

For example, in figure 3.6 the edge between states 1 and 3, annotated with $x_1+x_3-|y_2+$, is converted into input transition $\uparrow 0 \downarrow\ 011 \rightarrow \uparrow 1 \downarrow\ 0 \uparrow$, and current state transition $100\ \uparrow 1 \downarrow \rightarrow 110\ 01$. The input part of each transition consists of six variables. The first three variables, belong to the primary input variables x_1, x_2, x_3. The other three variables correspond with the current state variables, xs_1, xs_2, xs_3. The output part of each transition consists of five variables. The first three variables represent the next state variables, ys_1, ys_2, ys_3 and the last two variables correspond with the two output variables y_1 and y_2. Note that the state variables occur both as inputs and as outputs in each transition, which is in accordance with the architecture in figure 3.5. Each next state variable is connected to its

corresponding current state variable by means of a delay element. So, ys_1 is connected to xs_1, ys_2 is connected to xs_2 and ys_3 is connected to xs_3.

The goal of the logic synthesis step is to find a hazard–free implementation of the set of transitions derived. Because of the modifications to the state reduction, and state assignment steps, a hazard–free implementation of the given set of transitions is guaranteed to exist. That is: each transition can be implemented free from hazards under the unbounded wire–delay model. More stronger: a hazard–free two–level implementation of the given set of transitions does exist.

From the computed set of transitions, the onset, don't–care set, and offset of each state variable and each output variable can be derived. With the help of a normal two–level minimizer it is possible to calculate a minimum implementation in e.g. the number of cubes. However, this solution is in general not free from hazards. It can be shown [Ung69] that there are two kind of hazards that can occur in two–level implementations: static–1 hazards and dynamic 1–to–0 hazards.

As we saw in chapter 2, the first kind of hazard describes the possibility for an output (or state variable) to temporarily turn off during a transition. This will typically occur when an AND–gate turns off, while another AND–gate switches on: a term–takeover. This kind of hazard can be avoided by introducing a cube that remains static at one during the transition. This cube is called a *required cube*. If the solution is to be free from static–1 hazards, then the implementation of each static output and state variable must contain an implicant that covers the corresponding required cube.

A dynamic 1–to–0 hazard occurs if an output does not change monotonically from 1 to 0, but instead can show numerous changes before settling to value 0. There are two possible causes for this behavior. A dynamic hazard can, again, be caused by a term–takeover. That is: the dynamic hazard actually is a static–1 hazard followed by a monotonous change of the output from 1 to 0. Required cubes are again needed to avoid this kind of behavior.

The second cause for a dynamic 1–to–0 hazard is due to an implicant that temporarily switches on. It is possible that the behavior of this implicant will be observed after the output has made a monotonous transition from 1 to 0. An example of this behavior is given in figure 2.7. This kind of hazard can be avoided by demanding that, during a transition, no implicant can temporarily switch on. All implicants in the implementation of the output may only switch off, or remain static at 0. To ensure that the implicants in a solution obey this constraint, *privileged cubes* are introduced. A privileged cube consists of two cubes, where the first cube describes all the points/minterms that can be

entered during the transition. The second cube describes the starting point/minterm, from where the transition does start. An implicant in an implementation is said to illegally intersect a privileged cube, if it intersects the first cube, but does not intersect the second cube. This implicant might temporarily switch on during the transition. In this thesis we will show several methods that guarantee that the implicants in the found two–level solution do not intersect any privileged cube illegally.

In figure 3.13 an implementation is shown, calculated by a common synchronous minimizer. In this implementation, the delay elements that connect the current state variables and the next state variables are not shown. This implementation is hazardous. We can show this by considering input transition $\uparrow 0 \downarrow \ 011 \rightarrow \uparrow 1 \downarrow \ 0 \uparrow$. During this transition the second state variable must remain stable at one. If we examine the circuit in figure 3.13, and consider only the AND–gates that drive the second state variable, we can observe that at the start of the transition, gates 7 and 8 are on. However, during the transition, both gates 7 and 8 switch off, while gate 10 switches on. Therefore, we can observe a static–1 hazard at the second state variable during this transition. This clearly is a faulty implementation of the combinational logic part of the BMM.

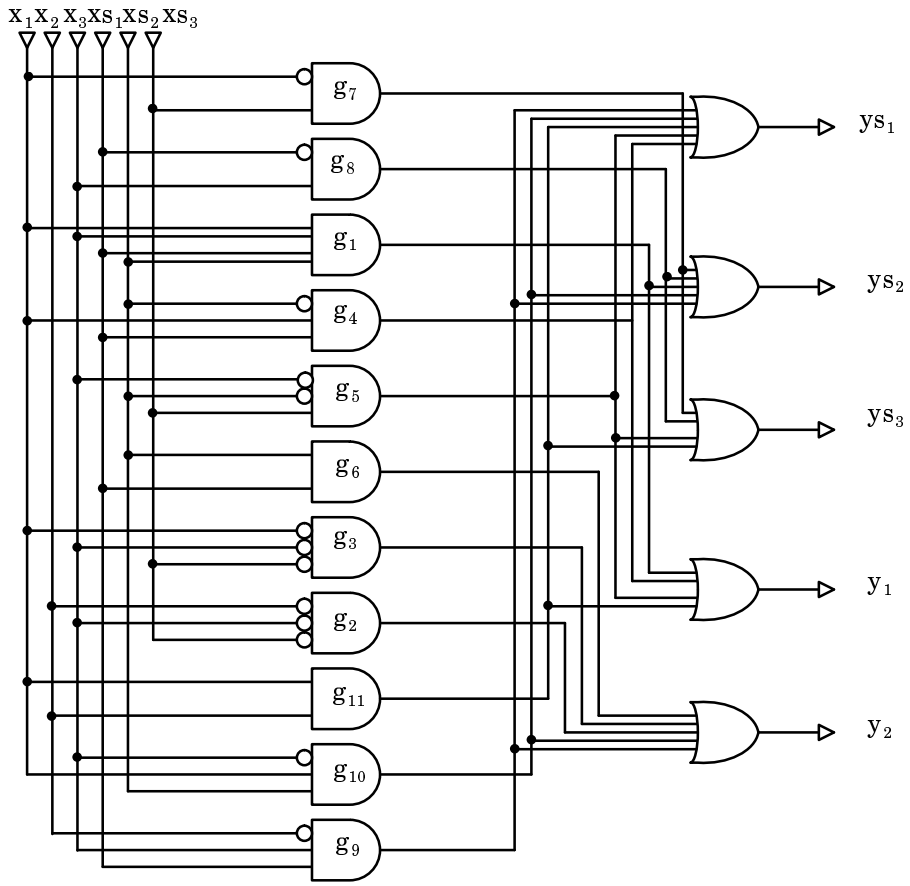


FIGURE 3.13. Hazardous implementation of the BMM in figure 3.6

In figure 3.14, a correct hazard–free two–level implementation of the combinational logic part of our example BMM is depicted. Note that the hazard–free implementation is somewhat larger in the number of cubes: thirteen cubes versus eleven cubes for the hazardous solution. In general, generating hazard–free logic tends to give a slight overhead.

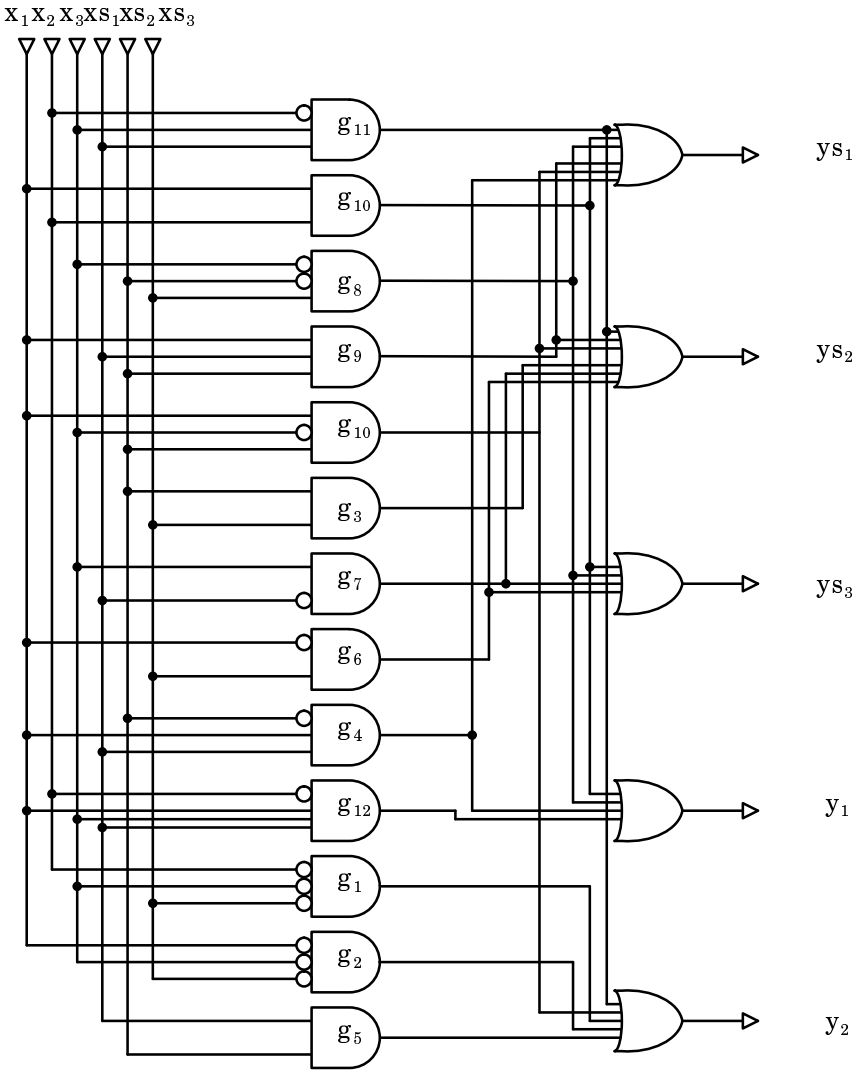


FIGURE 3.14. Hazard–free implementation of the BMM in figure 3.6

By using hazard–non–increasing transformations, the computed two–level solution can be shaped into a multi–level solution [Kung92]. For example, both common cube extraction and kernel extraction have been proved to be hazard–non–increasing transformations. Applying hazard–non–increasing transformations can lead to the network depicted in figure 3.15. This network has been mapped on a complete 3x3 library. Note that in practice a library will never contain all the cells belonging to a 3x3 library. However, any structural mapper, that only applies those transformations that have been proved to be hazard–non–increasing can be used, so the example, and any two–level implementation of any BMM, can be mapped onto any library.

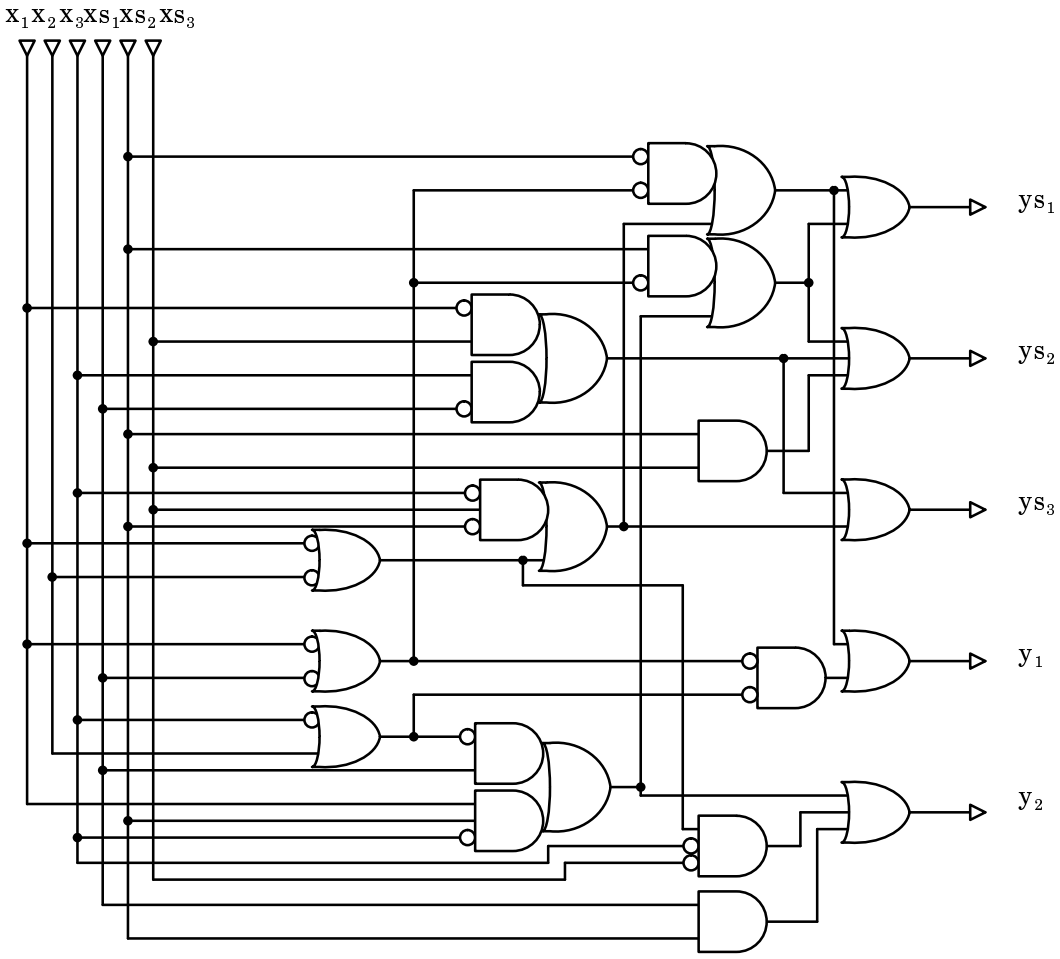


FIGURE 3.15. The combinational logic in figure 3.14, mapped onto a complete 3x3 library

The resulting end point of our synthesis system is the combinational logic part of a BMM. A BMM may also incorporate a set of pure delay elements. In order to determine the amount of delay that these elements must provide, analysis methods described in e.g. [Chak97] can be used. It turns out that in practice, few BMMs really need additional delay elements.

The circuits shown in figures 3.14 and 3.15 are quite complex. It is not so easy to 'see' that they are correct. Therefore verification is important. The hazard–free implementations of all circuits mentioned in this thesis have been verified to be correct. In chapter 5 a more detailed explanation on the topic of verification is provided.

A note on the testability of BMMs. Let us consider only the combinational logic part of a given BMM. Furthermore, let us constrain ourselves to two–level implementations. It is possible that the combinational logic is redundant when considering the logic function implemented by the circuit.

That is, we can remove one or more gates without disturbing the Boolean function implemented by the circuit. Note that the circuit is not redundant from a transition–based point of view. Removing any gate is bound to introduce hazards under the unbounded wire–delay model. However, as was shown in [deM94] it is impossible to find a test vector for a Boolean redundant circuit that is able to test each gate completely. Therefore, in order to obtain 100% testability, it might be necessary that in some BMMs a number of internal signals are made observable. In [Now97] a method is presented that minimizes the number of untestable gates. However, this method also cannot guarantee that the network will be testable without having to make internal signals observable.

3.7 The history of BMMs

BMM–based synthesis was first applied by the section of the HP–laboratories, known a the postoffice team [Coat93]. The goal of the postoffice team was to design a system that supported inter–node communication for a parallel–processing system. Since this system had to be scalable, a global clock would be infeasible. They felt it was necessary to design the system using asynchronous logic. The team initially decided to use SIC (Single Input Change) machines since SIC machines are attractive in the sense of modularity. However, "*The SIC AFSM methodology soon proved to be both a performance bottleneck, and consumed unreasonable amounts of silicon real estate.*" MIC (Multiple Input Change) machines however, would severely undermine the modularity of the design. Therefore, a constrained form of MIC/Huffman machines was developed: the Burst Mode Machines.

The BMM specifications by the postoffice team were rather informal and the implementations generated had to be verified to be hazard–free. Dill and Nowick introduced a formal model for BMMs. They also introduced the requirements of unique entry point and maximal set property in order to guarantee a hazard–free implementation. They also developed the first BMM–synthesis system that was fully automated. The BMMs generated by this system were based on a local–clocking principle [Now91]. The main advantages of this system are that the constraints imposed on the combinational logic are less severe than the constraints imposed on the BMM–synthesis design–style that was introduced in this chapter. Furthermore, no special state assignment procedure is needed. Any state assignment suffices in the case of locally clocked BMMs. A few disadvantages, however, are that this design style requires a number of latches: one for each output, and two for each state variable. Furthermore, the circuitry needed to implement the local clock can be quite significant.

The local clocked BMM design style was developed at a time when a hazard–free logic minimizer was unavailable. The first hazard–free minimizer was also proposed by Nowick and Dill [Now92]. With this minimizer it became possible to design a circuit that would no longer need a local clock.

Yun extended the BMM specifications into the so–called *Extended Burst–Mode Machine (EBMM) specifications*. He added the possibility for a BMM specification to allow conditionals and directed don't–cares [Yun94]. Conditionals allow for different behavior for the same set of input changes, depending on the values of some input variables that remain stable during the input burst. Directed don't–cares specify that a certain input variable might be changed by the environment before the environment has completed one of the possible input bursts.

Yun also proposed, and implemented the 3D synthesis system. This system can synthesize a given EBMM specification from start to end. This requires an adapted state–minimization procedure, since EBMMs can violate the unique entry–point requirement. Yun proposed a bottom–up approach, where new states are introduced to avoid conflicts in the flow table, and to remove otherwise unavoidable hazards. The 3D system proposed by Yun is capable of generating both two–level solutions and multi–level solutions, based on a multiplexer standard–cell architecture. In this thesis we do not deal with EBMM specifications.

3.8 Contributions

In this thesis, we present the following contributions to the field of BMM–based synthesis:

- In chapter 4, we present the threeway method, an efficient algorithm to generate hazard–free two–level logic. We also propose a fast heuristic algorithm. The latter is able to dramatically reduce the runtimes required to synthesize the combinational part of a BMM.
- In chapter 6, we present a multi–level synthesis methodology, based on hazard–free transformations, that can make use of the set of controllability and observability don't–cares.
- In chapter 5, we present a verification method, largely based on Kung's work [Kung92], that can detect all hazards we are interested in. We propose a modification that also allows Kung's method to verify if each transition is implemented such that the Burst–Mode condition is met.
- In chapter 8, we present some improvements to the asynchronous state–assignment methodology proposed by Fuhrer [Fuhr95]. We show that our improvements significantly reduce the number of state variables required, resulting in smaller implementations.

Chapter 4 Hazard–free two–level logic minimization

4.1 Introduction

In chapter 3 we saw that the logic part of a BMM must be implemented free from hazards for a given set of transitions. Normal logic synthesis methods targeted at synchronous logic can in general not be used without some modifications. In this chapter we will focus on the automated synthesis of hazard–free two–level logic. We start with an analysis of the classes of hazards that we have to deal with in the case of two–level logic.

4.2 Hazards in two–level logic

Hazard–free logic synthesis is based on implementing a given set of transitions T free from hazards. We assume that the set of transitions does not conflict internally, as was explained in section 2.7. This will automatically rule out all function hazards. So, we have to deal with logic hazards. As we will see, a function–hazard–free specification is not enough to guarantee a hazard–free implementation in two–level logic. For now, we assume the specification to be implementable. Later on, we will show sufficient conditions in order to guarantee implementability. To facilitate the discussion we introduce a *transition cube* [Now92].

Definition 4.1
A transition cube belonging to a transition is the smallest cube, in the number of minterms, that covers all minterms that can be entered during the transition. Let the starting point be noted by A, and the end point with B, then the transition cube is noted by [A, B].

In this definition, the starting point A of a transition corresponds with the minterm at which the transition starts. Similarly, the end point B corresponds with the minterm at which the transition ends.

Consider transition $\uparrow 1 \downarrow\!\rightarrow 1$. The transition cube that covers all minterms that can be entered is –1–, or simply x_2, if the sequence of the input variables

is $x_1 x_2 x_3$. Transforming the input part of a transition to a transition cube can simply be done by replacing each arrow with a don't care.

For now, we consider single output functions. As we saw in section 2.7, an output can assume four different values, namely: $0, 1, \uparrow$ and $\downarrow$. Therefore there are four possibilities in which hazards can occur. We start by examining the case where the output assumes value 0, meaning that it is expected to remain static at zero. The following two theorems are by Unger [Ung69].

THEOREM 4.1
A two–level logic implementation of a set of transitions is free from hazards during static–0 transitions.

PROOF
If a hazard does occur, it means that at least one cube temporarily turns on. Therefore, for some minterm in the transition cube this cube will turn on. This means that there are two different values specified for the same minterm. This clearly is a function hazard, not a logic hazard. There is one exception. Unger [Ung69] demonstrated that it is possible to get logic hazards during static–0 transitions. This is accomplished by implementing a cube that has in its support set both an input variable and the negated input variable. An example is shown in figure 4.1.

FIGURE 4.1. A static–0 hazard in a two–level implementation

We discard any two–level implementation that contains cubes like the cube depicted in figure 4.1, since these designs can be optimized trivially by just removing these cubes from consideration. □

THEOREM 4.2
A two–level logic implementation of a set of transitions is free from hazards during 0–to–1 transitions.

PROOF
No static–0 hazard can occur, and all cubes that turn on make a monotonous transition (if not, they can be expanded to do so). Therefore, no hazard is possible in this case. □

That leaves only two cases to be considered: hazards during static–1 transitions, and hazards during dynamic 1–to–0 transitions. In both cases

hazards can occur. In figures 4.2a and 4.3a examples of both a static and a dynamic hazard are depicted.

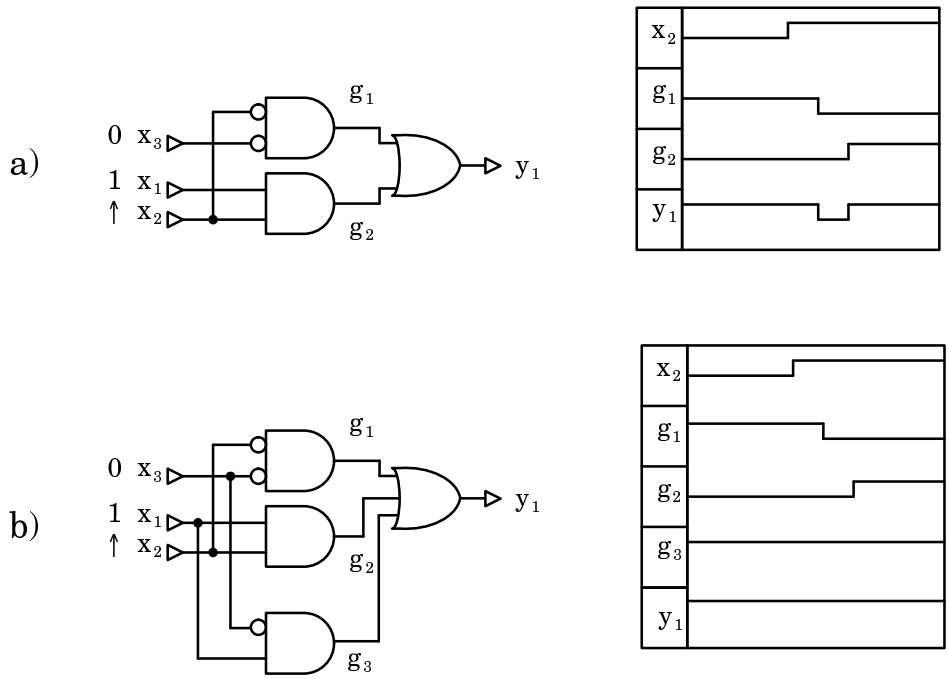


FIGURE 4.2. Removing a static–1 hazard by adding a required cube

We start with examining the case where the output is supposed to remain static at 1. A hazard might occur due to a so–called *term–takeover*. That is, during the transition all cubes that are enabled switch off, while other cubes concurrently turn on. Due to some timing differences, the output might temporarily drop to zero. In order to avoid this hazard from occurring, at least one cube has to remain on during the transition, for all possible combinations of delays on the wires of the circuit (recall that we require that the circuit implements each transition free from hazards under the unbounded wire–delay model). Therefore, the transition cube belonging to this transition has to be covered by at least one implicant in the two–level solution. Such a transition cube is also called a *required cube*.

DEFINITION 4.2

A required cube is a cube necessary to avoid a hazard due to a term–takeover. A required cube is necessary, because in a hazard–free two–level solution, each required cube should be covered by at least one implicant.

Both static–1 and dynamic 1–to–0 transitions (as we will see) introduce a number of required cubes.

Consider again the circuit that exhibits a static hazard in figure 4.2a. This hazard occurs during transition $1 \uparrow 0 \rightarrow 1$. The transition cube belonging to

this transition is $x_1\bar{x}_3$. This transition cube is necessary to avoid the static hazard shown, so the transition cube is also a required cube. A hazard–free implementation of the circuit in figure 4.2a is shown in figure 4.2b. In this implementation the required cube has been added.

In the case of figure 4.3 the output is expected to make a monotonous change from 1 to 0. There are two different causes for a possible hazard. The first one is again the term–takeover. This hazard can, again, be removed by introducing a number of required cubes. In this case, several required cubes might be required, since the transition cube itself cannot be used as a required cube, because the output will be 0 at the end point B. The set of required cubes consist of the maximal cubes that cover the transition cube minus the end–point B. A so–called sharp operation, indicated by #, [deM94] can be used to derive the set of required cubes. The set of required cubes is equal to [A, B]#B.

Consider transition $\uparrow 1 \downarrow 0 \uparrow \rightarrow \downarrow$. The transition cube belonging to this transition is –1–0–. The end point of the transition is 11001. The set of required cubes can be calculated by: –1–0– # 11001 = {01–0–, –110–, –1–00} = $\left\{\bar{x}_1x_2\bar{x}_4, x_2x_3\bar{x}_4, x_2\bar{x}_4\bar{x}_5\right\}$. The transition cube is copied as many times as there are don't cares in the transition cube. In each copy one don't care is replaced by 0/1 if the corresponding literal form in the end point is equal to 1/0.

The second cause of a dynamic hazard is due to the fact that during the transition a cube might temporarily turn on. The glitch produced by this cube might be observed after the output has gone to zero. An example of this behavior is shown in figure 4.3a. Note that the hazard caused by gate g_3 is not a function hazard. This kind of hazard can be removed by making sure that no cube can temporarily turn on during a 1–to–0 transition. This is accomplished by introducing a so–called *privileged cube*.

DEFINITION 4.3
A privileged cube p is a tuple consisting of a transition cube and an additional starting point. Thus, $p = (p^c, p^s) = ([A, B], A)$. Here p^c is called the body of the privileged cube and p^s is called the starting point of the privileged cube.

DEFINITION 4.4
An implicant is said to illegally intersect a privileged cube p if the implicant intersects the body of the privileged cube p^c and the intersection does not include the starting point p^s.

If an implicant illegally intersects a privileged cube p it might temporarily turn on during a dynamic 1–to–0 transition, causing a hazard. An example

of this behavior is shown in figure 4.3a. In this circuit a dynamic hazard can occur during transition $\uparrow 0 \uparrow \rightarrow \downarrow$. This transition introduces two required cubes, namely $\overline{x}_2\overline{x}_3$ and $\overline{x}_1\overline{x}_2$, and one privileged cube $(\overline{x}_2, \overline{x}_1\overline{x}_2\overline{x}_3)$. As we can see, both required cubes are covered by the cubes in the implementation. However gate g_3 illegally intersects the privileged cube. The time diagram of the circuit in figure 4.3a shows that gate g_3 might temporarily turn, causing a hazard at the output. We can remove the illegal intersection by adding variable x_2 to the support set of gate g_3. The resulting hazard–free network is depicted in figure 4.3b.

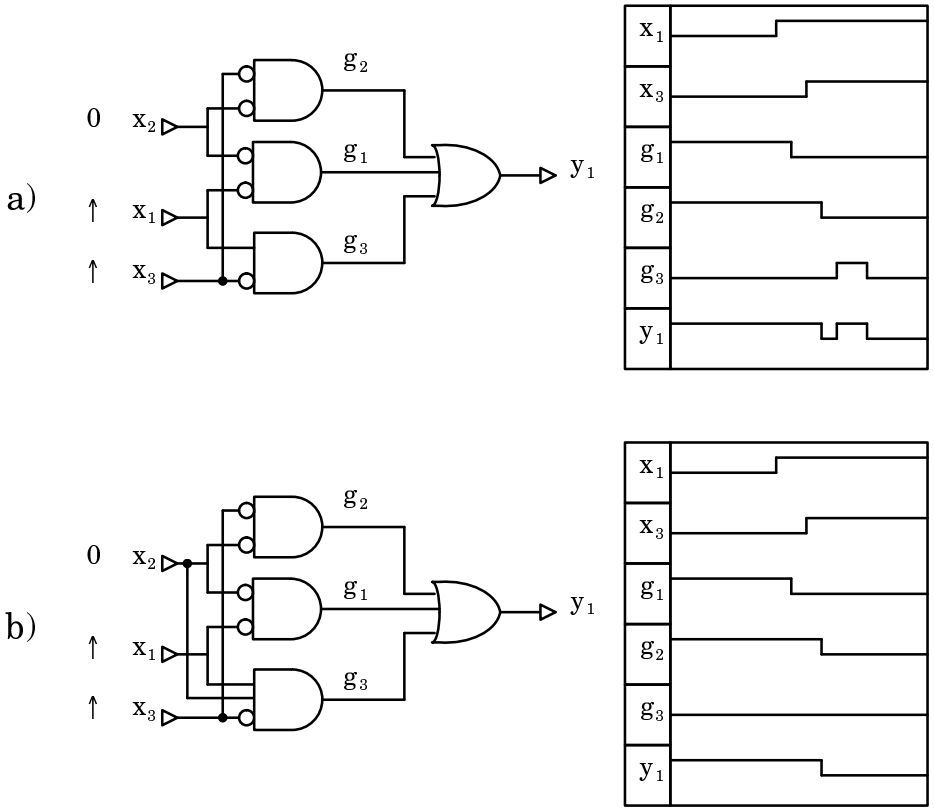


FIGURE 4.3. Removing a dynamic hazard by removing an illegal intersection with a privileged cube

By introducing required and privileged cubes, we can guarantee a hazard–free two–level implementation as long as the specification, in the form of a set of transitions, is *implementable*.

THEOREM 4.3

A set of transitions is guaranteed to be implementable if no required cube in the set of required cubes illegally intersects any privileged cube.

If a required cube illegally intersects a privileged cube, then there is no guarantee that an implicant exists that covers the required cube but doesn't intersect the privileged cube illegally.

In figure 4.4a, a specification, in the form of a set of transitions, of a circuit with 4 inputs and one output is depicted in a Karnaugh map. An empty entry corresponds with a don't–care entry. The transitions are all represented by arrows. The depicted transitions can be captured by a set T: {↑ 000 → 1 , 1 ↑ 0 ↑→↓ , 11 ↑ 1 → 0 , 111 ↓→↑ , 1 ↓ 10 →↓ , ↓ 010 → 0 , 00 ↓ 0 → 0}. The order of the input variables in these transitions is $x_1x_2x_3x_4$. We know that only static–1 transitions and dynamic 1–to–0 transitions can introduce required and privileged cubes. Let us consider transition ↑ 000 → 1. During this transition, input variable x_1 switches on. To avoid a possible static hazard, we have to introduce a required cube $\overline{x}_2\overline{x}_3\overline{x}_4$. Transition 1 ↑ 0 ↑→↓ is correctly implemented if the solution covers two required cubes, namely $x_1\overline{x}_3\overline{x}_4$ and $x_1\overline{x}_2\overline{x}_3$, and if no implicant in the solution illegally intersects privileged cube $(x_1\overline{x}_3, x_1\overline{x}_2\overline{x}_3\overline{x}_4)$. The minimum (in the number of cubes) two–level implementation of the onset implied by the given set of transitions is represented by figure 4.4b. This solution is hazardous for a number of reasons. First of all, no implicant covers required cube $x_1\overline{x}_3\overline{x}_4$. Furthermore implicant $x_2\overline{x}_4$ illegally intersects privileged cube $(x_1\overline{x}_3, x_1\overline{x}_2\overline{x}_3\overline{x}_4)$. This solution will therefore show a hazardous behaviour for transition 1 ↑ 0 ↑→↓. A correct hazard–free solution can be found in figure 4.4c.

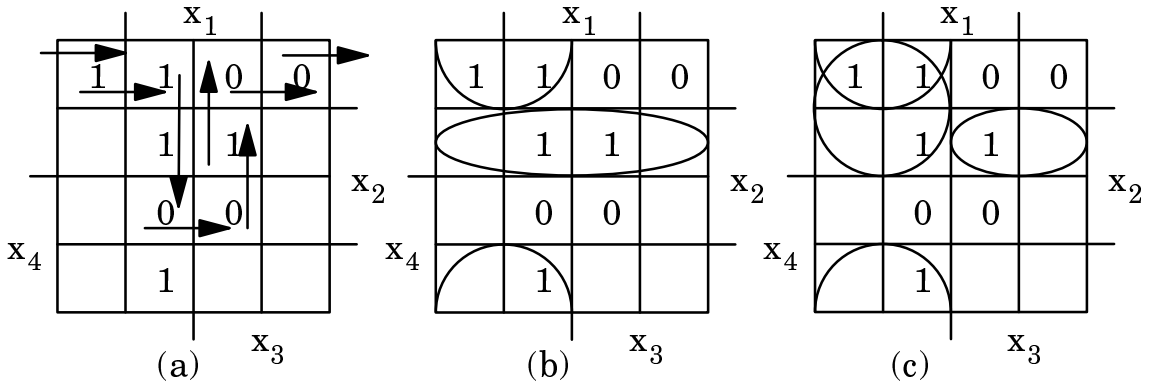


FIGURE 4.4. Example of a hazardous, and a hazard–free implementation of a set of transitions

The theory described sofar covers two–level logic. Until now we used this term to denote sum of cubes, also called sum of products. However, a well known

property of sums of products (SOP) expressions is that they have dual counterparts, called product of sums (POS) expressions.

POS expressions have an interesting property: the hazards that are possible in POS expressions are the dual of the hazards possible in SOP expressions. So, POS expressions can suffer from static–0 hazards during static–0 transitions, and dynamic 0–to–1 hazards during dynamic 0–to–1 transitions, but they are immune to static–1 hazards and dynamic 1–to–0 hazards.

One might be able to use this property in order to implement transitions that are not implementable with SOP expressions. Consider the transitions $\uparrow\uparrow 0 \rightarrow\downarrow$, $10 \downarrow\rightarrow 1$. Let the order of the input variables in these transitions be $x_1x_2x_3$. The second transition introduces required cube $x_1\overline{x}_2$. However, this required cube illegally intersects privileged cube $(\overline{x}_3, \overline{x}_1\overline{x}_2\overline{x}_3)$ introduced by the first transition. Therefore, with SOP expressions, we cannot guarantee a hazard free solution. A POS implementation, however, is possible since POS expressions are immune for hazards during static–1 and dynamic 1–to–0 transitions, and is shown in figure 4.5. In this thesis, when we talk about two–level logic, it will always be SOP expressions, unless stated otherwise.

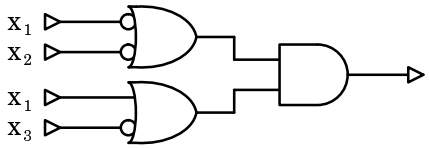


FIGURE 4.5. Hazard–free POS implementation

4.3 Generating hazard–free logic

Nowick [Now92] proposed the first method that is capable of generating a hazard–free solution for a given set of implementable transitions which are free of function hazards. The method uses a normal 'synchronous' minimizer to generate all prime implicants. The method is based on the following theorem.

THEOREM 4.4
All required cubes are covered by the set of all prime implicants.

PROOF
By contradiction: Suppose a required cube is not covered by the set of all prime implicants. Then by simply adding this required cube, we would have added an implicant that is not covered completely by any of the other prime implicants, which implies that we didn't generate all prime implicants. □

Theorem 4.4 shows that it is possible to select a set of prime implicants such that all required cubes are covered. The selected set of prime implicants can however still illegally intersect some, or more privileged cubes. To avoid this, each prime implicant that illegally intersects a privileged cube is sharpened, such that the resulting implicants no longer illegally intersect the privileged cube.

The operation of removing an illegal intersection introduces new implicants. Unfortunately each of these implicants can have an illegal intersection with privileged cubes which were e.g. legally intersected by the original (prime) implicant. Therefore an algorithm that removes illegal intersections is necessarily recursive. An algorithm that removes all illegal intersections of a given set of prime implicants is depicted in algorithm 4.1.

ALGORITHM 4.1. Algorithm for removing all illegal intersections

```
Primes MakeDHF(PrivCubes Priv, Primes In)
{
   Primes dhf := ⊘;
   Primes nondhf := ⊘;

   if (In = ⊘) return ⊘;
   Foreach(i ∈ In, p ∈ Priv)
   {
      if ((p^c · i ≠ 0)  ∧  (p^s · i = 0))
         nondhf := nondhf ∪ i#p^c;
      else
         dhf := dhf ∪ i;
   }
   dhf := SCC(dhf ∪ MakeDHF(Priv, nondhf));
   return(dhf);
}
```

In algorithm 4.1, Priv represents the set of privileged cubes and In represents the set of prime implicants. The algorithm returns a set of dhf– (dynamic hazard free) prime implicants. The SCC operator is a single cube containment operator. A minimum two–level implementation can be obtained by making a minimum selection of these dhf–prime implicants such that each required cube is covered by at least one dhf–prime implicant.

Although relatively simple, the depicted algorithm has some severe limitations. In order to be able to guarantee a solution consisting of dhf–prime implicants, the algorithm has to be applied to the set of all prime implicants.

The problem of generating the set of all prime implicants has an exponential complexity, and it can therefore only be performed for small examples. Furthermore, the algorithm in algorithm 4.1 itself is also expensive. It can introduce many new dhf–prime implicants, and it might take many recursions to make sure that sharpened implicants no longer intersect any privileged cube illegally. So both the process of generating all prime implicants, and the proposed algorithm can be a bottleneck in the generation of a valid two–level solution.

In the next sections we propose a method that overcomes these two bottlenecks. We show how we can avoid the use of algorithm 4.1 altogether, by avoiding the generation of hazardous prime implicants during the process of generating all prime implicants.

A well known divide & conquer algorithm that generates all prime implicants is based on the so–called unate recursive paradigm, described in chapter 2. It makes sense to try to modify this algorithm to generate only dhf–prime implicants. That way no a–posteriori filtering operation is needed. Unfortunately this is not easy to do. In section 4.4 we propose a new divide & conquer algorithm, called the *threeway* method [Rut97b][Rut98a][Rut98c], which is based on Coudert's work [Coud94]. In section 4.4.2 we show that the threeway method is capable of generating all dhf–prime implicants *implicitly* (The latter term should not be confused with the notion of using implicit data structures like BDDs [Hach96]). By doing so, we can avoid the a–posteriori filtering operation described by algorithm 4.1. Furthermore, in section 4.4.3 we show that the threeway method can be modified to only generate those dhf–prime implicants that can possibly contribute to a solution. We call these dhf–prime implicants the set of *contributing/useful dhf–prime implicants*. We show that by only generating the set of contributing dhf–prime implicants, the runtimes are significantly reduced, thereby reducing the bottleneck of generating the set of all prime implicants.

Section 4.4 only discusses the threeway method for single output functions. In section 4.5 we discuss multi–valued extensions to the threeway method, which allow us to represent multiple output functions. Although the threeway method is capable of significantly reducing the two mentioned bottlenecks, one bottleneck does remain: from the set of generated dhf–prime implicants a minimum cover must be selected that covers the set of required cubes. This problem, known under the name unate covering, is a well–known NP–complete problem [Gar79, Problem SP5, Minimum Cover].

In section 4.6 we show how the threeway method can be used to develop heuristic methods, by proposing one heuristic method. Both sections 4.5 and 4.6 can be skipped safely by first–time readers. Finally, we discuss our work in relation to the work of other people.

4.4 The threeway method for single output functions

4.4.1 Introduction

The unate recursive paradigm, described in section 2.3, calculates the set of all prime implicants by splitting the original set of cubes into two (smaller) sets. With the help of recursion, and later on, by combining the calculated sub–solutions, the set of all prime implicants can be calculated. Here, we use another approach, which is called the threeway method. The threeway method partitions the problem of generating the set of all prime implicants into the three smaller sub–problems of finding the set of prime implicants in which the literal form of a selected input variable x is fixed to one of the three possible forms: 0, 1 and don't care. In formula this partitioning can be expressed as:

$$\text{Primes}(F) = \text{ONEPrimes}(F) \cup \text{ZEROPrimes}(F) \cup \text{DCPrimes}(F) \qquad (4.1)$$

In this formula, ONEPrimes(F) generates the set of prime implicants in which the literal form of the selected input variable x is fixed to 1 for each implicant. Similarly, for each prime implicant generated by ZEROPrimes(F), the literal form of x is fixed to 0, and in each prime implicant generated by DCPrimes(F) the literal form of x is fixed to a don't care. Like the unate recursive paradigm in formula 2.2, we want to cast formula 4.1 into a formula that is completely expressed in operator Primes. By doing so, each sub–problem itself can also be reduced into three even smaller sub–problems. This process can continue until the sub–problems become trivial.

We first examine the primes generated by ONEPrimes(F). Consider a prime implicant $c \in \text{ONEPrimes}(F)$. Since $c \Rightarrow x \cdot F$, the set of primes in ONEPrimes(F) are contained by the set of implicants generated by $\text{Primes}(x \cdot F)$. The latter set will however also contain implicants which are covered by DCPrimes(F), and which therefore are not prime. As we will see, these implicants can be removed with the help of a SCC operation. Remember that a SCC operation is a Single Cube Containment operation. The set of prime implicants $\text{Primes}(x \cdot F)$ can be rewritten as:

$$\text{Primes}(x \cdot F) = x \odot \text{Primes}(F|_x) \qquad (4.2)$$

For a definition of operator $\odot$, we refer to chapter 2. A similar expression can be derived for set ZEROPrimes(F).

In the set of prime implicants contained by DCPrimes(F) the literal form of x is a don't care. So, DCPrimes(F) operates on that part of the onset of F that does not depend on x. An expression for this onset can be derived by taking

the Boolean product of $F(..., x, ...)$ and $F(..., \overline{x}, ...)$. With help of a Shannon expansion this can be rewritten as:

$$F(..., x, ...) \cdot F(..., \overline{x}, ...) = (x \cdot F|_x + \overline{x} \cdot F|_{\overline{x}}) \cdot (x \cdot F|_{\overline{x}} + \overline{x} \cdot F|_x) = F|_x \cdot F|_{\overline{x}} \quad (4.3)$$

So, DCPrimes(F) can be expressed as:

$$\text{DCPrimes}(F) = \text{Primes}(F|_x \cdot F|_{\overline{x}}) \quad (4.4)$$

The threeway method in formula 4.1 can be written as:

$$\text{Primes}(F) = \text{SCC}(x \odot \text{Primes}(F|_x) \cup \overline{x} \odot \text{Primes}(F|_{\overline{x}}) \cup \text{Primes}(F|_x \cdot F|_{\overline{x}})) \quad (4.5)$$

In this formula, each sub–problem can also be divided into even smaller sub–problems. This process can be applied recursively, until the problem of finding the set of all prime implicants becomes trivial. In the case that all functions are described with two–level expressions, a problem becomes trivial when the two–level expression of the function is unate in all variables, since the set of implicants in that expression is in that case equal to the set of prime implicants, as was argued in section 2.3.

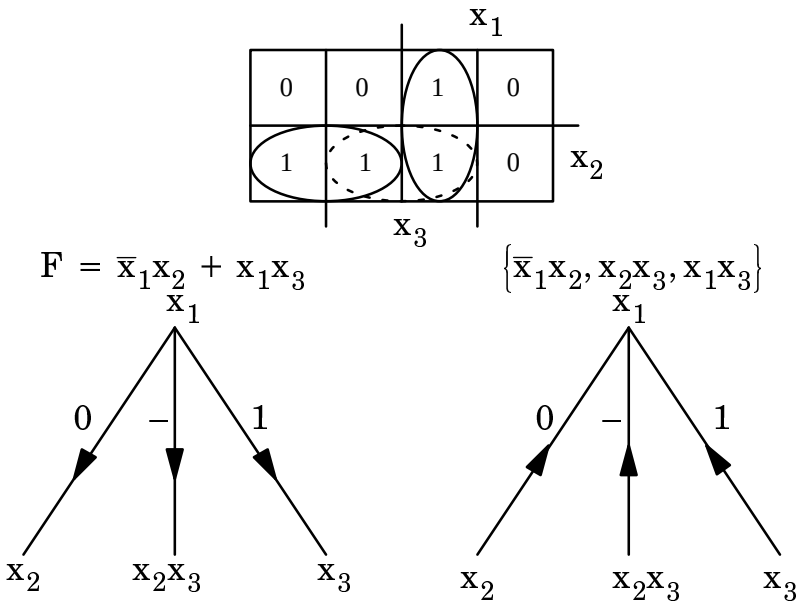


FIGURE 4.6. Example of the threeway method

In figure 4.6 an example of a specification is shown. Since F is not unate in all variables, we have to apply the described threeway method. The threeway

method itself can be represented as a tree, shown on the left in figure 4.6, where each edge corresponds with one of the three sub–problems. Observe that the three sub–problems are indeed unate in all variables. Therefore, we can collect the depicted implicants, and form the set of prime implicants. The process of collecting all prime implicants can also be represented by a tree, shown on the right in figure 4.6.

4.4.2 Generating all dhf–prime implicants with the threeway method

The threeway method introduced in the previous section can be adapted to generate the set of all dhf–prime implicants. This can be accomplished by taking into account the set of all privileged cubes P. We start, again, by partitioning the problem of generating all, in this case, dhf–prime implicants into three sub–problems:

$$\begin{aligned}\text{DHFPrimes}(F,P) = {} & \text{DHFONEPrimes}(F,P) \cup \text{DHFZEROPrimes}(F,P) \cup \\ & \text{DHFDCPrimes}(F,P)\end{aligned} \tag{4.6}$$

The first term, DHFONEPrimes(F, P), will only generate dhf–prime implicants in which the literal form of the selected input variable x is equal to 1. We already saw that in order to generate implicants in which the literal form of x is fixed to 1, we only need to take into account part of the onset, namely $x \cdot F$, since each prime implicant $c \in$DHFONEPrimes(F,P) implies this part of the onset, i.e. $c \Rightarrow x \cdot F$.

Each prime c can be calculated based on the onset $x \cdot F$ and on the set of privileged cubes, P. The latter set is necessary to make sure that each prime c does not illegally intersect any privileged cube $p \in P$. Based on the onset, $x \cdot F$, we can partition the set of privileged cubes P into two sets, Pi and Pp. The first set, Pi does contain all privileged cubes that can never be intersected legally by any prime implicant that implies $x \cdot F$. This set can be expressed as:

$$Pi = \{p \mid p \in P \wedge x \cdot p^s = 0\} \tag{4.7}$$

Set Pi will contain all privileged cubes that have a starting point that cannot be covered by x, and therefore also not by $x \cdot F$. Since the privileged cubes in set Pi can never be intersected legally by any prime c implying $x \cdot F$, we might as well make it impossible for any prime implying $x \cdot F$ to intersect any privileged cube $p \in$ Pi. This can be guaranteed by removing all the bodies of the privileged cubes belonging to set Pi from onset $x \cdot F$. The modified onset can be expressed as:

$$x \cdot F - x \cdot Pi^c \tag{4.8}$$

In this formula Boolean function Pi^c is expressed as:

$$Pi^c = \sum_{p \in Pi} p^c$$

Here, p^c is the body of a privileged cube belonging to set Pi.

Apparently, in generating the set of hazard–free prime implicants DHFONEPrimes(F, P), we only need to consider the onset expressed in formula 4.8, while making sure that the primes generated do not illegally intersect a privileged cube belonging to set Pp. This can be expressed as:

$$\text{DHFPrimes}(x \cdot F - x \cdot Pi^c, Pp) \tag{4.9}$$

Set Pp simply is the set of privileged cubes P minus the set of privileged cubes in set Pi. Since in all the prime implicants in DHFONEPrimes(F, P) variable x has a literal form equal to 1, we might as well make sure that the literal form of variable x in all privileged cubes in set Pp is also equal to 1. This can be accomplished by calculating $x \odot Pp$.

Here, operator $\odot$ operates on a cube and a set of privileged cubes, and is defined as: $x \odot P = \{(x \cdot p^c, x \cdot p^s) \mid (p^c, p^s) \in P\}$. Privileged cubes for which either the body p^c or the starting point p^s is equal to 0 are removed from the set. Since $x \odot Pp$ is equal to $x \odot P$, we can express formula 4.9 as:

$$\text{DHFPrimes}(x \cdot F - x \cdot Pi^c, x \odot P) \tag{4.10}$$

This formula can be rewritten as:

$$x \odot \text{DHFPrimes}(F|_x - Pi^c|_x, P|_x) \tag{4.11}$$

The cofactor operator on the set of privileged cubes is defined as: $P|_x = \{(p^c|_x, p^s|_x) \mid (p^c, p^s) \in P\}$. Again, privileged cubes for which either the body p^c or the starting point p^s is equal to 0 are removed from the set. Of course, a similar derivation exists for expression DHFZEROPrimes(F, P).

In calculating the set of dhf–prime implicants contained by DHFDCPrimes(F, P), we can no longer use the literal form of variable x to determine which subset of privileged cubes will always be intersected illegally, as we did in the case for DHFONEPrimes(F, P) and DHFZEROPrimes(F, P), because DHFDCPrimes(F, P) generates dhf–prime implicants in which the literal form of variable x is fixed to a don't care. Therefore we do not modify the onset, resulting in the following expression for DHFDCPrimes(F, P).

$$\text{DHFDCPrimes}(F, P) = \text{DHFPrimes}(F|_x \cdot F|_{\overline{x}}, P') \tag{4.12}$$

As a first approach $P' = P$. Since we cannot reduce the set of privileged cubes, set P' has to take into account the complete set of privileged cubes. We can partition the set of privileged cubes, P into two sets, P_0 and P_1. Each privileged cube in set P_0 has a starting point for which the literal form of variable x is equal to 0. The privileged cubes in set P_1 have a starting point for which the literal form of variable x is equal to 1. Since we assume that the starting point of a privileged cube is a minterm, privileged cubes that have a starting point for which the literal form of variable x is fixed to a don't care do not exist.

Consider a privileged cube $p_1 \in P_1$. In all implicants generated by DHFDCPrimes(F, P) the literal form of variable x is fixed to a don't care. So, if an implicant illegally intersects privileged cube p_1, then it also illegally intersects privileged cube $p_1(..., x = 1, ...) = p_1|_x$, and viceversa. A similar observation is possible for the set of privileged cubes, P_0, resulting in the following expression for set P':

$$P' = P_1|_x \cup P_0|_{\overline{x}} = P|_x \cup P|_{\overline{x}} \tag{4.13}$$

The set of hazard–free prime implicants, generated by DHFDCPrimes(F, P) can therefore be calculated by the following formula:

$$\text{DHFPrimes}(F|_x \cdot F|_{\overline{x}}, P|_x \cup P|_{\overline{x}}) \tag{4.14}$$

The complete threeway method that is capable of generating all dhf–prime implicants can be written as:

$$\begin{aligned} \text{DHFPrimes}(F, P) = \text{SCC}(&x \odot \text{DHFPrimes}(F|_x - Pi^c|_x, P|_x) \cup \\ &\overline{x} \odot \text{DHFPrimes}(F|_{\overline{x}} - Pi^c|_{\overline{x}}, P|_{\overline{x}}) \cup \\ &\text{DHFPrimes}(F|_x \cdot F|_{\overline{x}}, P|_x \cup P|_{\overline{x}})) \end{aligned} \tag{4.15}$$

Again, each sub–problem can be divided into smaller sub–problems. In the case that all functions are represented with two–level expression, this process can be repeated until these expressions are unate in all variables *and* set P is empty.

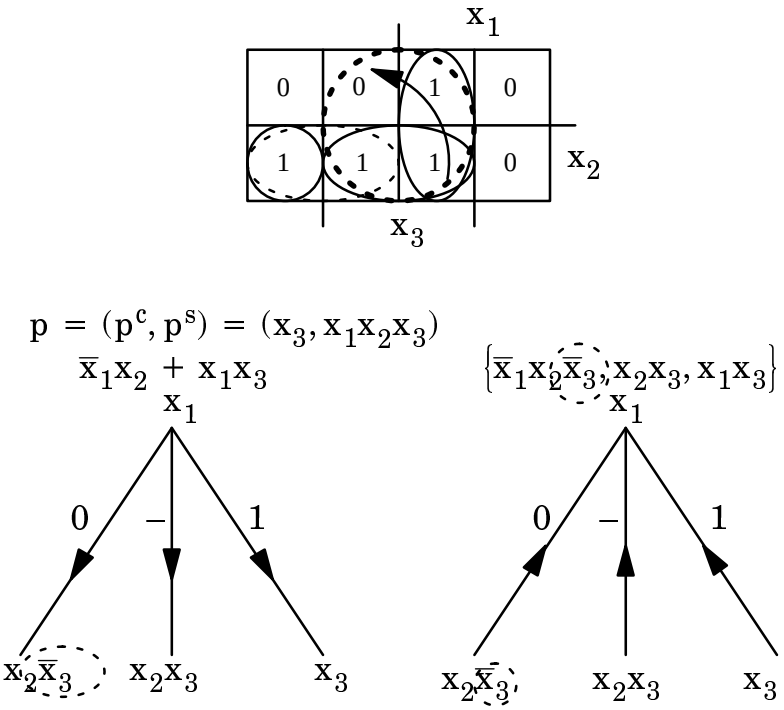


FIGURE 4.7. Example of the dhf–threeway method

An example is given in figure 4.7. In this figure the same specification is shown as in figure 4.6. We have added a transition $\downarrow\downarrow 1 \rightarrow\downarrow$ to this specification. This transition requires the introduction of a privileged cube $(x_3, x_1x_2x_3)$, which is shown in figure 4.7 as a dotted circle. The 0–edge of the tree on the left generates the set of dhf–prime implicants for which input variable x_1 is fixed to 0. These prime implicants can never legally intersect privileged cube $(x_3, x_1x_2x_3)$. Therefore the body of this privileged cube is removed from the onset. The original onset belonging to the 0–edge, x_2, is sharpened to $x_2\bar{x}_3$. Since all the sub–problems in figure 4.7 are unate in all variables, and the set of privileged cubes is empty, the recursion can be stopped, and the depicted implicants can be collected to form a set of dhf–prime implicants. The latter process is shown in the tree on the right.

4.4.3 Pruning the number of dhf–prime implicants

The dhf–threeway method is capable of generating all dhf–prime implicants. However, many of these dhf–primes will never contribute to a valid solution because they only cover the don't–care set or because they only partially cover some required cubes. The threeway method can be modified to avoid the generation of these prime implicants. This is done by also keeping track of the set of required cubes. For each sub–problem, we can determine the set of

required cubes each of which cannot be covered by any prime implicant generated by this sub–problem. This process is similar to determining, for each sub–problem, the set of privileged cubes that can never be intersected legally. These required cubes can then be dropped from consideration. If this will result in an empty set of required cubes, we can avoid the generation of the set of prime implicants altogether, since these prime implicants can not contribute to a minimum solution. This way, we will only generate the set of so–called *contributing implicants*, i.e. implicants that cover at least one required cube completely. These observations allow us to adjust the threeway method to also take into account the set of required cubes. The threeway method modified to take into account the set of required cubes can be written as follows:

$$\begin{aligned} \mathrm{DHFPrimes}(F, P, Q) = {} & \mathrm{DHFONEPrimes}(F, P, Q) \cup \\ & \mathrm{DHFZEROPrimes}(F, P, Q) \cup \\ & \mathrm{DHFPrimes}(F, P, Q) \end{aligned} \tag{4.16}$$

In this formula Q represents the set of required cubes. After some derivations similar to those applied in the previous sections, the set of dhf–prime implicants generated by the first term are covered by those generated by:

$$x \odot \mathrm{DHFPrimes}(F|_x - Pi^c|_x, P|_x, Qa|_x) \tag{4.17}$$

Where,

$$Qa|_x = \sum_{q \in Q \,:\, x \cdot q = q} q|_x \tag{4.18}$$

A required cube q can only be covered by a dhf–prime implicant generated by DHFONEPrimes, if the literal form of variable x in this required cube is equal to 1. All other required cubes can be discarded from consideration. If this observation will result in an empty set of required cubes, we can skip calculating the set of dhf–prime implicants generated by formula 4.17.

An expression for DHFZEROPrimes(F, P, Q) is derived similarly. Finally, the expression for DHFDCPrimes(F, P, Q) can be expressed as:

$$\mathrm{DHFPrimes}(F|_x \cdot F|_{\overline{x}}, P|_x \cup P|_{\overline{x}}, Q|_x \cup Q|_{\overline{x}}) \tag{4.19}$$

For DHFDCPrimes(F, P, Q) it is not possible to determine which required cubes will never be covered completely, just by considering the literal form of variable x. Therefore all required cubes are taken into account.

The complete dhf–threeway method, that is capable of generating the set of contributing dhf–primes exclusively can be described by the following formula:

$$\begin{aligned}\text{DHFPrimes}(F,P,Q) = \text{SCC}(&x \odot \text{DHFPrimes}(F|_x - \text{Pi}^c|_x, P|_x, \text{Qa}|_x) \cup \\ &\overline{x} \odot \text{DHFPrimes}(F|_{\overline{x}} - \text{Pi}^c|_{\overline{x}}, P|_{\overline{x}}, \text{Qa}|_{\overline{x}}) \cup \\ &\text{DHFPrimes}(F|_x \cdot F|_{\overline{x}}, P|_x \cup P|_{\overline{x}}, Q|_x \cup Q|_{\overline{x}}))\end{aligned} \tag{4.20}$$

Example:

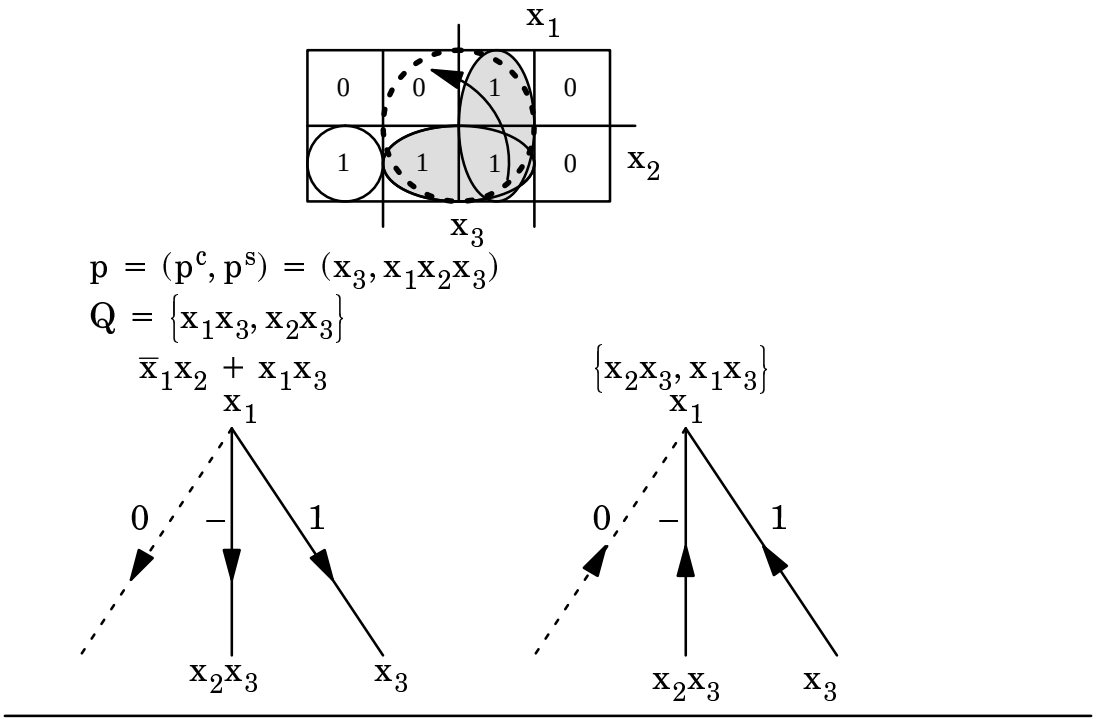


FIGURE 4.8. Pruning the number of dhf–primes

In figure 4.8, we have taken the specification of figure 4.7. We have added two required cubes to this specification, namely x_2x_3 and x_1x_3, they are depicted as dark ellipsoids in figure 4.8. If we again consider the 0–edge of the tree on the left, we now notice that the dhf–primes generated by this edge can never cover any of the required cubes completely. Therefore we might as well skip this edge, which will lead to a reduced number of dhf–prime implicants, and to a reduction in runtime. The tree on the right again shows the collection process of the prime implicants generated.

4.4.4 Explicit representation of implicants

The threeway methods described in the previous sections do not map well on implementations that use an explicit representation to represent all sets of

cubes. In such a representation, a set of cubes is represented by e.g. a list. The reason for this is that both the consensus operation: $F|_x \cdot F|_{\overline{x}}$ and the sharp operation are very expensive operations to perform on explicit sets of cubes. As was shown in [Coud94], the consensus operation can be performed very efficiently if the sets of cubes are represented implicitly, by means of e.g. ZBDDs. However, in our current implementation we have chosen for an explicit representation of sets of cubes.

All threeway methods become more interesting if we start using the offset R of a given function F. With the help of deMorgan, we can rewrite the threeway method of section 4.4.1 as:

$$\text{Primes}(R) = \text{SCC}(x \odot \text{Primes}(R|_x) \cup \overline{x} \odot \text{Primes}(R|_{\overline{x}}) \cup \text{Primes}(R|_x + R|_{\overline{x}})) \tag{4.21}$$

Here, the expensive consensus operation has been replaced by the operation that takes the union of two offsets, which is much cheaper. The recursion is now terminated (the sub–problems become trivial) in two cases: when the offset is empty, or when it is equal to a tautology. In the former case, the cube that represents a tautology is returned. In the latter case the empty set is returned. Note that Primes(R) does *not* calculate the set of prime implicants belonging to the offset R. Instead, operator Primes(R) as defined in formula 4.21 calculates the set of prime implicants belonging to the onset F.

Similarly, we can use the offset to rewrite the threeway method for generating the set of all dhf–prime implicants to become:

$$\begin{aligned} \text{DHFPrimes}(R, P) = \ & \text{SCC}(x \odot \text{DHFPrimes}(R|_x + Pi^c|_x, P|_x) \cup \\ & \overline{x} \odot \text{DHFPrimes}(R|_{\overline{x}} + Pi^c|_{\overline{x}}, P|_{\overline{x}}) \cup \\ & \text{DHFPrimes}(R|_x + R|_{\overline{x}}, P|_x \cup P|_{\overline{x}})) \end{aligned} \tag{4.22}$$

In formula 4.15, we use a sharp operation to remove the body of a privileged cube from the onset, if it is clear that the privileged cube under consideration can never be intersected legally. Removing a cube from the onset is equivalent to adding the same cube to the offset. This observation converts an expensive sharp operation into an operation that simply adds a cube to a list of cubes. The threeway method described by formula 4.21 is converted to a dhf–threeway method by simply augmenting the offset at the appropriate places during the recursion.

Basing the threeway method of section 4.4.3 on the offset representation of a function results in the following expression:

$$\begin{aligned} \text{DHFPrimes}(R, P, Q) = \ & \text{SCC}(x \odot \text{DHFPrimes}(R|_x + Pi^c|_x, P|_x, Qa|_x) \cup \\ & \overline{x} \odot \text{DHFPrimes}(R|_{\overline{x}} + Pi^c|_{\overline{x}}, P|_{\overline{x}}, Qa|_{\overline{x}}) \cup \\ & \text{DHFPrimes}(R|_x + R|_{\overline{x}}, P|_x \cup P|_{\overline{x}}, Q|_x \cup Q|_{\overline{x}})) \end{aligned} \tag{4.23}$$

4.5 The threeway method for multiple–output functions

4.5.1 Introduction

In section 2.4, it was argued that a multiple–output function can be represented with the help of a multi–valued function by mapping the output variables onto one multi–valued variable. That is, the multiple–output function is represented by a function that contains a multi–valued variable. Also, the set of privileged cubes, and the set of required cubes, that ensure a hazard–free operation of the multiple output function for a given set of transitions, can both contain multi–valued variables in order to represent multiple outputs., i.e. to express for which outputs these cubes have a meaning. So, it makes sense to incorporate the ability in the threeway method to deal with multi–valued functions. We will first show how this extension can be incorporated into the framework of the threeway method. Then we will show how the multi–valued threeway method can be used to generate the set of dhf–prime implicants.

4.5.2 The threeway method for multi–valued functions

The binary threeway method can be extended to deal with multi–valued functions. It is tempting to partition the problem of generating all prime implicants into $2^{|x_i|}$ sub–problems, which corresponds to all the literal forms that can be assumed by variable x_i, where x_i itself is a multi–valued variable that is selected in order to partition the problem. This way the problem is partitioned into each possible literal.

It is not difficult to see that in this approach the number of sub–problems simply explodes, therefore this approach is not very interesting. We would again like to partition the original problem into three sub–problems at the most. This is possible by partitioning the set of values that can be assumed by a selected multi–valued variable, x_i, into two sets, α and β. The problem of generating the set of prime implicants is now again partitioned into three sub–problems, based on this variable x_i. The first sub–problem addresses the generation of the set of prime implicants in which each literal form of variable x_i is a subset of α. Similarly, the second problem addresses the generation of prime implicants in which each literal form of variable x_i is a subset of β. The last sub–problem generates the set of prime implicants in which the literal form of variable x_i is a subset of both α and β. In formula this can be expressed as:

$$\text{Primes}(F) = \text{AlphaPrimes}(F) \cup \text{BetaPrimes}(F) \cup \text{AlphaBetaPrimes}(F) \tag{4.24}$$

The set of prime implicants generated by AlphaPrimes(F) is covered by the set of prime implicants generated by Primes($x_i^{\alpha} \cdot F$). It can be proven that Primes($x_i^{\alpha} \cdot F$) = $x_i^{\alpha} \odot$ Primes($F|_{\alpha}$). Therefore, the primes generated by AlphaPrimes(F) are contained by the set of prime implicants belonging to function $x_i^{\alpha} \odot$ Primes($F|_{\alpha}$). A similar derivation is possible for BetaPrimes(F).

In the prime implicants generated by AlphaBetaPrimes(F), the literal form of variable x_i has a non–empty intersection with both α and β. So, we must avoid the generation of prime implicants in which the literal form of x_i only contains values of e.g. set α.

We avoid the generation of these prime implicants, by introducing a new function F′ in which variable x_i is split into two new multi–valued variables, x_{i1} and x_{i2}, where $x_{i1} \in \alpha$ and $x_{i2} \in \beta$. Each prime implicant belonging to function F′ must have valid literal forms (meaning that an all–zero literal is not a valid literal) for both variables x_{i1} and x_{i2}. Since the literal forms of x_{i1} and x_{i2} combined together form the literal form of x_i, the latter literal must cover elements of both sets α and β.

F′ has a changed support set compared with function F. Let the changed support set be denoted by $\vec{x}_+$, where $\vec{x}_+ = x_1, ..., x_{i1}, x_{i2}, ..., x_n$. Consider two input vectors $\vec{x}'$ and $\vec{x}''$ that are equivalent except for input variables x_i' and x_i'', where $x_i' = x_{i1} \in \alpha$ and $x_i'' = x_{i2} \in \beta$. Function F′ must evaluate to 1 if function F evaluates to one for both $\vec{x}'$ and $\vec{x}''$. Based on this observation function F′ can be expressed as:

$$F'(\vec{x}_+) = F(..., x_i = x_{i1}, ...) \cdot F(..., x_i = x_{i2}, ...) \tag{4.25}$$

Since x_{i1} can only assume values in set α and since x_{i2} can only assume values in set β, this formula can be rewritten into:

$$F'(\vec{x}_+) = F|_{\alpha}(..., x_i = x_{i1}, ...) \cdot F|_{\beta}(..., x_i = x_{i2}, ...) \tag{4.26}$$

Therefore:

$$\begin{aligned} &\text{AlphaBetaPrimes}(F) = \text{Primes}(F'(\vec{x}_+)) = \\ &\text{Primes}(F|_{\alpha}(..., x_i = x_{i1}, ...) \cdot F|_{\beta}(..., x_i = x_{i2}, ...)) \end{aligned} \tag{4.27}$$

Note that there is an implicit conversion in this formula. The prime implicants belonging to function F′ are converted to prime implicants belonging to function F. This is done by concatenating the literal forms of variables x_{i1} and x_{i2} into a literal form describing variable x_i.

The complete threeway method, for multi–valued functions, can be written as:

$$\text{Primes}(F) = \text{SCC}(x_i^{\alpha} \odot \text{Primes}(F|_{\alpha}) \cup x_i^{\beta} \odot \text{Primes}(F|_{\beta}) \cup \text{Primes}(F|_{\alpha} \cdot F|_{\beta})) \quad (4.28)$$

Note that for readability reasons, we have chosen to write the last term as: $\text{Primes}(F|_{\alpha} \cdot F|_{\beta})$, instead of the much longer form used in formula 4.27.

As we can see, formula 4.28 and formula 4.5 are very similar. However, we have to keep in mind the important difference. As we saw, for multi–valued functions, the support set of the function belonging to the last term in formula 4.28 is different. This is not the case in the binary threeway method. The binary threeway method is a special case, formed by $|x_i| = 2$. Both $|x_{i1}|$ and $|x_{i2}|$ are 1, meaning that the literal form of both x_{i1} and x_{i2} must be equal to 1, since a 0–literal form of either x_{i1} or x_{i2} will result in invalid implicants.

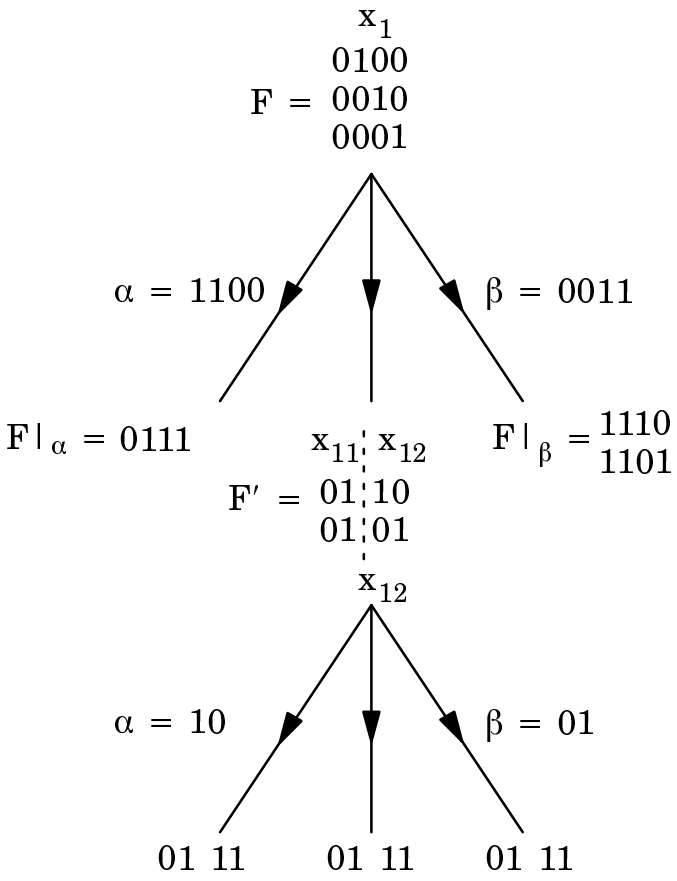


FIGURE 4.9. Applying the threeway method to multi–valued functions

In figure 4.9 an example is given of a multi–valued function consisting of a single variable x_1 that can assume four values. In this example we use a bitvector to represent the literal forms of x_1. The specification of function F in figure 4.9 states that the output will assume value 1 for three distinct values of the multi–valued variable. By intuition it is clear that the literals corresponding with these values can be combined into one literal, namely: 0111. The multi–valued threeway method is able to find this literal. First, the values that can be assumed by variable x_1 is partitioned into two sets α and β. Based on α and β, $F|_\alpha$, $F|_\beta$ and $F|_\alpha \cdot F|_\beta$ are calculated. Function $F|_\alpha \cdot F|_\beta$ consists of two multi–valued variables, x_{11} and x_{12}, where x_{11} can assume $|\alpha|$ values and x_{12} can assume $|\beta|$ values. Function $F|_\alpha \cdot F|_\beta$ is unate in variable x_{11}, however, it is not unate in variable x_{12}. Therefore we choose variable x_{12} to split the problem into three new sub–problems. Again, we partition the values of the selected variable into two sets α and β and we calculate $F|_\alpha$, $F|_\beta$ and $F|_\alpha \cdot F|_\beta$. The resulting implicants are all unate. This is, of course, obvious since each edge in figure 4.9 bears only one implicant.

The prime implicant belonging to function F′ can be described by: $x_{11}^{\{2\}}x_{12}^{\{1,2\}}$ (bitvector 01 11). This prime implicant is converted to a prime implicant belonging to function F by concatenation of the literal forms of variables x_{11} and x_{12}, which results in $x_1^{\{2,3,4\}}$ (bitvector 0111), which is the prime implicant we are looking for.

4.5.3 Generating all multiple–output prime implicants

As we saw in chapter 2, a multiple–output description is specified as a list of cubes, where each cube implies a number of outputs. These descriptions can be trivially converted to a multi–valued description, where one multi–valued variable is used to represent all outputs. Consider the following specification:

x_1x_3	01
$\bar{x}_1x_2$	11
$\bar{x}_1$	10
x_3	10

In this specification we can observe three inputs and two outputs. By considering each bitvector, describing the outputs that are implied, as a bitvector describing a multi–valued variable, x_4, the above given specification becomes: $F = x_1x_3x_4^{\{2\}} + \bar{x}_1x_2x_4^{\{1,2\}} + \bar{x}_1x_4^{\{1\}} + x_3x_4^{\{1\}}$. Applying the multi–valued threeway method, discussed in the previous section, we can

generate the set of prime implicants: $\{x_1x_3x_4^{\{1,2\}}, \bar{x}_1x_2x_4^{\{1,2\}}, \bar{x}_1x_4^{\{1\}}, x_3x_4^{\{1\}}, x_2x_3x_4^{\{2\}}\}$, which corresponds to the following prime implicants for the multiple–output description:

x_1x_3	11
$\bar{x}_1x_2$	11
$\bar{x}_1$	10
x_3	10
x_2x_3	11

In figure 4.10, this process is shown in detail.

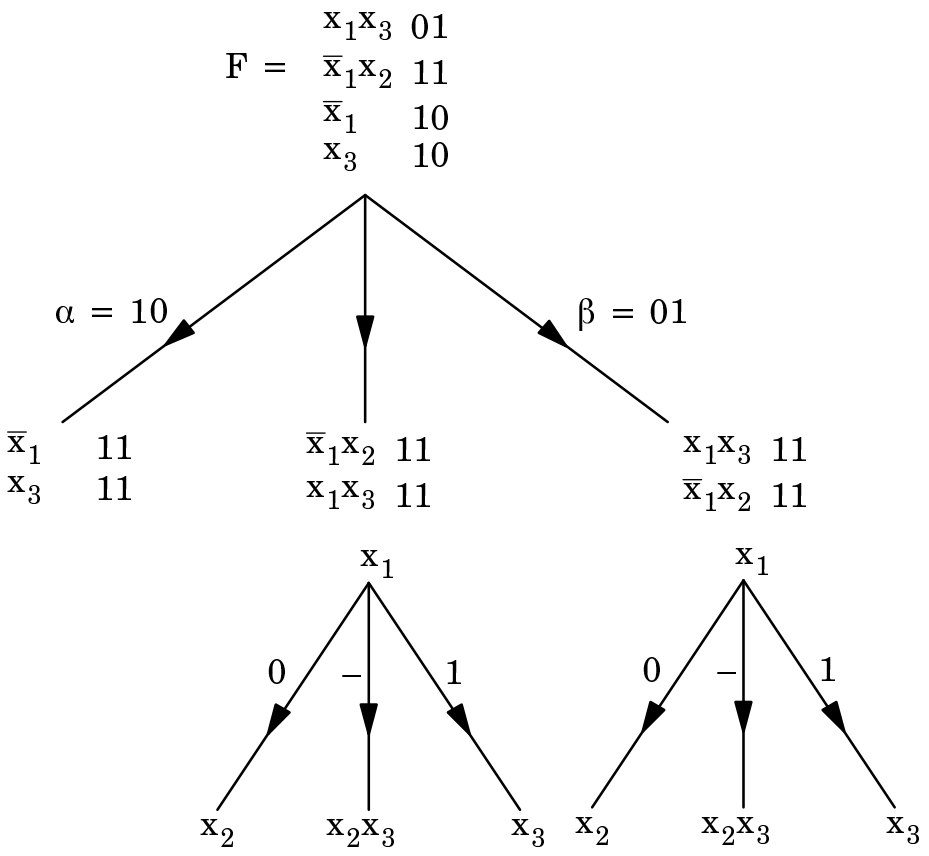


FIGURE 4.10. Using the multi–valued threeway method to generate multiple–output prime implicants

4.5.4 Generating all multiple–output dhf–prime implicants

In order to generate the set of all dhf–prime implicants, we must, again, take into account the set of privileged cubes. The set of privileged cubes can be

derived by introducing for each transition a privileged cube for each output that changes from 1 to 0. The literal form of the multi–valued variable in each privileged cube corresponds with the output for which the privileged cube is meant.

Consider transition $\uparrow 0 \downarrow 1 \rightarrow 1 \downarrow\uparrow\downarrow$. Two privileged cubes are created for the second and fourth output variables: y_2 and y_4. These privileged cubes are: $(\overline{x}_2 x_4 x_5^{\{1\}}, \overline{x}_1\overline{x}_2 x_3 x_4 x_5^{\{1\}})$ and $(\overline{x}_2 x_4 x_5^{\{4\}}, \overline{x}_1\overline{x}_2 x_3 x_4 x_5^{\{4\}})$, where variable x_5 is the multi–valued variable representing the multiple outputs. These privileged cubes can also be written as: $(\overline{x}_2 x_4\ 0100, \overline{x}_1\overline{x}_2 x_3 x_4\ 0100)$ and $(\overline{x}_2 x_4\ 0001, \overline{x}_1\overline{x}_2 x_3 x_4\ 0001)$.

The multi–valued threeway method can be adapted to take into account the set of, in this case, multi–valued privileged cubes. This results into the following formula:

$$\begin{aligned}\text{DHFPrimes}(F, P) = {} & \text{DHFAlphaPrimes}(F, P) \cup \text{DHFBetaPrimes}(F, P) \cup \\ & \text{DHFAlphaBetaPrimes}(F, P)\end{aligned} \tag{4.29}$$

Here, DHFAlphaPrimes(F, P) generates the set of dhf–prime implicants for which the literal form of a chosen input variable x_i is a subset of α, where α and β represent, again, a partition of the values that can be assumed by a selected variable x_i. The prime implicants generated by DHFAlphaPrimes(F, P) are included in the implicants generated by:

$$\text{DHFPrimes}(x_i^{\alpha} \cdot F, x_i^{\alpha} \odot P) \tag{4.30}$$

In contrast with formula 4.10, the set of privileged cubes do not cause any change of the onset. Consider multi–valued variable x_i. The literal form of variable x_i is the same in the starting point and the body of each privileged cube. Therefore, based on variable x_i, we cannot determine a subset of the set of privileged cubes that will always be intersected illegally. Formula 4.30 can be rewritten as:

$$x_i^{\alpha} \odot \text{DHFPrimes}(F|_{\alpha}, P|_{\alpha}) \tag{4.31}$$

A similar result can be obtained for DHFBetaPrimes(F, P). In the prime implicants generated by DHFAlphaBetaPrimes(F, P), the literal form of a chosen variable x_i has a non–empty intersection with both set α and β. We already saw that, in order to take this into account, it is necessary to introduce a new function, F′, that has a changed support set compared to function F. The chosen multi–valued variable x_i is split into two new multi–valued variables, x_{i1} and x_{i2}. We can express DHFAlphaBetaPrimes(F, P) as:

$$\text{DHFPrimes}(F|_{\alpha} \cdot F|_{\beta}, P') \tag{4.32}$$

Note that in this formula there is again an implicit conversion from support set $\vec{x}'$ to $\vec{x}$. Set P′ is used to avoid illegal intersections by prime implicants generated based on $F|_\alpha \cdot F|_\beta$. In order to do so, P′ must have the same support set as $F|_\alpha \cdot F|_\beta$, meaning that in set P′, multi–valued variable x_i is also split into two multi–valued variables x_{i1} and x_{i2}.

Consider a prime implicant c in which variable x_i has a non–empty intersection with both sets α and β. Suppose prime c illegally intersects a privileged cube $p \in P$. If we convert both c and p from support set $\vec{x}$ to $\vec{x}'$, this illegal intersection might disappear, that is: in the new support set no illegal intersection will occur.

As an example, suppose $c = x_1 x_3^{\{1,4\}}$ and $p = (\overline{x}_2 x_3^{\{1,3\}}, \overline{x}_2 \overline{x}_1 x_3^{\{1,3\}})$. Here variable x_3 is a multi–valued variable. It is clear that c illegally intersects p. Now, suppose we split variable x_3 into variables x_{31} and x_{32}, where $x_{31} \in \{1,2\}$ and $x_{32} \in \{3,4\}$. Then cube c and privileged cube p can be written as: $c = x_1 x_{31}^{\{1\}} x_{32}^{\{4\}}$ and $p = (\overline{x}_2 x_{31}^{\{1\}} x_{32}^{\{3\}}, \overline{x}_2 \overline{x}_1 x_{31}^{\{1\}} x_{32}^{\{3\}})$. Now, c no longer illegally intersects privileged cube p, because of variable x_{32}. Variable x_{32} masks the illegal intersection.

So, converting the set of privileged cubes P to set P′, by changing the support set from $\vec{x}$ to $\vec{x}'$ can result in primes being generated by formula 4.32 that are hazardous after they are cast back in support set $\vec{x}$. In order to prevent this, each privileged cube p will add both $p|_\alpha$ and $p|_\beta$ to set P′. Privileged cube $p|_\alpha$ guarantees that illegal intersections can no longer be masked by variable x_{i1}. Similarly, privileged cube $p|_\beta$ guarantees that illegal intersections can no longer be masked by variable x_{i2}. Since P′ can be expressed as: $P' = P|_\alpha \cup P|_\beta$, we can express formula 4.32 as:

$$\text{DHFPrimes}(F|_\alpha \cdot F|_\beta, P|_\alpha \cup P|_\beta) \tag{4.33}$$

The threeway method for multiple outputs can be described by:

$$\begin{aligned} \text{DHFPrimes}(F,P) = \text{SCC}(&x^\alpha \odot \text{DHFPrimes}(F|_\alpha, P|_\alpha) \cup \\ &x^\beta \odot \text{DHFPrimes}(F|_\beta, P|_\beta) \cup \\ &\text{DHFPrimes}(F|_\alpha \cdot F|_\beta, P|_\alpha \cup P|_\beta)) \end{aligned} \tag{4.34}$$

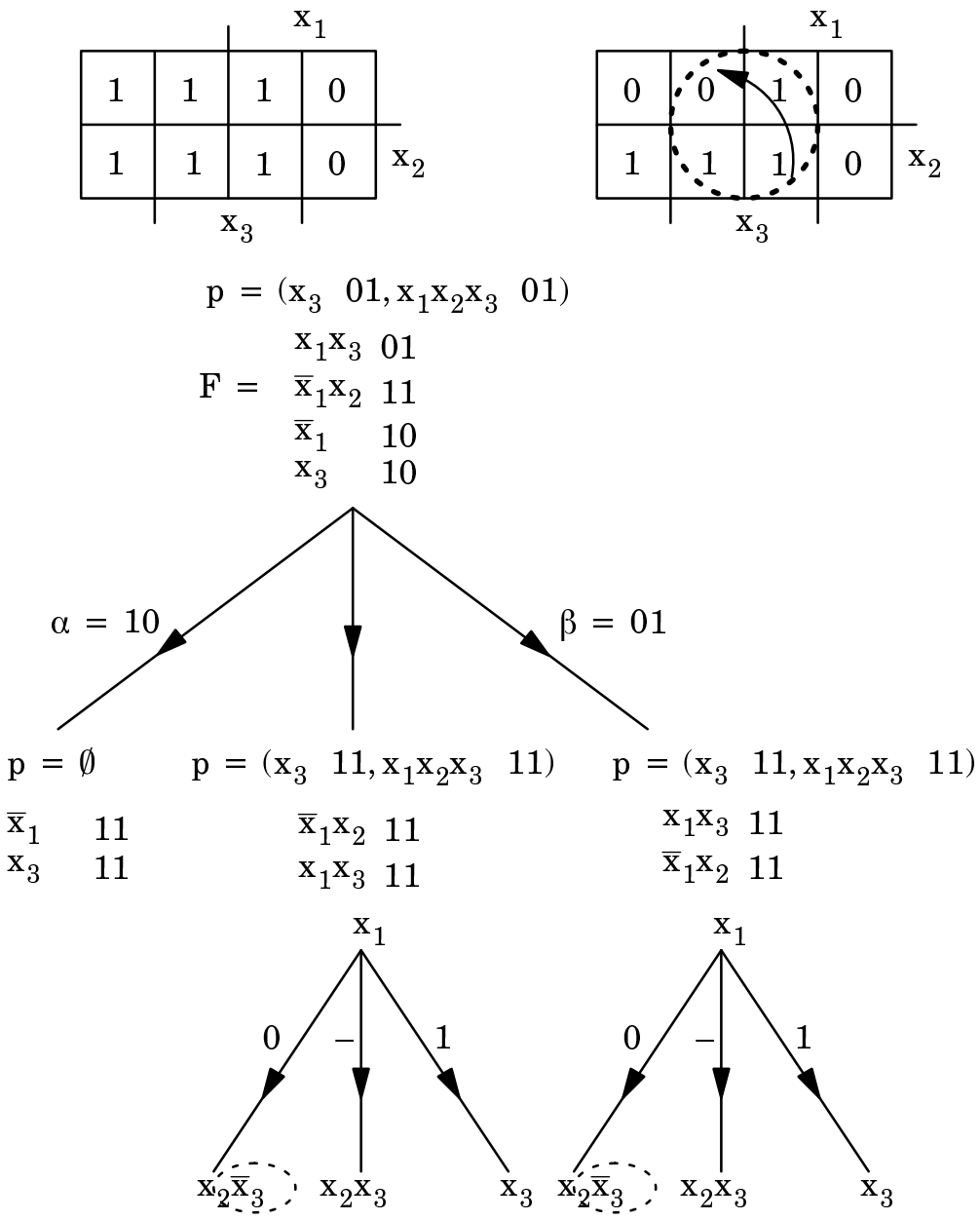


FIGURE 4.11. The multi–valued dhf–threeway method in action

Consider the example in figure 4.11. We once more use the specification introduced in section 4.5.3. By adding transition $\downarrow\downarrow 1 \rightarrow 1 \downarrow$ to this specification, we also add a privileged cube: the transition introduces a privileged cube for the second output variable. The karnaugh map of each output variable is also shown. The two output variables are combined into one multi–valued variable. Sets α and β are straightforward in the case of two output variables. Since the privileged cube is meant for the second output variable, the set of privileged cubes for the α–edge is empty. The set of set of privileged cubes for both the β–edge and the middle–edge do contain the privileged cube

cofactored to β. Since the resulting functions are all unate in the multi–valued variable, a binary variable is chosen in the next stage, except for the α–edge, since this edge only contains implicants that are unate in each variable. The resulting prime implicants are:

$\bar{x}_1$	10
$\bar{x}_1 x_2 \bar{x}_3$	11
$x_2 x_3$	11
$x_1 x_3$	11
x_3	10

4.5.5 Pruning the number of multiple–output dhf–prime implicants

We can adjust the multi–valued threeway method in order to take into account the set of required cubes so that we can prevent the generation of non–contributing prime implicants. The derivation of the set of required cubes for a set of transitions containing multiple–outputs is similar to the derivation of the set of privileged cubes, which is described in the previous section.

Edges of the search tree are again discarded if no required cube remains. By incorporating the set of required cubes, the multi–valued threeway method can be expressed as:

$$\begin{aligned} \text{DHFPrimes}(F, P, Q) = {} & \text{DHFAlphaPrimes}(F, P, Q) \cup \\ & \text{DHFBetaPrimes}(F, P, Q) \cup \\ & \text{DHFAlphaBetaPrimes}(F, P, Q) \end{aligned} \tag{4.35}$$

Let us consider DHFAlphaPrimes(F, P, Q). This formula can be written as:

$$x_i^{\alpha} \odot \text{DHFPrimes}(F|_{\alpha}, P|_{\alpha}, Q') \tag{4.36}$$

Q′ should contain the set of required cubes that imply outputs belonging to set α. We only remove those required cubes from consideration that do not imply any output in set α. This can be accomplished by a mere cofactoring operation. Formula 4.36 can therefore be written as:

$$x_i^{\alpha} \odot \text{DHFPrimes}(F|_{\alpha}, P|_{\alpha}, Q|_{\alpha}) \tag{4.37}$$

After some derivations similar to those discussed in the previous sections, we obtain the expression for the complete dhf–threeway method that is capable of calculating the set of contributing prime implicants.

$$\begin{aligned} \text{DHFPrimes}(F, P, Q) = {} & \text{SCC}(x^{\alpha} \odot \text{DHFPrimes}(F|_{\alpha}, P|_{\alpha}, Q|_{\alpha}) \cup \\ & x^{\beta} \odot \text{DHFPrimes}(F|_{\beta}, P|_{\beta}, Q|_{\beta}) \cup \\ & \text{DHFPrimes}(F|_{\alpha} \cdot F|_{\beta}, P|_{\alpha} \cup P|_{\beta}, Q|_{\alpha} \cup Q|_{\beta})) \end{aligned} \tag{4.38}$$

4.5.6 Explicit representations of multiple output implicants

The multi–valued threeway methods also do not map well on implementations targeting explicit data structures. Therefore, we again make use of the offset representation of a function, to derive a set of threeway methods, which are more suitable for our explicit data structures.

By applying deMorgan, we can rewrite the multi–valued threeway method in formula 4.28 as:

$$\begin{aligned} \text{Primes}(R) = \text{SCC}(&x^{\alpha} \odot \text{Primes}(R|_{\alpha}) \cup x^{\beta} \odot \text{Primes}(R|_{\beta}) \cup \\ &\text{Primes}(R|_{\alpha} + R|_{\beta})) \end{aligned} \tag{4.39}$$

Note that $R|_{\alpha} + R|_{\beta}$ also has a changed support set compared to R. Similar to the explicit expression in section 4.4.4, Primes(R) does not calculate the set of prime implicants belonging to the offset R. Again, the problems become trivial when the offset is empty or equivalent to a tautology.

Based on the offset, the expression for generating the set of all dhf–prime implicants becomes:

$$\begin{aligned} \text{DHFPrimes}(R, P) = \text{SCC}(&x^{\alpha} \odot \text{DHFPrimes}(R|_{\alpha}, P|_{\alpha}) \cup \\ &x^{\beta} \odot \text{DHFPrimes}(R|_{\beta}, P|_{\beta}) \cup \\ &\text{DHFPrimes}(R|_{\alpha} + R|_{\beta}, P|_{\alpha} \cup P|_{\beta})) \end{aligned} \tag{4.40}$$

Finally, the expression that only generates a set of contributing dhf–prime implicants becomes:

$$\begin{aligned} \text{DHFPrimes}(R, P, Q) = \text{SCC}(&x^{\alpha} \odot \text{DHFPrimes}(R|_{\alpha}, P|_{\alpha}, Q|_{\alpha}) \cup \\ &x^{\beta} \odot \text{DHFPrimes}(R|_{\beta}, P|_{\beta}, Q|_{\beta}) \cup \\ &\text{DHFPrimes}(R|_{\alpha} + R|_{\beta}, P|_{\alpha} \cup P|_{\beta}, Q|_{\alpha} \cup Q|_{\beta})) \end{aligned} \tag{4.41}$$

4.6 A heuristic dhf–minimizer

As we saw in the previous section, we can extend the threeway method such that it is able to deal with multi–valued functions. The latter allows us to generate the set of all dhf–prime implicants of a general multiple output specification. Unfortunately, the biggest problem instances can not be solved using our exact method, therefore we have to consider heuristic methods. We propose a heuristic minimizer based on the threeway method. It is heuristic in that it will not generate the set of all dhf–prime implicants but only a subset. Although we can not guarantee that the smallest solution is contained

in this subset, we can guarantee that at least one solution is contained in this subset.

The heuristic method can be seen as consisting of two parts. One part deals with binary valued variables and the other part deals with multi–valued variables. For binary valued variables, the pruned threeway method described in section 4.4.3 is used.

For multi–valued variables, we will use a modified version of the method explained in section 4.5.5. Consider formula 4.38, where we leave out the set of prime implicants that cover both elements of set α and β. This formula can be written as:

$$\begin{aligned} \text{DHFPrimes}(F, P, Q) = {} & x^{\alpha} \odot \text{DHFPrimes}(F|_{\alpha}, P|_{\alpha}, Q|_{\alpha}) \cup \\ & x^{\beta} \odot \text{DHFPrimes}(F|_{\beta}, P|_{\beta}, Q|_{\beta}) \end{aligned} \tag{4.42}$$

The implicants generated by formula 4.42 are not prime, but they are hazard–free, and they do cover each required cube for each separate output. Avoiding the generation of the set of prime implicants that cover both elements from set α and β has a huge impact on the runtime of the algorithm. As was said in section 2.3, there can be as many as $3^n/n$ prime implicants for synchronous functions. There is no reason to believe that this number is smaller for asynchronous logic. If function F is specified as the disjunction of the set of required cubes Q, where each required cube represents a single output, then formula 4.42 will generate at most the same number of implicants.

By taking a minimum selection of the set of implicants generated by formula 4.42, we can obtain a hazard–free cover. This cover will most probably not be minimum in the number of implicants selected since the implicants selected are not necessarily prime, and since a minimum cover might require implicants that cover both elements from set α and β. Although we cannot guarantee a minimum solution, we can convert each implicant generated by formula 4.42 into a prime implicant, and thus possibly obtain smaller covers.

Consider an implicant i generated by $x^{\alpha} \odot \text{DHFPrimes}(F|_{\alpha}, P|_{\alpha}, Q|_{\alpha})$. The literal form of variable x in this implicant covers no values in β. We will convert this implicant to a prime implicant by adding values in set β to the literal form of variable x. It is of course clear that it is not possible to just add all values in β. We only add those values such that the implicant does not intersect the offset, and does not illegally intersect any privileged cube.

Let the set of values to be added be called γ. Thus: $\gamma \subseteq \beta$. Then γ must be such that $x^{\alpha \cup \gamma} \cdot i|_{\alpha}$ does not intersect the offset R and does not illegally intersect

any privileged cube P. Let the operator that determines set γ be denoted by EXPAND. Operator EXPAND takes three argument, the set of implicants to be 'expanded', the offset R and the set of privileged cubes P. By converting each implicant generated by formula 4.42 to a prime implicant, this formula can be expressed as:

$$\begin{aligned} \text{DHFPrimes}(F, P, Q) = \; & \text{SCC}(\text{EXPAND}(I_\alpha, R, P) \cup \\ & \text{EXPAND}(I_\beta, R, P)) \end{aligned} \tag{4.43}$$

Where,

$$I_\alpha = x^\alpha \odot \text{DHFPrimes}(F|_\alpha, P|_\alpha, Q|_\alpha) \tag{4.44}$$

$$I_\beta = x_\beta \odot \text{DHFPrimes}(F|_\beta, P|_\beta, Q|_\beta) \tag{4.45}$$

In formula 4.43, the selected multi–valued variable x upon which operator EXPAND operates is implicit.

Consider the prime implicants generated by EXPAND(I_α, R, P). For each implicant $i \in I_\alpha$ a maximal set γ is determined such that i does not intersect the offset. So γ should obey the following equation:

$$x^{\alpha \cup \gamma} \cdot i|_\alpha \cdot R = 0 =$$

$$x^{\alpha \cup \gamma} \cdot i|_\alpha \cdot (x^\alpha \cdot R|_\alpha + x^\beta \cdot R|_\beta) = 0 =$$

$$x^{\gamma \cap \beta} \cdot i|_\alpha \cdot R|_\beta = 0$$

In this derivation, we used a multi–valued extension of the Shannon expansion. Based on the last equation, we conclude that operator EXPAND can also determine set γ based on set $R|_\beta$ instead of the complete offset R.

In order to prevent an illegal intersection, set γ must be such that $x^{\alpha \cup \gamma} \cdot i|_\alpha$ does not illegally intersect any privileged cube $p \in P$. Since implicant i is a member of set I_α, it will not intersect any privileged cube belonging to set $P|_\alpha$ illegally. Therefore, we only need to consider the privileged cubes in set $P|_\beta$ in order to derive set γ. Based on this observation, we conclude that operator EXPAND does not need the set of all privileged cubes: the set of privileged cubes in $P|_\beta$ are sufficient. The set of prime implicants can therefore be expressed as:

$$\text{EXPAND}(I_\alpha, R, P) = \text{EXPAND}(I_\alpha, R|_\beta, P|_\beta) \tag{4.46}$$

This observation allows us to rewrite equation 4.43 into:

$$\begin{aligned} \text{DHFPrimes}(F, P, Q) = \; & \text{SCC}(\text{EXPAND}(I_\alpha, R|_\beta, P|_\beta) \cup \\ & \text{EXPAND}(I_\beta, R|_\alpha, P|_\alpha)) \end{aligned} \tag{4.47}$$

As we will see in the next section, the heuristic method described by formula 4.47 dramatically reduces the runtime of the logic synthesis step.

4.7 Results

We have implemented the described threeway method, and compared it with Dill/Nowick's method. For Dill/Nowick's method, we used the two–level minimizer espresso to generate the set of all prime implicants. These primes are then rendered dynamic hazard–free by Dill/Nowick's hazard removal algorithm. In Tables 4.1 and 4.2 the results are presented.

TABLE 4.1 Single output results

Name	I	Espr + HF	Threeway	Contr	Min
sbuf–send–ctl.pla	6	8/–	8/–	4/–	2
isend–bm.pla	9	10/–	10/–	10/–	3
pe–send–ifc.pla	8	20/–	20/–	8/–	6
isend.pla	8	27/–	27/–	7/–	4
p2.pla	13	45/–	45/–	12/–	5
pscsi.pla	16	522/0.6	522/1.0	175/0.3	17
p1.pla	17	187/0.2	187/0.1	60/0.1	6
cache–ctrl.pla	21	740/3.6	740/27	59/4	17
scsi.pla	18	2001/6.5	2001/3.5	414/0.7	14
dramc.pla	9	32/–	32/–	5/–	4
ircv.pla	7	6/–	6/–	6/–	3
isend–csm.pla	7	15/–	15/–	3/–	2
trcv–bm.pla	7	10/–	10/–	3/–	2
trcv–csm.pla	7	15/–	15/–	3/–	2
tsend.pla	8	17/–	17/–	12/–	4
tsend–bm.pla	9	30/–	30/–	8/–	3
tsend–csm.pla	9	26/–	26/–	7/–	3

'–' : runtime too small to measure (< 0.1 [s])

Table 4.1 depicts the results for single output benchmark circuits. Table 4.2 shows the results for multiple–output benchmark circuits. All benchmark circuits are derived from well known burst–mode machines, and represent the largest examples available. The name of these benchmark circuits is placed in the first column, "Name". In the second column "I/O" the number of input and output variables of each benchmark circuit is depicted. In table 4.1 the number of outputs is always equal to 1, therefore the "O" is omitted in this table. In the third column "Espr+HF", the results of Dill/Nowick's method are depicted. These results are represented with two numbers: the number of

dhf–prime implicants and the runtime necessary, in seconds. In Column "Threeway" we can observe the results of the threeway method, in which we generate all dhf–prime implicants. The number of primes in this column must therefore be equal to the number of primes in the third column. Again, the run times are shown. The numbers in column "Contr" represent the number of contributing dhf–prime implicants and the runtime by the contributing threeway method. Column "Min" represents the minimum solution. A unate covering algorithm, like mincov [Sent92], was used to obtain these results.

TABLE 4.2 Multiple–outputs results

Name	I/O	Espr + HF	Threeway	Contr	Min	Heur	Heur Min
sbuf–send–ctl.big.pla	6/6	34/–	34/–	19/–	11	18/–	11
isend–bm.big.pla	9/8	135/0.17	135/0.62	71/0.20	17	53/–	19
pe–send–ifc.big.pla	8/6	72/–	72/0.17	43/–	21	40/–	21
isend.big.pla	8/7	131/0.2	131/0.4	72/0.1	20	48/–	20
p2.big.pla	13/17	285/1.26	285/19.4	179/1.78	28	133/0.37	28
pscsi.big.pla	16/11	2878/89	2878/2135	1219/139	68	844/4.7	71
p1.big.pla	17/18	1145/18	1145/1428	659/117	56	353/2.73	59
cache–ctrl.big.pla	21/24	*	*	*	?	1999/120	126
scsi.big.pla	18/14	26434/23020	*	6491/938	90	3052/13.6	99
dramc.pla	9/8	87/0.2	87/0.8	48/0.2	22	41/0.1	22
ircv.pla	7/6	41/–	41/–	23/–	11	19/–	11
isend–csm.pla	7/6	41/–	41/–	26/–	12	21/–	13
trcv–bm.pla	7/7	43/–	43/0.1	22/–	14	21/–	14
trcv–csm.pla	7/6	38/–	38/–	21/–	12	19/–	12
tsend.pla	8/7	107/0.1	107/0.4	68/0.1	20	59/–	21
tsend–bm.pla	9/8	134/0.2	134/0.4	65/0.1	19	50/–	19
tsend–csm.pla	9/8	148/0.1	148/0.3	63/0.1	15	45/–	16

'*' : calculation aborted after 15 hours, '–' : runtime too small to measure (< 0.1 [s]), '?' : Solution unknown

Table 4.2 has two extra columns compared to table 4.1. Column "Heur" represents the results of our proposed heuristic method. Again, the number of dhf–prime implicants is combined with the runtime in seconds. The last column represents the minimum solution based on the set of dhf–prime implicants generated by the heuristic method. These two columns are omitted from table 4.1, because for single–output examples, the heuristic method is equal to the exact threeway method, where only contributing dhf–prime implicants are generated. All programs, including espresso, were compiled with the same compiler, and with the same compiler options to be able to make a fair comparison between the different methods.

From table 4.1 it is clear that the threeway method generates comparable runtimes compared to Dill/Nowick's method. However, it is clear that pruning the number of dhf–prime implicants has a big impact on the runtime of the algorithm. For the larger examples, the pruned threeway–method clearly outperforms Dill/Nowick method.

These observations, unfortunately, do not apply for the multiple–output benchmark circuits. In table 4.2 it can be observed that the threeway method doesn't perform well, in terms of runtime, compared to Dill/Nowick's method. The reason for the poor performance can be explained by the increase in the support set for the sub–problem that calculates the set of prime–implicants that have a non–empty intersection with two sets α and β. By introducing new input variables, we actually make the recursion tree 'deeper'. Also, the number of cubes that these sub–problems have to deal with tends to *increase* for the first couple of recursions, which seemingly increases the complexity of the problem instead of reducing it.

Both facts are to be blamed for the many recursions it takes before the sub–problems become trivial. We can also see in table 4.2 that generating the set of contributing primes has a dramatic impact on the runtimes of the threeway method. It is difficult to compare the performance of the contributing threeway method with Dill/Nowick's method. For most benchmark circuits Dill/Nowick's method clearly outperforms the contributing threeway method. However, it seems that the contributing threeway method outperforms Dill/Nowick's method for larger benchmarks. More experiments with larger benchmarks are needed to confirm this observation. Unfortunately, there are currently no other large benchmark circuits available.

It can be observed from table 4.2 that the heuristic method we have proposed is fast and efficient. It is able to solve, in seconds, problems where Dill/Nowick's method is cancelled after 15 hours of calculation. Furthermore, the degradation of the final solution is acceptable. In fact, it turns out that the heuristic method is capable of finding the exact solution.

4.8 The work of Theobald

In [Theo98a] Theobald describes an alternative method that is capable of generating the set of all dhf–prime implicants from which a minimum selection is made. For a single output function f, an auxiliary function g is introduced. Function g depends on all the input variables of function f, and on an additional set input variables. The cardinality of the latter set is equal to the number of privileged cubes.

Function g is created by first copying function f, so g=f. For each privileged cube p a new input variable z is introduced, which doubles the Boolean space on which the function g is defined. Since g doesn't depend on variable z, $g|_{\bar{z}} = g|_{z}$. Now the body of privileged cube p is removed from $g|_{z}$. That is: $g|_{z}$ is made equal to 0 for the Boolean space belonging to the body of privileged cube p. The described process is repeated for each privileged cube: a new variable is introduced, and in the positive cofactor of function g the Boolean space covered by the body of that privileged cube is made equal to 0.

Theobald showed that the set of all prime implicants of function g covers the set of all dhf–prime implicants of f. By removing the hazardous prime–implicants, and by performing a SCC filtering operation, one can obtain the set of dhf–prime implicants. From this set of dhf–prime implicants a minimum selection is made that covers the set of all required cubes. The described method can be easily adapted to also take into account multiple–output functions.

The method developed by Theobald effectively transforms the problem of generating all dhf–prime implicants into a normal 'synchronous' problem of generating all prime implicants. Therefore his method can use any existing exact synchronous minimizer. In [Theo98a] Theobald used the method proposed by Coudert in [Coud94]. In this paper Coudert describes how the problem of two–level minimization can be solved by using implicit techniques. Boolean functions are described by BDDs and set of cubes by so–called combinational sets. For the latter Theobald uses ZBDDs.

Coudert/Theobald's approach highly benefits from being able to represent Boolean functions/sets of cubes with the described data structures. BDDs/ZBDDs/combinational sets are in practice able to represent functions/cubes that cannot be represented explicitly with e.g. a linked list due to the large number of elements required. Many operations that are very expensive on explicit data structures are cheap with the above data structures. For example, comparing if two functions are equal takes constant time if these functions are represented with BDDs.

The most important reason, however, to use BDDs during the generation of the prime implicants is that they are able to recognize an already encountered sub–function. For this sub–function the set of prime implicants has already been calculated, and this result can be used immediately. An explicit method cannot do this as efficiently.

Coudert also shows that by using implicit data structures, there is no need to generate an explicit covering matrix to solve the unate covering problem of finding a minimum selection of dhf–prime implicants. He shows that the

unate covering problem can be mapped onto a set covering problem over a lattice, where the two sets are the prime implicants and minterms of the function respectively. Both sets can be represented by combinational sets or ZBDDs. In Theobald's approach the two sets constitute the set of dhf–prime implicants and the set of required cubes.

In [Theo96] Theobald also proposes a hazard–free heuristic minimizer based on the heuristic version of espresso, espresso–II [Rud87]. In [Theo98b] the runtimes of the implementation of this heuristic algorithm have been dramatically improved. The runtimes of Theobald's heuristic method are comparable with our heuristic method.

4.9 Theobald's method versus the threeway method

The dhf–threeway method, in its current implementation, uses an explicit data structure. Both Boolean functions and cubes are represented by a linked list, where each element represents a single cube. Because of this, and because we cannot detect already encountered sub–functions, the exact threeway method in its current form is being outperformed by Theobald's method, especially for larger benchmark circuits.

It is possible, in future work, to implement the threeway method by using implicit data structures like BDDs and ZBDDs. BDDs make use of tables to hash the nodes of BDDs and to store computed results of so–called ITE–operations [Hach96]. Next to these tables we might introduce an additional table, which we will call the prime table that stores the result of calculated (dhf–)prime implicants. The prime table will check if the result of calculating the set of prime implicants of function F and privileged set P is available.

It is difficult to predict if the implicit threeway method will outperform Theobald's method. The implicit threeway method only generates the set of dhf–prime implicants, so there is no need to filter away the non–dhf implicants necessary in Theobald's method. Also, there is no need to introduce an auxiliary function g that has a larger support set of input variables compared to function f. Additional variables imply bigger BDDs which will degrade the performance. Instead of adding additional variables, the implicit threeway method modifies function f by removing bodies of privileged cube when these cubes can never be intersected legally. The latter might degrade the performance of the implicit threeway method. The implicit threeway method can still use the set of required cubes to avoid the generation of so–called non–contributing prime implicants. This is likely to have an impact on the performance, for it means that parts of the BDD can be skipped when the prime implicants generated by those parts will not be contributing.

Chapter 5 Verification

5.1 Introduction

Verification is very important during the synthesis of digital systems in general. Verification can be applied to check important properties, for example, verification can check if a logic network is functionally equivalent to the original specification. Here, we apply verification to check whether an asynchronous circuit does exhibit hazardous behavior.

Eichelberger [Eich65] proposed a ternary algebra in order to detect hazards. His ternary algebra is an extension to the ordinary Boolean algebra used to analyze switching circuits. Next to values 0 and 1, an additional value 1/2 is introduced. The ternary algebra itself is no longer a Boolean algebra. Value 1/2 is used to indicate that a signal might change, given a set of input changes. He showed how one can obtain a ternary function of each logic gate. He proved that outputs that assume value 1/2 during a static transition exhibit a static hazard. His approach has as a drawback that it is not able to detect dynamic hazards. A more detailed, and formal explanation of ternary algebra, and its applications is given by [Brzo95].

Kung [Kung92] noted that the possible waveforms that can be assumed by a signal, can be partitioned into nine different classes. Each class can be assigned a different value, as is shown in table 5.1.

TABLE 5.1 The nine different classes to which the behavior of a signal can belong

Value	waveform	Description
1		static at 1
0		static at 0
↑		0–to–1 change
↓		1–to–0 change
S0		static 0 hazard
S1		static 1 hazard
D+		dynamic 0–to–1 hazard
D–		dynamic 1–to–0 hazard
*		don't care

Like Eichelberger, Kung extends the logic functions used to describe logic gates to accommodate his nine–valued algebra, which is also not a Boolean algebra. This is accomplished by introducing a partial order on the nine values. The order introduced by Kung is: $* > 0$, $* > 1$, $* > \downarrow$, $* > \uparrow$, $* >$ S0, $* >$ S1, $* >$ D+, $* >$ D–, S0 > 0, S1 > 1, D+ $> \uparrow$, and D– $> \downarrow$.

The value of a logic gate is defined as the maximum value, according to the partial order relation, of the values belonging to the classes of the waveforms that might occur under the given set of input waveforms. Kung's approach is able to detect both static and dynamic hazards.

The method we describe in this chapter is equivalent to Kung's approach, with one minor modification. Instead of nine values, we show that for our applications 5 values suffice to detect all hazards, static and dynamic. Although Kung's method is able to detect all hazards, it cannot verify if a combinational circuit implements each transition such that it obeys the Burst–Mode condition. We show how Kung's method can be modified to check the latter.

5.2 Problem definition

We want to verify that an asynchronous burst–mode machine does not exhibit hazardous behavior. In figure 5.1 we have again depicted a general architecture of a BMM.

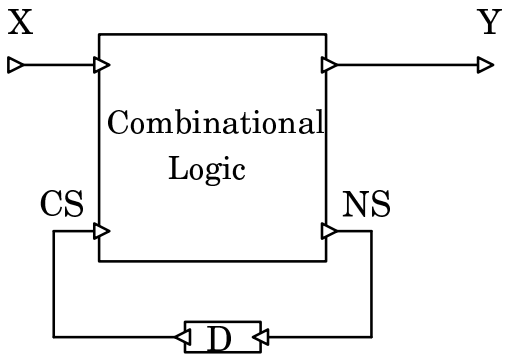


FIGURE 5.1. General architecture of a BMM

Any hazardous behavior of the circuit is caused by the combinational logic block. The delay elements cannot introduce hazards. Furthermore, as was explained in chapter 3.4, we assume that the delay elements are such that the combinational logic block will observe two separate transitions: an input transition applied by the environment, followed by a possible current state transition. The delay elements must be such that the machine can be assumed to be in rest when the current state transition arrives.

Because of these assumptions we can cut the feedback in figure 5.1. The resulting architecture is depicted in figure 5.2.

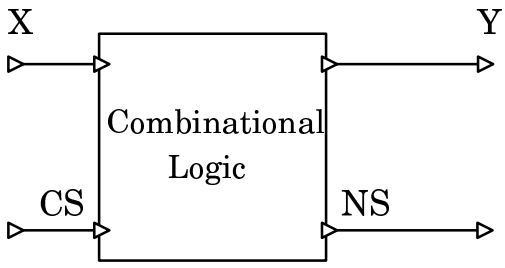


FIGURE 5.2. Cutting the feedback of a BMM

The machine description can now be considered as consisting of a set of transitions, namely input transitions and current state transitions. We have to verify that the combinational logic is free from hazards for the combined set of input and current state transitions.

5.3 Verifying with 5–valued algebra

In section 2.7 a transition is defined as:

DEFINITION 5.1
$T \in D^n \times D^m$, where $D \in \{0, 1, \uparrow, \downarrow\}$.

To detect hazards, we are going to modify this definition, by introducing an extra value in set D, one that represents a hazardous behavior. The new definition becomes:

DEFINITION 5.2
$T \in D^n \times D^m$, where $D \in \{0, 1, \uparrow, \downarrow, H\}$.

If an input, or an output assumes value H, then that input or output shows hazardous behavior. The kind of hazard can be found by comparing the hazardous behavior to the original specification. So, if the original specification is $\uparrow\downarrow$ 0 $\rightarrow\uparrow$ 0 $\downarrow$, and the calculated response of the circuit is $\uparrow\downarrow$ 0 $\rightarrow$ HH $\downarrow$, then the first output exhibits a dynamic 0–to–1 hazard, while the second output clearly shows a static–0 hazard.

The transition definition in definition 5.2 can be considered as an expansion of the Boolean algebra used to analyze switching circuits. Instead of two values 0 and 1, we have 5 values, namely $0, 1, \uparrow, \downarrow$, and H. Like Kung's algebra, each value is associated with a class of possible waveforms. We associate values 0, 1, $\uparrow$, and $\downarrow$ with the same class of waveforms proposed by Kung. For value H we introduce a new class that covers Kung's hazardous classes S0, S1, D+, D–. We also induce a partial order relation on the 5 values, namely: $H > 0$, $H > 1$, $H > \uparrow$, and $H > \downarrow$. For our purposes, we don't need the don't care value *.

Each atomic gate, like an AND–gate, or an OR–gate can be described by a Boolean function. As is done by Kung, we extend these Boolean functions to become 5–valued functions. This is accomplished by evaluating the possible waveforms that can occur at the output of the gate under analysis, given the possible sets of input waveforms. This evaluation takes place with the assumption of an unbounded wire–delay model. Since each waveform belongs to a class to which a certain value is assigned, we can reduce this evaluating process to the process of determining the possible values at the output of a gate. Kung proved that these values are always comparable, i.e., the values are related by our partial order. By taking the maximum of the calculated values, we derive the worst–case situation. Because of our assumed delay model, this worst–case situation corresponds with the worst–case situation under the unbounded wire–delay model.

As an example, consider the Boolean operators $+$ and $\cdot$, which can be implemented with a 2–input OR–gate, 2–input AND–gate respectively. We can describe each of these operators with the help of a truth table. These truth

tables can be modified to take into account our 5–valued algebra. The truth tables are depicted in table 5.2 and table 5.3

TABLE 5.2 5–valued extension of the Boolean + operator

+	0	1	↑	↓	H
0	0	1	↑	↓	H
1	1	1	1	1	1
↑	↑	1	↑	H	H
↓	↓	1	H	↓	H
H	H	1	H	H	H

TABLE 5.3 5–valued extension of the Boolean · operator

·	0	1	↑	↓	H
0	0	0	0	0	0
1	0	1	↑	↓	H
↑	0	↑	↑	H	H
↓	0	↓	H	↓	H
H	0	H	H	H	H

In table 5.2, consider the situation in which the first input assumes a waveform belonging to the class described by value ↑, and in which the second input assumes a waveform belonging to the class described by value ↓. The waveforms assumed by the output belong to exactly two classes: class 1 and class H. Waveforms belonging to the first class occur if the first input changes before the second input. Waveforms belonging to the hazardous class occur when the second input changes before the first input. By taking the maximum of the two values, i.e. max(1, H), we derive that for this input combination the output will evaluate to value H. We can repeat this process for all input combinations in order to derive tables 5.2 and 5.3. As we saw, this approach corresponds exactly with a worst case analysis under an unbounded wire–delay model, which is assumed by the logic synthesis step described in chapter 4, and, as we will see, in chapter 6 as well. So the 5–valued algebra is fit for the analysis of the combinational logic of our BMMs.

It is relatively straightforward to derive truth tables for other Boolean operators and gates, like NAND–gates, NOR–gates, or XOR–gates. Extending the truth tables to take into account more than two inputs is also straightforward. With the help of these tables, we can use simulation to verify that a logic network is completely free from hazards. An example is given in figure 5.3. It is the same figure as figure 2.7.

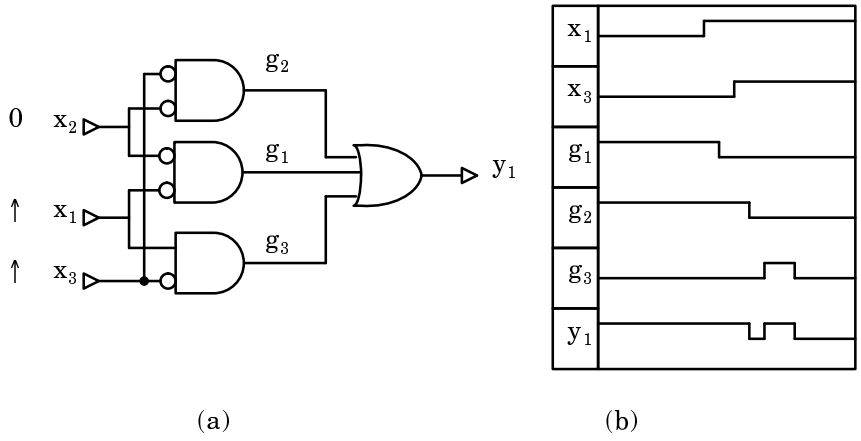


FIGURE 5.3. Example of a circuit exhibiting hazardous behavior

In figure 5.3, we use the proposed 5–valued algebra to find out if a hazard does occur in the depicted circuit. With help of the truth table in table 5.3 we can verify that the values of gates g1 and g2 are both equal to ↓. The value of gate g3, however, will be equal to H, since gate g3 observes two opposite input changes on its inputs. The OR–gate will observe input transition ↓↓ H. This will result in the output value being equal to H, which corresponds with the worst–case behavior under the unbounded wire–delay model.

With the described approach we are able to detect transitions that result in a hazardous behavior of an output of the combinational logic part of the BMM. We can use this to verify that the machine correctly implements each transition free from hazards by simulating the set of all transitions. Applying simulation in this case is fast enough since the set of transitions usually is not big, typically in the order of a few hundred transitions.

Although the described approach is capable of detecting hazardous behavior, the verification method is not able to verify that output changes only take place when all input changes have been observed, i.e. the verification method is not able to verify that each transition obeys the Burst–Mode condition.

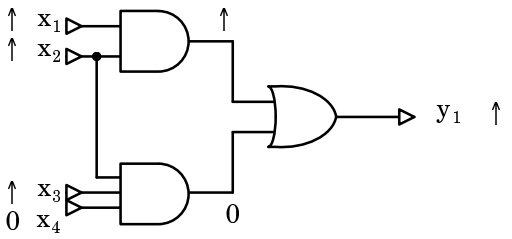


FIGURE 5.4. Hazard–free, but incorrect behavior at the output

The network in figure 5.4, for example, implements transition ↑↑↑ 0 →↑ free from hazards, as will be confirmed by the described verification method. However, the output might turn on before the changes to all the input variables have been completed. This is in violation with the Burst–Mode condition that should be obeyed by each transition, as was defined in section 2.7. Output changes are only allowed to occur when all input changes have taken place. Violation of the Burst–Mode condition could lead to synchronization errors in a system consisting of Burst–Mode machines. Fortunately, there is a rather straightforward method to check that each hazard–free transition also obeys the Burst–Mode condition.

Let t be a transition. Let X be the set of input variables that assume values ↑, or ↓ in transition t. Similarly, let Y be the set of changing input variables in transition s.

DEFINITION 5.3

Transition s is a sub–transition of transition t if $Y \subset X$, if the input variables of transition s in set X/Y do not change, so a ↑ value of the corresponding variable in transition t becomes 0, and similarly value ↓ becomes 1, and if the other variables of s and t are equal.

Transition t is correctly implemented, if for each sub–transition all outputs assume value 0, or value 1, i.e. they remain stable. So by simulating each sub–transition, we can check if a circuit obeys the constraint that output changes only occur after all input changes have been observed.

The number of sub–transitions can grow exponentially in the number of input changes per transition. The number of transitions is important, since the time needed to verify a circuit is a linear function of the number of transitions. Fortunately there is a way to check the desired property without introducing that many new transitions. We need at most $|X|$ new transitions. This reduction is based on the observation that if an output remains stable during a sub–transition, then it will remain stable for all sub–transitions of this sub–transition. So we only need to check those sub–transitions in which all but one input variable of set X assume values ↑, or ↓. There are exactly $|X|$ of these sub–transitions. We only need to add these extra sub–transitions for transitions in which at least one output assumes values ↑, or ↓. Transitions in which all outputs remain stable, for example all current state transitions, do not introduce additional transitions at all. Other reduction strategies, in order to reduce the number of transitions, might be possible by considering the structure of the network to be verified.

The extension described above is currently not implemented in our verifier. Therefore, we cannot guarantee that the circuits mentioned in this thesis obey the Burst–Mode condition.

5.4 Conclusions

The algebra proposed by Kung, modified to reduce the number of values, is capable of detecting hazardous behavior in combinational networks. However, the method is not capable of detecting whether a hazard–free implementation of a set of transitions also obeys the Burst–Mode condition. We have shown that with a small modification Kung's method can also check the latter condition. However, this extension is not implemented in our current implementation.

Our current implementation is based upon simulation. For each transition the network is simulated. If an output assumes value H, a hazard is reported. This approach is feasible because the number of transitions is usually small. To verify the largest circuits we did need to simulate approximately 200 transitions. This number will grow to about 1000 transitions if we also want to check the Burst–Mode condition. For this number of transitions we can continue to use simulation. If in the future the number of transitions does become a bottleneck, it might be interesting to represent transitions implicitly with the help of e.g. BDDs. For the time being, however, simulation suffices. All circuits mentioned in this thesis were verified within a few seconds.

Chapter 6 Multi–level logic

6.1 Introduction

In chapter 4 we saw that it is possible to implement a set of transitions with the help of two–level logic. However, in practice the use of two–level logic is limited to the application of PLAs. In this chapter we will extend the two–level synthesis approach described in chapter 4 to also deal with multi–level logic.

The two–level solutions generated by an asynchronous two–level minimizer as the one described in chapter 4 can be used as a starting point for multi–level logic synthesis. In this respect it is noteworthy that the current developments in the area of asynchronous synthesis follow the same path that has been taken by synchronous synthesis in the past. A multi–level implementation can be derived by applying hazard–non–increasing transformations. A set of these transformations were originally derived by Unger [Ung69]. Kung [Kung92] however, significantly extended this set of transformations. He showed that well–known multi–level synthesis operations like common–cube extraction and kernel extraction [deM94] are indeed hazard–non–increasing. Therefore, many synthesis steps used during synchronous design can be re–used for asynchronous design.

However, there are some important operations used in the synthesis of synchronous multi–level logic that lack the nice property of being hazard–non–increasing. One of these optimization steps consists of using the set of calculated don't cares [deM94] to optimize an asynchronous circuit. For example: a network node that is not observable in a synchronous circuit, might still introduce a hazard in an asynchronous circuit [Beer93]. In this chapter we introduce a multi–level synthesis system dedicated to asynchronous circuits exploiting the set of don't cares [Rut98b][Rut98d].

6.2 Multi level synthesis

Our starting point consists of a hazard–free two–level network generated by the algorithms described in [Theo96], [Theo98a], and chapter 4. By applying hazard–non–increasing transformations, described in [Kung92] we can

derive a multi–level network. What we try to do here, is take into account the set of so–called local don't cares [deM94]. We only deal with the set of CDCs (Controllability don't cares) and ODCs (Observability don't cares). In section 6.2.1 we will look at these so–called controllability don't cares. In section 6.2.3 we will incorporate the set of ODCs. In the remainder of this chapter we will talk about *network nodes*, or simply nodes when no confusion is possible. We define a network node as a two–level network in the multi–level network. So for example, in figure 6.2, the shaded region corresponds with one network node. The network nodes are derived by a program that applies only hazard–non–increasing transformations. The result generated by this program is a multi–level network described in terms of network nodes.

6.2.1 Controllability don't cares

For each network node in the multi–level network, the CDC set specifies the set of all the input assignments, which will never occur. These input combinations are not generated by preceding nodes, or, if the node is directly connected to the primary inputs, they are invalid combinations of the primary inputs. Therefore, this set can be used to optimize the two–level network of the node: the behavior of the node does not matter, i.e. it is a don't care, for this set of input assignments. An example is given in figure 6.1.

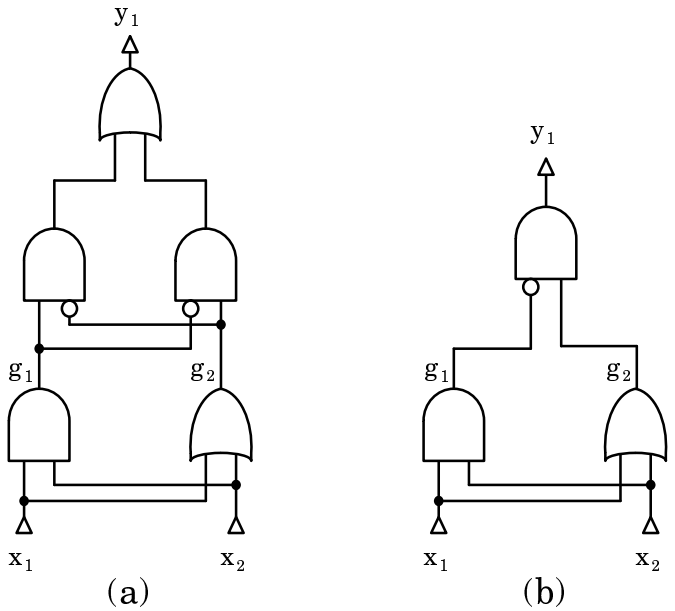


FIGURE 6.1. Example of a controllability don't care

Let's assume that all primary input assignments are valid assignments that can be applied to the circuit in figure 6.1a. If the output of gate g_1 is equal to

one, then the output of gate g_2 must also be equal to one. This means that the combination $g_1 = 1$ and $g_2 = 0$ does not occur. This combination can therefore be added to the set of CDCs for the nodes using the outputs of gates g_1 and g_2. In this case, the CDC set can be used to reduce the implementation of the top network node to the implementation shown in figure 6.1b.

In synchronous logic, a CDC set is calculated by means of a so–called image calculation [deM94]. Although this calculation can be implemented very efficiently with the help of BDDs, it can be a computationally expensive task. For the calculation of the set of CDCs in asynchronous logic, we make use of the 5–valued algebra introduced in chapter 5.

The idea is very simple: we just simulate every transition, as we do during verification of the network. This simulation will result in a set of so–called *node transitions* for each node in the network. Considering such a node, the set of node transitions describes the part of the input space, belonging to the care set, i.e. which local input assignments are valid. By taking the complement of this care set, we obtain the CDC set.

A rather straightforward approach in the optimization of the multi–level network is to resynthesize each node with the given set of calculated node transitions and the corresponding CDC set. That is: the set of required cubes and privileged cubes, derived from the set of calculated node transitions, together with the CDC set is given as input to a hazard–free two–level minimizer. Unfortunately, this approach will in general not lead to smaller implementations of each node. It even turns out that the logic needed to implement each node tends to increase.

This increase of logic can be explained by the fact that the two–level hazard–free minimizer operates on the assumption that each (node) transition, must be implemented such that the Burst–Mode condition, defined in section 2.7 is obeyed. This means that output changes may only occur after all input changes have been observed. Although this condition must be met by the network for each original transition, an internal node can actually violate this condition. So, an internal node can implement a node transition such that output changes can be generated before all the changes of the input variables have been completed. An example of this is given in figure 6.2.

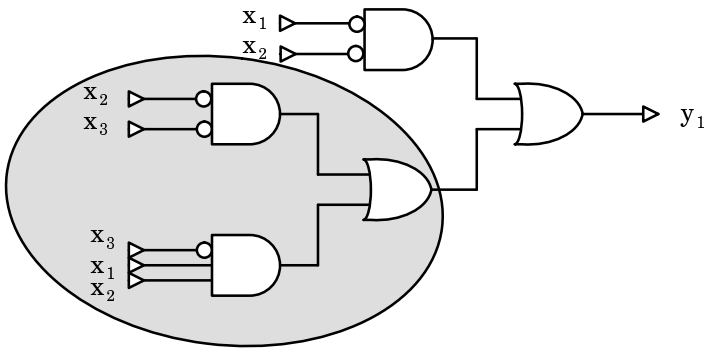


FIGURE 6.2. Internal nodes can violate the Burst–Mode condition

Let us assume that the circuit in figure 6.2 implements a certain set of transitions. One of the transitions implemented by this circuit is transition $\uparrow 0 \uparrow \rightarrow \downarrow$. The circuit as a whole does implement this transition correctly, meaning that the Burst–Mode condition is met. The shaded node has the same support set as the complete network. Simulation with the described 5–valued algebra indicates that the shaded network also implements node transition $\uparrow 0 \uparrow \rightarrow \downarrow$ correctly without hazards. However, if we examine the two–level network of the shaded node more closely, we see that the output of this node can already change from 1 to 0 after input x_3 has completed a 0 to 1 change.

So, although the circuit as a whole has to obey the Burst–Mode condition for each transition it implements, this is not necessarily true for each internal node. By enforcing that each node implements each node transition such that it obeys the Burst–Mode condition, the implementation cost of the node might increase. Even worse, the behavior of the node might become unimplementable under the notion of theorem 4.3.

For internal nodes, we therefore want to relax the Burst–Mode condition for each node transition in such a way that the circuit as a whole still implements each transition correctly: under the Burst–Mode condition. This is accomplished by splitting each node transition into so–called *sub–transitions*. Here the notion of a sub–transition is slightly different from the sub–transition notion introduced in chapter 5. Also note that both a node transition, and a sub–transition, for a network node only contain one output.

6.2.2 Deriving the set of sub–transitions

We define a sub–transition as follows:

DEFINITION 6.1
A sub–transition is a copy of an original node transition, where some of the input changes have been replaced by a '*' change, where value '*' is used to indicate that the corresponding signal can change during the sub–transition.

Suppose that an internal node in a network implements node transition $\uparrow 0 \downarrow 1 \uparrow\downarrow\rightarrow\uparrow$. Suppose also that we know that the output is allowed to make a change from 0 to 1 after inputs x_1 and x_3 have changed, or after inputs x_5 and x_6 have completed their changes. The original node transition is converted to two sub–transitions, namely $\uparrow 0 \downarrow 1 * *\rightarrow\uparrow$ and $* 0 * 1 \uparrow\downarrow\rightarrow\uparrow$. The meaning of value * is similar to the meaning of the directed don't cares in an extended Burst–Mode specification [Yun94].

DEFINITION 6.2
A sub–transition t is said to cover a sub–transition s if the value for the output is the same, and if for each input variable in t its value is greater than the value of the corresponding input variable in s. Here, we introduce the partial order $* > 0$, $* > 1$, $* > \uparrow$ and $* > \downarrow$. Two values both unequal to $*$ are not comparable.

Sub–transitions are calculated only for those node transitions in which the output changes. For node transitions in which the output remains fixed at 0 or 1, no sub–transitions are possible, and for a hazardous change of an output no node transition is introduced at all (note that an internal node can show hazardous behavior for a node transition). The set of sub–transitions can be used to recalculate the set of required cubes. The set of privileged cubes is still derived from the set of node transitions, since both the starting point p^s and the body p^c of each privileged cube do not change.

The set of required cubes can be derived relatively easy once a set of sub–transitions has been determined. So, the question is how to determine the set of sub–transitions. The set of derived sub–transitions for each network node must be such that the complete circuit implements each transition correctly under the Burst–Mode condition. This constraint does allow for a lot of freedom in determining appropriate sub–transitions for each node.

One way to make sure that the circuit obeys the Burst–Mode condition for each transition is by making sure that the behavior of each node remains the same. For each node transition in which the output changes, the behavior can be described exactly by a set of irredundant sub–transitions. If we reimplement this set of sub–transitions, where we will also make use of the CDC–set, we can guarantee that the circuit as a whole obeys the Burst–Mode condition for the corresponding transition. As we saw, sub–transitions are calculated only for those node transitions in which the output of a network

node does assume value ↑ or value ↓. We will first consider node transitions in which the output does assume value ↑.

Remember that each internal node is considered to be a two–level network. In order to determine the set of sub–transitions with which we can describe the behavior of a network node, we start by considering those cubes in the two–level network that will turn on. The two–level expression consisting of these cubes is a unate expression. This can be verified from the fact that these cubes all cover the minterm that corresponds with the end point of the transition. The resulting set of cubes after a SCC operation (Single Cube Containment) are therefore essential and prime. Note that primeness here is based on the function described by the set of cubes that turn on. It is not based on the onset of the network node. Each set of input changes required to enable each of these cubes separately gives rise to a new sub–transition. An example is shown in figure 6.3.

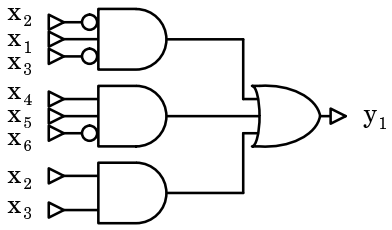


FIGURE 6.3. Example network used to determine sub–transitions for an 0–to–1 change of the output

The network in figure 6.3 consists of three AND–gates. Consider node transition ↑ 0 ↓ 1 ↑↓→↑, which is implemented by the network node. We can observe that during this node transition cubes $x_1\bar{x}_2\bar{x}_3$ and $x_4x_5\bar{x}_6$ will turn on. Cube $x_1\bar{x}_2\bar{x}_3$ will give rise to sub–transition ↑ 0 ↓ 1 * *→↑. This is not difficult to observe, since cube $x_1\bar{x}_2\bar{x}_3$ will turn on when inputs x_1 and x_3 have changed in value and since cube $x_1\bar{x}_2\bar{x}_3$ does not depend on inputs x_5 and x_6. Similarly, cube $x_4x_5\bar{x}_6$ will introduce sub–transition * 0 * 1 ↑↓→↑. The calculated sub–transitions introduce required cubes $x_1\bar{x}_2\bar{x}_3x_4$ and $\bar{x}_2x_4x_5\bar{x}_6$. So, a replacement implementation for this network node should cover these two required cubes.

We now consider node transitions in which the output does assume value ↓. In order to do so, we consider the set of cubes in the two–level network, representing the network node, that will turn off. We start again by determining the set of irredundant sub–transitions.

The set of sub–transitions can be determined by evaluating for which input changes all cubes turn off. By taking the complement of the function, represented by the set of cubes that turn off, the set of sub–transitions can be determined by evaluating for which input changes the complement turns on instead.

The two–level expression obtained by taking the disjunction of the set of cubes that will turn off is an unate expression. This is because these cubes all share the minterm that represents the starting point of the transition. Each cube in an irredundant unate two–level expression is an essential prime implicant. By applying DeMorgan on an irredundant unate two–level expression, we can obtain another irredundant unate two–level expression representing the complement. Again, the cubes making up this expression are essential and prime.

Each of these prime implicants introduces one sub–transition. The input changes necessary to turn one of these prime implicants on represent a sub–transition. In this sub–transition all other inputs which change under the original node transition are assigned value *. So, by taking the complement of the set of cubes that turn off, we can derive the set of sub–transitions in the same way as for an ↑ change of the output. An example is given in figure 6.4.

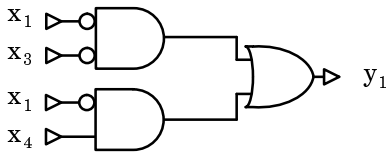


FIGURE 6.4. Example network used to determine sub–transitions for a 1–to–0 change of the output

Consider the circuit in figure 6.4 implementing node transition ↑ 0 ↑↓→↓. Both cubes in the network turn off, so we must take them both into account. Applying DeMorgan on expression $x_1x_3 + x_1x_4$ results in the two–level expression: $x_1 + x_3\bar{x}_4$. Both expressions are irredundant and unate. The two cubes in this latter expression introduce sub–transitions ↑ 0 * *→↓ and * 0 ↑↓→↓.

Based on the set of sub–transitions we can again determine the set of required cubes. In order to do so, we must first determine which part of the transition cube of the node transition belongs to the offset. A transition cube of a (node) transition is obtained by replacing changes, ↑, ↓, and * with a don't care value.

Let operator C return the transition cube of a transition. Note that a transition cube is defined on the set of input variables, so any output is discarded. Let a sub–transition be denoted by t_s, and let the set of all sub–transitions be denoted by T_s. Let E(t) return a transition in which a ↑ value has been replaced by value 1, and in which a ↓ value has been replaced by value 0. So operator E 'executes' the input changes. The resulting transition will contain values 0, 1, and * exclusively. Consider $E(t_s)$. The transition cube belonging to this transition, $C(E(t_s))$, will belong to the offset, since all the necessary input changes have been applied to turn all cubes of the network node off. We can derive the contribution to the offset of each sub–transition in the same way. The contributions to the offset of each sub–transition can be combined into the following formula:

$$\sum O_{T_s} \tag{6.1}$$

where,

$$O_{T_s} = \{C(E(t_s)) \mid t_s \in T_s\} \tag{6.2}$$

O_{T_s} forms a set of cubes. Note that formula 6.1 forms again an irredundant unate expression. This is because the set of sub–transitions is irredundant, and because each cube covers the end point of the original node transition. By sharpening the transition cube of the original node transition with the set of cubes of O_{T_s}, we can derive a set of cubes that form a two–level expression that is unate and irredundant.

This follows from the fact that the above sharp operation can be calculated by taking the intersection of the transition cube with the complement of expression 6.1. The intersection is calculated by a mere AND–operation. Since expression 6.1 is an irredundant unate expression, the complement is also an irredundant unate expression. Therefore, the sharp operation will result in a unate and irredundant two–level expression. The cubes in this expression form the onset and also the set of required cubes. In formula this is expressed as:

$$C(t) \# \sum O_{T_s} \tag{6.3}$$

Consider again the example in figure 6.4. The required cubes belonging to the sub–transitions ↑ 0 * *→↓ and * 0 ↑↓→↓ can be derived as follows:

C(t) = –0––
E(↑ 0 * *→↓) = 10 * *→ 0
E(* 0 ↑↓→↓) = * 010 → 0
C(E(↑ 0 * *→↓)) = 10––

$C(E(*\ 0 \uparrow\downarrow\rightarrow\downarrow)) = -010$
$-0-- \ \# \ \{10--, -010\} = \{000-, 00-1\}$

The required cubes obtained are $\overline{x}_1\overline{x}_2\overline{x}_3$ and $\overline{x}_1\overline{x}_2x_4$.

6.2.3 Observability don't cares

Observability don't cares (ODC) are associated with a gate in a synchronous network. An ODC is a condition that specifies when the behavior of a gate is not observable at any output of the network. Therefore, under such a condition the behavior of the gate doesn't matter, and this can be used to optimize the network.

Efficient methods do exist to calculate the ODC–set, or CODC– (compatible observability don't care) set [deM94] of each gate. Unfortunately, these techniques cannot be applied directly to optimize asynchronous logic. This is due to the fact that (synchronous) ODCs cannot deal with transitions. So, the output of a gate that is not observable in a synchronous network, can still introduce a hazard in an asynchronous network [Beer93]. An example is given in figure 6.5.

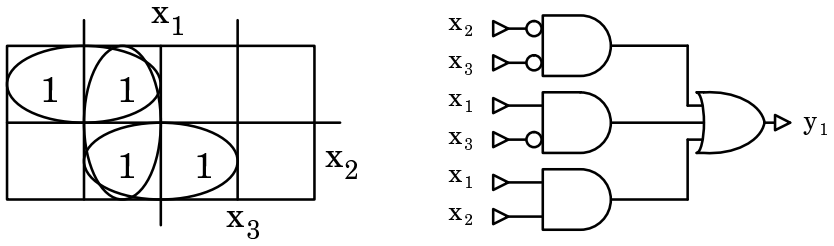


FIGURE 6.5. Exploiting 'synchronous' ODCs can cause a hazardous behavior

In figure 6.5, a node is depicted. If we calculate the ODC set for cube $x_1\overline{x}_3$, it is relatively clear that this gate can be removed completely. However, if the circuit is used to implement node transition $1 \uparrow 0 \rightarrow 1$ free from hazards, then removing cube $x_1\overline{x}_3$ might introduce a static–1 hazard.

We can still use the set of calculated ODCs to *nullify* a given node transition, meaning that the node transition is removed completely from consideration. If the transition cube of a node transition is completely covered by the set of ODCs, then this transition is nullified. For this node transition no required and privileged cubes are introduced, reducing the set of constraints for this node.

The set of ODCs can be used only in combination with a set of simulated node transitions. Since the ODC set is useless without this set of node transitions at each node, we do not calculate the synchronous ODC set at all. Instead, we make again use of simulation. From the left of the circuit to the right, we examine for each node transition belonging to the node under current investigation if we can safely replace the calculated output value by hazard value H. This way we will obtain the same result that we would obtain if we would have used the set of ODCs to nullify transitions.

By converting some node transitions to hazardous transitions (nullifying those node transitions), we remove the possibility of nullifying other node transitions. So the approach is sensitive to the sequence of applications of this step.

6.3 Results

We have taken some of the biggest known asynchronous benchmark circuits. Our starting point is a hazard–free two–level implementation of these circuits. To these circuits we have applied a set of known hazard–non–increasing transformations, including kernel extraction and common cube extraction. For this purpose we used the tool *log_decom* [Blaar92]. The nodes in the resulting circuits have been resynthesized with the described techniques. The results are depicted in table 6.1.

TABLE 6.1 Effect of using CDCs and ODCs on a set of asynchronous benchmark circuits.

Name	#Literals	#Literals CDC	#Literals CDC+ODC
bad–merge	70	70	70
cache–ctrl	1182	1109	1098
ircv	47	47	47
p1	455	434	430
p2	256	241	241
pe–send–ifc	115	115	115
pscsi	539	533	533
scsi	741	691	687
tsend	119	109	109

In column “Name”, the name of the asynchronous benchmark is given. In the second column, “#Literals”, the number of literals of the multi–level network is shown. Hazard–non–increasing transformations were used to obtain these

networks. In the third column, "#Literals CDC", the number of literals is shown when our synthesis method is allowed to use the set of local CDCs in combination with our current method of calculating the set of sub–transitions.

As we can see, applying the set of CDCs to optimize the above given set of asynchronous circuits does not result in a dramatic reduction in the number of literals. A maximum reduction of about 6% can be observed in the table. Only a small part of the gain observed in the table is due to the set of CDCs. The original two–level network was already optimized with the use of the set of CDCs expressed in its primary inputs. Most of the gain is due to the fact that multi–level synthesis is based on hazard–non–increasing transformations. This means that some transformations are actually sometimes *hazard–decreasing*. Therefore, by applying those transformations, the overall amount of constraints, in the number of privileged cubes/required cubes necessary in order to guarantee a hazard–free circuit, decreases, resulting in smaller circuits.

In the last column of the table we also take into account the set of ODCs. It turns out that the set of ODCs has almost no effect on the optimization process at all. This can be explained by the fact that almost all nullified transitions belong to an internal node that is near the primary inputs of the network. These nodes in general are very small, typically one cube. These nodes are not likely to be reduced.

Our approach is based on calculating a set of sub–transitions, which we use to optimize a network node. In order to guarantee a correct implementation, we demand that the behavior of each network node does not change. In future work, we hope to obtain better results by deriving sub–transitions, that are no longer based on this assumption.

6.4 Conclusions

We have investigated the effect of resynthesizing the nodes of an asynchronous multi–level circuit with the help of the set of CDCs and ODCs. For the biggest circuits we observe a total reduction of the size of the circuit of about 7%. The observed reduction is not due to the CDC set, certainly not due to the ODC set, but due to the fact that many of the hazard–non–increasing transformations applied are actually hazard–decreasing. Although the reduction in the number of literals is not very impressive, the reduction that is obtained is not expensive in terms of computation time. The biggest benchmarks were optimized in a few seconds.

In future work, we can improve the described methods, by deriving better methods to derive a set of sub–transitions. We believe that this will have a

dramatic impact on the number of literals. We will also investigate the effect of allowing some primary output variables to become hazardous. This is justified when it is known that the network connected to will allow for some hazardous output signals. This will probably occur in BMMs. In a network used in a BMM, the next state variables are fed back to the current state variables. ODC analysis might show that some of the current state variables are allowed to be hazardous. This means that the next state variables are also allowed to becomes hazardous. This might have a big impact, since transitions near the top of the network will become nullified, instead of near the bottom.

We have only looked at local transformations. In future work, one can also think of replacing and substituting part of the network, by e.g. taking into account the set of SDCs (satisfiability don't cares). The set of SDCs can be used, in combination with our hazard–free two–level minimizer, to find new substitutions, reducing the network even more.

Chapter 7 State Minimization

7.1 Introduction

A BMM can base future actions on past behavior. The past behavior is captured by the state of the BMM. The number of states in the specification is related to the complexity of the implementation of the BMM. That is: the less states there are, the smaller the implementation is. Although this proposition is true in general, especially for controller BMMs, counter–examples do exist [McCl86]. Since almost all BMM specifications are controller specifications, it makes sense to minimize the number of states necessary to describe a BMM in either a state graph, or a flow table.

The state minimization problem for synchronous machines is well understood, and has been described many times in the literature [Gras65], [Ung69], [Hill93], [deM94]. We will only outline the most important concepts. An important concept in the minimization of the number of states of a sequential circuit in general is the notion of an *incompatible state pair*. We will show that this notion needs some modification in order to guarantee a successful BMM–based synthesis system. This chapter summarizes the work by Nowick and Coates [Now94], and is only included for the sake of completeness.

7.2 Incompatible state pairs

Given a specification of a BMM, state minimization, or state reduction, can be regarded as the process of merging some states together to derive a new, more compact specification with less states. In order to do so, we must determine which states can be merged together. Consider the specification in figure 7.1.

	000	001	011	010	110	111	101	100
0	0,000							1,110
1	2,110							1,110
2	2,110			3,010				
3	4,011			3,010				
4	4,011	5,010						6,011
5	0,000	5,010						
6	6,011	7,110					6,011	6,011
7	2,110	7,110						

FIGURE 7.1. Example flow table

The flow table specification in figure 7.1 consists of 8 states. It is clear that states 0 and 1 cannot be merged together into one single state, because that state would have to implement a conflicting output behavior for input vector 000. In state 0 the implied output vector is equal to 000 whereas in state 1 the output vector is defined to be equal to 110. States 0 and 1 are said to be *output incompatible*.

DEFINITION 7.1

Two states are output incompatible if for some input vector both states define output vectors that are in conflict.

In order to merge a set of states, determining whether states are output compatible is not enough. Consider for example states 3 and 6 in figure 7.1. States 3 and 6 are in fact output compatible, but they cannot be merged into a single state. This fact can be observed by considering the next state function Δ for input vector 000 and for states 3 and 6. $\Delta(000, 3) = 4$, whereas $\Delta(000, 6) = 6$. If we would like to merge states 3 and 6 into a single state, s′, then $\Delta(000, s') = s''$, where $s'' = \{4, 6\}$. So another state s′′ must exist in which states 4 and 6 have been merged. But states 4 and 6 are output incompatible, and cannot be merged into a single state. Therefore states 3 and 6 can also not be merged into a single state. States 3 and 6 are said to be *next state incompatible*.

Definition 7.2
Two states are next state incompatible, if for some input vector both states imply next states that are output incompatible.

Based on definition 7.1 and 7.2, we can derive the following definition of incompatibility between two states:

Definition 7.3
Two states s_1 and s_2 are incompatible if they are output incompatible, or if the implied next states are incompatible.

This definition implies that the incompatibility relation of all state pairs can be derived by a fixed point calculation.

7.3 Maximal and prime compatibles

From the set of all incompatible state pairs it is possible to derive the set of all *maximal compatibles*.

Definition 7.4
A maximal compatible is a maximal set of states, such that each pair of states in this set is compatible.

Several methods have been proposed to derive the set of maximal compatibles [Ung69], [deM94]. The method proposed by Unger uses an incompatible state–pair chart to represent the set of all incompatible state pairs. This chart is used to derive the set of all maximal compatibles. The incompatible state–pair chart of the flow table in figure 7.1 is given in figure 7.2.

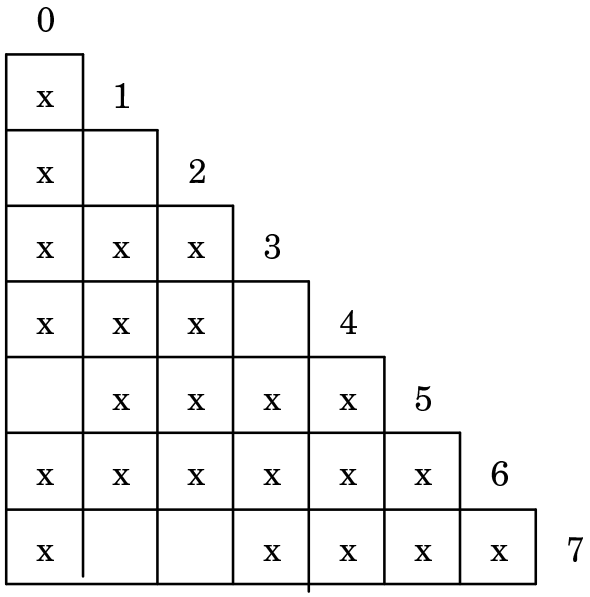


FIGURE 7.2. Incompatible state pair chart of figure 7.1

The symbol 'x' is used to indicate that two state pairs are incompatible. The maximal compatibles are constructed by evaluating each column, and the state corresponding to that column. We start with the rightmost column that contains at least one empty entry, and we finish with the leftmost column. The state pairs corresponding to the empty entries of the column we start with are added to a list C. In this example we add state pair $\{3, 4\}$ to list C. For each column we evaluate next, we add the state corresponding to that column to each set of states in C which is compatible with this state. In order to do so, we use the notion that a set of states is compatible with a given state, if each state in the set is compatible with the given state. This compatibility can therefore be checked by examining the non–empty entries of the incompatible state pair chart. We also add all the state pairs corresponding to empty entries in the current column of the incompatible state pair chart to list C. While adding sets of states to list C we make sure that we remove each set of states from list C that is covered completely by at least one other set of states. In the example, for column 2 we add state pair $\{2, 7\}$ to list C. So after column 2, list C = $\{\{2, 7\}, \{3, 4\}\}$. After column 1, list C = $\{\{1, 2, 7\}, \{3, 4\}\}$. Finally after column 0 has been dealt with, list C = $\{\{0, 5\}, \{1, 2, 7\}, \{3, 4\}\}$. After the evaluation of all columns is complete, we add the missing states. In this case state 6 is missing, and is therefore added. The final list of maximal compatibles becomes: C = $\{\{6\}, \{0, 5\}, \{1, 2, 7\}, \{3, 4\}\}$.

A minimum selection of the maximal compatibles of figure 7.1 is depicted in figure 7.3. In this case the selection is trivial, since every maximal compatible

is also *essential* since each maximal compatible covers a state not covered by another maximal compatible.

	000	001	011	010	110	111	101	100
0'	0,011	3,110					0,011	0,011
1'	1,011	2,010		1,010				0,011
2'	2,000	2,010						3,110
3'	3,110	3,110		1,010				3,110

FIGURE 7.3. Reduced flow table, specified in figure 7.1

The assignment to the four states in figure 7.3 is as follows: $0' = \{6\}$, $1' = \{3, 4\}$, $2' = \{0, 5\}$, $3' = \{1, 2, 7\}$.

In general, the number of maximal compatibles can be exponential in the number of states. A unate covering algorithm like mincov [Rud87] can be used to select a minimum number of maximal compatibles. However, this procedure cannot guarantee a correctly minimized machine. The problem is that a maximal compatible might imply a set of states which is not covered as a whole by any of the other maximal compatibles selected. A selected maximal compatible represents a new state in the reduced flow table. Each of the states belonging to the maximal compatible can imply a next state for each input entry. The set of implied next states for each input entry must be covered by another selected maximal compatible. The problem of selecting the minimum number of maximal compatibles therefore is a binate covering problem.

In our implementation we ignore the implications by maximal compatibles. We solve the resulting unate covering problem, and if there are implications that are not satisfied by the selected set of maximal compatibles, we add additional maximal compatibles until the set of maximal compatibles becomes closed under implication. It turns out that in many examples, solving the unate covering problem suffices.

Even if we would have solved the binate covering problem exactly, we still would not have been able to guarantee a minimum solution in the number of states. A selected maximal compatible might cover more states than required by implications of other selected compatible sets. This could lead to additional

implications that must be satisfied. In exact state minimization the concept of maximal compatibles is therefore refined into the notion of *prime compatibles* [Gras65] which are used instead. In our implementation, like in Nowick's implementation [Now94], we only make use of maximal compatibles.

7.4 Asynchronous incompatible state pairs

Let us consider the flow table of figure 7.4 (reprinted from figure 3.7).

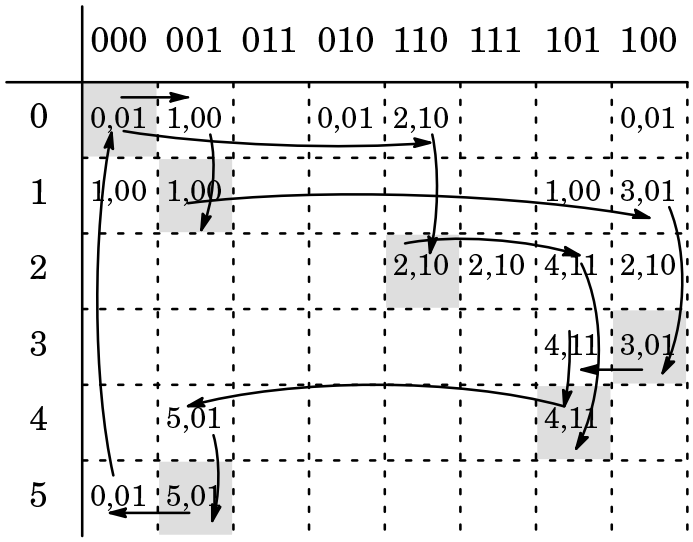


FIGURE 7.4. Flow table representation of "*Bad–Merge*"

A minimum flow table representation of the flow table in figure 7.4, consisting of three states, is shown in figure 7.5. Here $0' = \{0, 3\}$, $1' = \{1\}$ and $2' = \{2, 4, 5\}$.

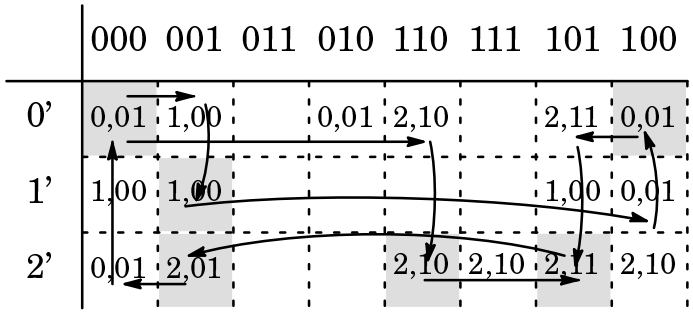


FIGURE 7.5. Minimal flow table, specified in figure 7.4

Let us examine state 0′ in the reduced flow table. In figure 7.6, this state, and its transitions, are shown in detail.

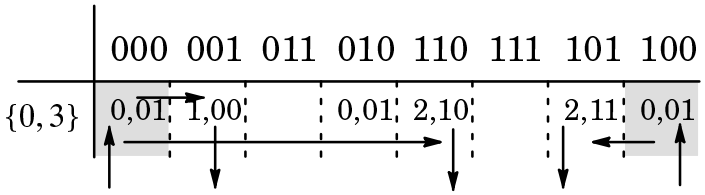


FIGURE 7.6. State 0' of the flow table in figure 7.5

Three input transitions are visible in figure 7.6. For now, let us leave out the states and the state variables used to encode each state, and let us focus only at the output variables. In that case, the three input transitions can be described by: 00 ↑→ 0 ↓; ↑↑ 0 →↑↓ and 10 ↑→↑ 1. In chapter 4 we saw that transitions introduce a set of required and privileged cubes. Consider transition ↑↑ 0 →↑↓ This transition introduces a privileged cube $(--0, 000)$ for the second output variable. Transition 10 ↑→↑ 1 introduces a required cube 10– for the second output variable. This required cube illegally intersects privileged cube $(--0, 000)$. This is in violation with theorem 4.3, meaning that it is possible that no hazard–free 2–level implementation exists for the given set of transitions belonging to the minimized flow table, whereas such a solution *does* exist for the non–minimized flow table.

THEOREM 7.1

Each valid BMM specification can be implemented free from hazards if no state minimization is applied.

PROOF

This theorem was proven by Nowick in [Now94]. It is not difficult to see that this theorem holds. In chapter 3 two restrictions are given that must be obeyed by each valid BMM specification: the unique entry point and the maximal set property. One of these restrictions is that each state in the state graph, or flow table, representation of the BMM should have a unique entry point. Therefore, if a required cube introduced by some transition intersects a privileged cube, we can guarantee that the intersection is legal, since the starting point will be included in the intersection. □

Since a non–minimized version of a BMM specification can be implemented dhf–free, it makes sense to reconsider the state reduction process. The notion of incompatibility, by definition 7.3, does not take into account the implementability constraint defined in theorem 4.3. This is because the

incompatibility relation only considers input *vectors,* and not input *transitions*.

If theorem 4.3 is violated, we cannot guarantee that the hazard–free minimizer in chapter 4 will generate a valid hazard–free solution. Therefore we have to modify the original incompatibility relation in definition 7.3.

DEFINITION 7.5
Two states are said to be dhf–incompatible if by merging these two states into one state, a privileged cube is intersected illegally by a required cube.

DEFINITION 7.6
Two states s_1 and s_2 are incompatible if they are output incompatible, dhf–incompatible, or if the implied next states are incompatible.

The only question remaining is how we can determine whether merging two states will result in an unimplementable set of transitions, i.e. in a privileged cube being intersected illegally by a required cube. As we saw, privileged cubes are introduced whenever an output or state variable changes from 1 to 0. We do know which output variables change, and which ones will remain stable. However, this is not the case for the state variables.

State minimization takes place before state assignment. Therefore, all states are still symbolic: they have not been encoded yet. Recently, some work has been done to combine both state minimization and state encoding in one single step [Fuhr97]. Unfortunately, this approach is still computationally expensive. Here we do not take this approach. We consider state minimization and state assignment as two separate steps.

Because the states are all symbolic, we don't know how many state variables there are. Even if we knew, then for each state transition we don't know which of these state variables are going to change, and which will remain stable. Therefore, we cannot know how many privileged and required cubes are necessary to implement the behavior of all state variables free from hazards.

So, in order to determine if two states are compatible or incompatible, we must be able to guarantee that they are compatible regardless of the number and the encoding of the state variables. In order to do so, we'll have to assume a worst–case scenario. We saw in definition 7.5 that two states are dhf–incompatible if, due to the merger, a privileged cube is intersected illegally by a required cube. In order to assert that this will never happen after state assignment, we introduce privileged cubes for each state variable, and for each input transition. I.e. we assume that after each input transition, each state variable will make a change from 1 to 0. This corresponds with our worst–case scenario.

Although the above assumption is valid, in that it can be used to reduce the number of states of a BMM, it is possible to relax this assumption. It is not always necessary to assume that all state variables make 1 to 0 changes. Consider figures a and b in figure 7.7.

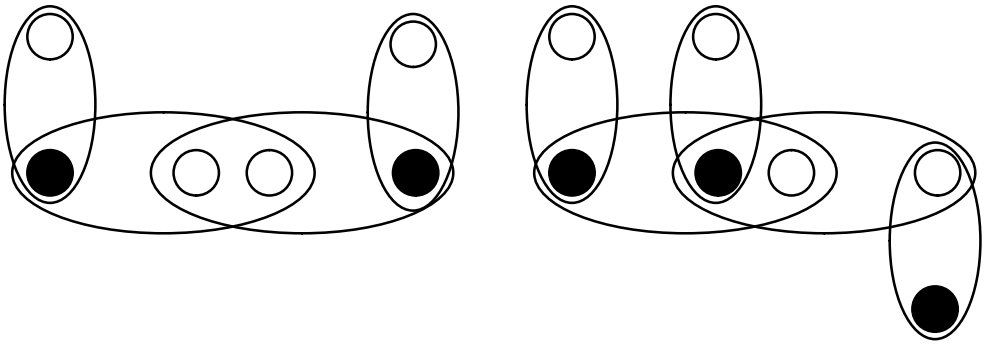


FIGURE 7.7. Nullified transitions

Here, we depict some possible transitions in a symbolic way that resembles a flow–table–based representation. The vertical ellipsoids represent current state transitions. The horizontal ellipsoids represent input transitions. A current state transition starts with a white dot, being the end point of a certain input transition, and it ends with a black dot, the entry point of a given state and the starting point of another input transition.

Consider the situation in figure 7.7a. Here, two input transitions have been merged. In the unminimized machine we can associate each input transition with a certain state. Suppose we know that the two states associated with these two input transitions are output compatible and next state compatible. The end point of each input transition is located within the transition cube corresponding to the other input transition. The next state that is implied by the end point of each input transition must therefore be compatible with the state associated with the other input transition. We can therefore *nullify* the current state transition. I.e. we can change the implied next state value at the end point of the input transition to the associated state of the other input transition. For these input transitions we can therefore assume that the state variables will remain stable after the input transition has been completed. For these input transitions, we do not need to introduce privileged cubes for each state variable. In figure 7.7b a similar situation is shown.

In [Now94] Nowick shows that, in determining which states are dhf–incompatible, it suffices to consider the worst–case behavior of the symbolic states

of the machine. He claims that we do not need to consider the behavior of the outputs, since all incompatible state pairs introduced by only considering the outputs are all contained by the set of incompatible state pairs introduced by considering the symbolic states.

With the above given observations, we can guarantee that the set of transitions, after state minimization are implementable. Note that we do not claim that the incompatibility relation is optimal: if our incompatibility relation claims that two states are incompatible, then it is still possible that the minimized machine will generate a set of implementable transitions.

With our modified notion of incompatible state pairs, we derive that states 0 and 3 of figure 7.4 are incompatible. The resulting reduced flow table consists of four states, and can be observed in figure 7.8. In this reduced flow table the reduced states correspond with maximal compatibles: $0' = \{0\}$, $1' = \{1\}$, $2' = \{2\}$ and $3' = \{3, 4, 5\}$.

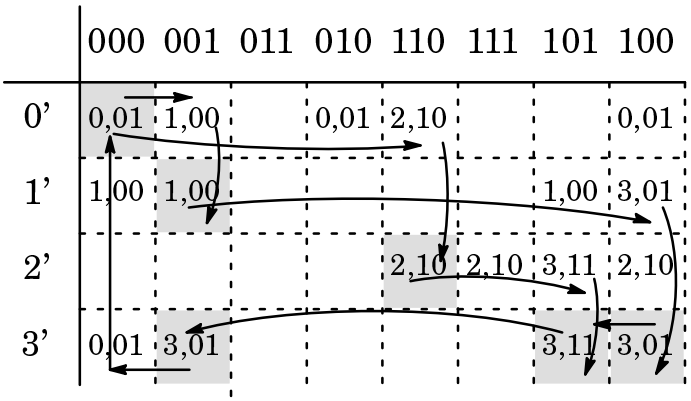


FIGURE 7.8. Minimum hazard–free flow table, specified in figure 7.4

7.5 Conclusions

In this chapter we provided an overview of state minimization for BMMs. We showed that by modifying the incompatibility relation, we can guarantee that the resulting minimized machine can be implemented with hazard–free two–level logic. The resulting minimized machines will possibly contain more states than the machines that are reduced by a normal 'synchronous' minimizer.

Chapter 8 State assignment

8.1 Introduction

State assignment is the step that assigns a binary code to the states of a given state machine specification. The step is usually performed on a minimized state machine. For synchronous machines, state assignment can be as simple as assigning to each symbolic state a unique encoding. The state assignment problem is more complex for asynchronous machines. As is the case in synchronous machines, each encoding must be unique, but the encoding must also be such that no *critical races* can occur. The constraints necessary for asynchronous (Huffman) machines were originally derived by Tracey [Trac66].

State assignment can have a big impact on the complexity of the final implementation. Therefore, methods have been derived for synchronous machines that will lead to efficient two–level and multi–level implementations. One of these methods is based on s*ymbolic minimization* [deM85], [deM86]. The description of the machine is minimized before the states are encoded. In this chapter we will show how symbolic minimization can be used to derive a valid state encoding for BMMs.

We start with an introduction of symbolic minimization and state assignment for synchronous machines in section 8.2. In section 8.3 we discuss the problem of hazard–free symbolic minimization for BMMs. As we saw in chapter 4, the binary two–level minimization step is constrained by a set of required and privileged cubes. We show that during symbolic minimization of BMMs we also encounter a set of *symbolic* required and privileged cubes. In section 8.4 we investigate the state assignment problem for asynchronous machines in general. As we will see, the state assignment must be such that we avoid anomalies called critical races. In order to avoid these anomalies, additional constraints are required. In section 8.5 we show how we can merge the constraints imposed by hazard–free symbolic minimization, discussed in section 8.3, with the constraints necessary in order to avoid critical races, discussed in section 8.4. After having derived a valid state encoding, the BMM has to be instantiated: the found encoding is used to generate a binary description of the machine. This is discussed in section 8.6. In section 8.7 we

propose some improvements that reduce the number of state encoding constraints introduced in section 8.5 [Rut97c]. This results in significant saving in the number of state variables needed to instantiate a BMM. Results are given in section 8.8. In section 8.9 we show how our improvements can be incorporated in Fuhrer's work [Fuhr95]. Conclusions are drawn in section 8.10.

8.2 Synchronous state assignment

In a synchronous machine, each state must be assigned a unique binary encoding. This is the only constraint that must be satisfied. This means that there is a lot of freedom in choosing a correct encoding. The chosen encoding can have a great impact on the final area of the implementation. An example is given in figure 8.1.

	00	01	10	11
0	0,0	2,0	0,1	0,1
1	1,1	1,1	1,0	0,1
2	0,0	2,0	1,0	3,1
3	1,1	1,1	0,1	3,1

(a)

S	Code
0	00
1	01
2	10
3	11

(b)

FIGURE 8.1. A 'synchronous' flow table

In figure 8.1a, a flow table is depicted. If we take the binary encoding in figure 8.1b to encode each state, 8 cubes are needed to describe the machine. This description is shown in figure 8.2. The implicants in figure 8.2 consist of four input variables and three output variables. The four input variables are partitioned into two primary input variables, I and two current state variables, CS. The three output variables are partitioned into two next state variables NS and one output variable O. In figure 8.2, the variables belonging to the input part of each implicant are separated from the variables belonging to the output part with a vertical line. We will show that by modifying the code, a smaller implementation can be obtained.

I	CS	NS	O
11	1–	11	1
0–	–1	01	1
––	11	00	1
1–	00	00	1
–0	01	01	0
1–	10	01	0
01	–0	10	0
–1	–1	00	1

FIGURE 8.2. Minimum two–level solution of the machine in figure 8.1a, by applying the state encoding in figure 8.1b

In [deM85] and [deM86], de Micheli relates the state assignment problem to the problem of symbolic two–level minimization of a machine. In this approach, the symbolic description of the machine is minimized first. In order to do so, the machine is converted to a symbolic description. This is performed by describing the machine as a set of symbolic implicants, where a symbolic implicant is defined as follows:

DEFINITION 8.1

A symbolic implicant is defined by the 4–tuple: <I, CS, NS, O>, where $I \in \{0, 1, -\}^n$, $CS \subseteq C$, $NS \in C \cup \{\epsilon\}$ and $O \in \{0, 1\}^m$.

A symbolic implicant consists of an input part I a current state part CS a next state part NS, and an output part O. Here, n is the number of inputs, m is the number of outputs, and C represents the set of states. Value ϵ is used to express that a symbolic implicant does not imply any next state.

Consider again the example in figure 8.1a. In this example, $C = \{0, 1, 2, 3\}$. A symbolic description of the machine, that is: a set of symbolic implicants, can be derived by writing down each entry in the flow table:

I	CS	NS	O
00	0	0	0
01	0	2	0
10	0	0	1
11	0	0	1
00	1	1	1
01	1	1	1
10	1	1	0
11	1	0	1
00	2	0	0
01	2	2	0
10	2	1	0
11	2	3	1
00	3	1	1
01	3	1	1
10	3	0	1
11	3	3	1

FIGURE 8.3. Symbolic description of the machine in figure 8.1a

This symbolic description can be minimized, resulting in the following description:

10	{0,3}	0	1
11	{2,3}	3	1
0–	{1,3}	1	1
11	{0,1}	0	1
01	{0,2}	2	0
10	{1,2}	1	0
00	{0,2}	0	0

FIGURE 8.4. Minimized symbolic description of the machine in figure 8.1a

The minimized symbolic description of the machine consists of 7 implicants. Consider the first implicant in figure 8.4. This implicant states that for input combination 10 and in case the machine is in state 0, or in state 3, the implied next state is state 0 and the implied output value is 1.

The description in figure 8.4 is smaller in the number of implicants than the description of the machine in figure 8.2, which requires 8 implicants.

Therefore, it makes sense to instantiate the symbolic machine, and use the resulting description instead. This is done by replacing the symbolic next state in each implicant by the corresponding binary encoding and by replacing the set of states in the current state part by the smallest cube that covers the encoding of each state in that set (we can consider an encoding of a state as a minterm expressed in the state variables introduced). In case that a symbolic implicant does not imply any next state (which happens when the next state assumes value ϵ), the next state is replaced by an all–zero encoding.

Suppose we would use the state encoding in figure 8.1b to instantiate the symbolic description in figure 8.4. Consider the first symbolic implicant in figure 8.4. By using the state encoding in figure 8.1b, this implicant becomes 10 –– 00 1. Note that this implicant can be enabled regardless the current state the machine is in. Whenever the first input is 1 and the second input is 0, the output is implied to be 1. As we can observe in the flow table in figure 8.1, this behavior is wrong. So, a minimized symbolic description of a machine does put constraints on the state encoding used to instantiate the machine.

A valid state encoding must be such that each instantiated symbolic implicant can only imply its next state and output when the machine is in one of the states belonging to the current state part of the symbolic implicant. So, if we again examine the first symbolic implicant in figure 8.4, then a valid encoding must be such that this implicant can only imply next state 0 and output value 1, when the machine is in state 0 or 3. In order to guarantee this, the smallest cube that covers the encoding of each state in the current state part should not cover an encoding that represents a state that is not present in the current state part.

This can only be accomplished by demanding that this smallest cube is at least distance–1 [Rud87] apart from each encoding representing a state not present in the current state part. Two cubes are distance–1 apart if one variable assumes literal form 0/1 in the first cube and the 'opposite' literal form 1/0 in the other cube. So, there should be at least one state variable that assumes literal form 0/1 in all encodings (or minterms) belonging to the set of states in the current state part, and the opposite form 1/0 for an encoding representing a state not represented in the current state part.

In our example, the state encoding must be such that a state variable assumes literal form e.g. 0 in states 0 and 3, and the opposite value 1 in state 1. Also, there must be one state variable that assumes e.g. value 0 for state 2, and the opposite value in states 0 and 3. Constraints formulated like this are called *face constraints* [Vill90]. They can be expressed as *dichotomies* [Trac66].

DEFINITION 8.2
A dichotomy consists of two non–empty disjoint subsets of the total set of states C. It takes the form (A, B).

Dichotomies express the constraint that there should be at least one state variable that distinguishes the states belonging to set A from the set of states belonging to set B.

The constraints generated by the first symbolic implicant of figure 8.4 can be captured by dichotomies $(\{0,3\},\{1\})$ and $(\{0,3\},\{2\})$. A valid encoding can be obtained by simply turning each dichotomy into one state variable. However, the symbolic implicants of a certain machine can generate many dichotomies, leading to many state variables. The machine in figure 8.4 will generate 14 dichotomies: two for each symbolic implicant. Fortunately, it is possible to merge dichotomies. The two dichotomies generated by the first symbolic implicant in figure 8.4, for example, can be merged into a single dichotomy $(\{0,3\},\{1,2\})$. By satisfying this dichotomy, the two earlier dichotomies are also satisfied. In order to determine which dichotomies can be merged a compatibility relation is introduced.

DEFINITION 8.3
A dichotomy (U,V) is compatible with dichotomy (X,Y), expressed as $(U,V) \sim (X,Y)$, if $(U \cap X = \emptyset \wedge V \cap Y = \emptyset) \vee (U \cap Y = \emptyset \wedge V \cap X = \emptyset)$.

So, a dichotomy is not compatible with another dichotomy if two states s_0 and s_1 are covered by one of the subsets of one of these dichotomies, while in the other dichotomy s_0 and s_1 each belong to a different subset. Two compatible dichotomies can be merged into one dichotomy.

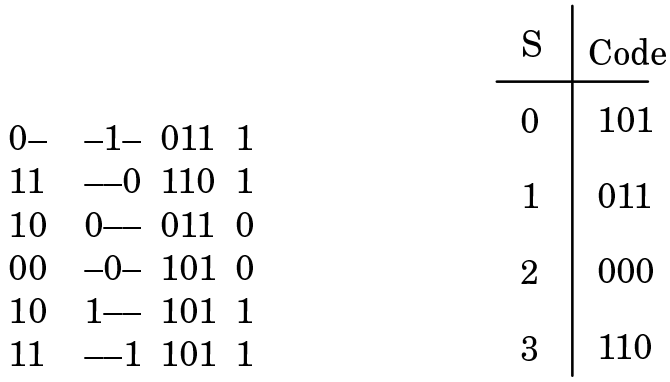
DEFINITION 8.4
A dichotomy (U,V) is covered by another dichotomy (X,Y) if $(U \subseteq X \wedge V \subseteq Y) \vee (U \subseteq Y \wedge V \subseteq X)$.

DEFINITION 8.5
A prime dichotomy is a dichotomy that is not covered by any other dichotomy.

The procedure of minimizing the number of state variables consists of constructing the set of all *prime dichotomies,* and by selecting from this set, which can be quite large, the smallest number of prime dichotomies, which cover all original dichotomies. This selection is again done by means of a unate covering algorithm. More details can be found in [Ash92] and [Sald91].

The smallest state assignment, in the number of state variables, that satisfies all the constraints imposed by the symbolic description of figure 8.4 is shown

in figure 8.5b. We can use this state encoding to instantiate the symbolic description of figure 8.4. After applying an additional two–level minimization step, the result is depicted in figure 8.5a.



0–	–1–	011	1
11	—0	110	1
10	0—	011	0
00	–0–	101	0
10	1—	101	1
11	—1	101	1

(a)

S	Code
0	101
1	011
2	000
3	110

(b)

FIGURE 8.5. Minimum two–level implementation of figure 8.4, by applying the depicted state encoding.

Note that the final implementation consists of 6 implicants, whereas the symbolic description consists of 7 implicants. The symbolic description is therefore an upper bound. The last two–level minimization step is able to reduce the description of the machine even further. This is because the effect of symbolic minimization is "*to group states together that are mapped for some input space onto the same next state, and assert the same output values*" [deM85]. Symbolic minimization does not take into account the fact that a state can be assigned an all–zero code, which might reduce the description of the machine. This was the case in the above description of the machine. Furthermore, symbolic minimization cannot take into account the fact that the encoding of some states are covered by the encoding of other states. Also, the fact that an encoding might be implied by multiple implicants by disjunction is not taken into account [Ash92]. Therefore it is unlikely that symbolic minimization will result in an exact minimum two–level implementation of a state machine. Exact algorithms [Ash92] are however very expensive, and are feasible for the smallest benchmark circuits only. Furthermore, as can be observed in [Vill90], heuristic methods that can also take into account some of the additional optimization possibilities described above, do in general not generate much smaller circuits, compared to plain symbolic minimization. Therefore, we will only focus on symbolic minimization.

8.3 Hazard–free symbolic minimization

Since symbolic minimization is capable of reducing the description of a synchronous machine, we would also like to apply symbolic minimization in order to reduce the description of BMMs. We modify symbolic minimization in order to guarantee a hazard–free instantiation of the symbolic solution. This is a choice we make, it is not a necessity. It is perfectly valid to use the constraints generated by a synchronous symbolic minimization step, as long as the encoding is such that no critical races can occur (the derivation of the minimum number of constraints needed to ensure a critical–race–free encoding is discussed in the next section). In our approach we target a hazard–free symbolic solution in an attempt to derive an upper bound of the final binary solution, just like in a synchronous state assignment.

Before we examine the hazardous behavior of a symbolic description of a BMM, we first formalize the different kind of transitions that we can encounter in a symbolic description of a BMM.

As we saw in section 3.4, the behavior of a BMM can be described by two kind of transitions:

- Input transitions. During input transitions the environment applies changes to input variables. After all changes have been applied, the BMM will calculate the new next state, and will apply the appropriate changes to the output variables. Input transitions can therefore be modelled as follows:

DEFINITION 8.6

An input transition is a 4–tuple: $<I, CS, NS, O>$, where $I \in \{0, 1, \uparrow, \downarrow\}^n$, $CS \in C$, $NS \in C$ and $O \in \{0, 1, \uparrow, \downarrow\}^m$.

- Current state transitions. During current state transitions the BMM evaluates the changes applied to its state variables. During these changes the BMM will maintain the proper output values, and will continue to imply the correct (new) next state. The environment is not allowed to apply new changes to the input variables as long as the machine is evaluating the changes to its state variables. Current state transitions can therefore be modelled as follows:

DEFINITION 8.7

A current state transition is defined as: $<I, CS, NS, O>$, where $I \in \{0, 1\}^n$, $CS \subseteq C$, $NS \in CS$ and $O \in \{0, 1\}^m$.

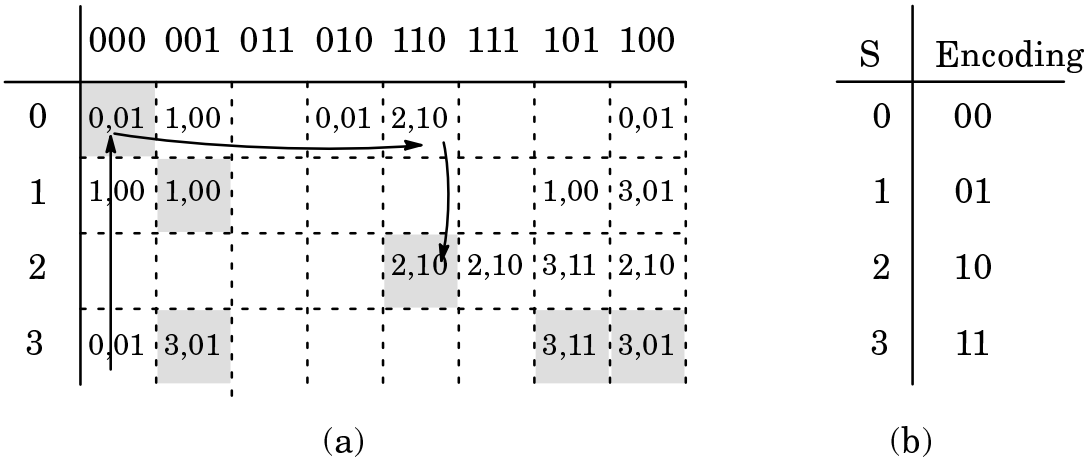


FIGURE 8.6. Example flow table

As an example, consider the flow table in figure 8.6a. In this table an input transition is shown in the row belonging to state 0 and a current state transition is shown for input column 110. The input transition is: <↑↑ 0, 0, 2, ↑↓>. The current state transition for input column 110 is: <110, {0, 2}, 2, 10>.

Applying symbolic minimization without any modification to a symbolic BMM specification might lead to a set of symbolic implicants that are hazardous after state assignment, and will therefore be removed by any binary minimization step that will follow. An example is shown in figure 8.7.

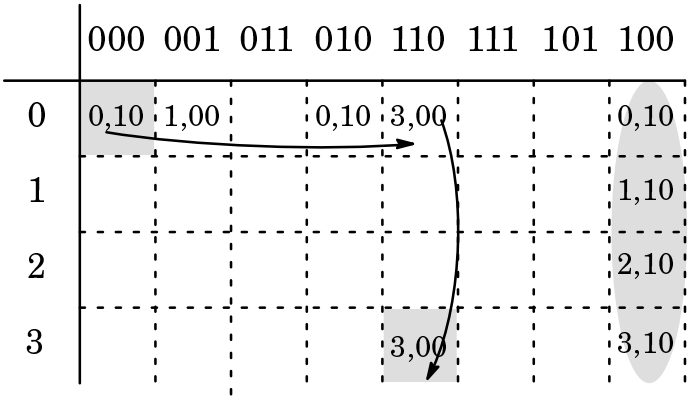


FIGURE 8.7. Flow table with a hazardous symbolic implicant

In figure 8.7, part of a flow table is shown. In the flow table we can observe two transitions, and an implicant A described by: <100,{0,1,2,3},ϵ,10>. Implicant A is shown as a shaded ellipsoid. It is easy to see that implicant

A can temporarily turn on during the input transition. The second output might therefore show a dynamic 1–to–0–hazard. Implicant A will therefore not occur in a binary minimized implementation of the BMM, since it is not dhf–free. So a 'synchronous' symbolic minimized version of a BMM does not correspond well with the final binary implementation, as is the case for synchronous machines. In order to avoid this as much as possible, we modify the symbolic minimization step in order to take into account possible hazardous situations after binary instantiation. This is done by the introduction of a set of symbolic required and privileged cubes.

From the set of input transitions and current state transitions, we can determine both the set of required cubes and the set of privileged cubes. Both sets can be partitioned into two subsets: required/privileged cubes necessary to implement output variables free from hazards, and required/privileged cubes necessary to implement the (still unknown number of) state variables free from hazards. We first consider the output variables.

For each output variable that remains stable at one during an input transition, a required cube is created. The input part I of this required cube simply is the transition cube, defined in definition 4.1. So, for input transition <10↑,0,2,↑1>, required cube <10–,0,ϵ,01> is created.

For each output variable that changes from 1 to 0, after all input changes have been applied, a privileged cube is created. The input part of the body of such a privileged cube p^c is again equal to the transition cube. Therefore, for input transition <↑↑0,0,2,↑↓> one privileged cube is introduced, namely (<––0,0,ϵ,01>,<000,0,ϵ,01>). Note that the privileged cube does not imply any next state. The output part of both the body and starting point of the privileged cube indicate for which output the privileged cube is valid.

Note that the above described set of required and privileged cubes is not complete. For example, input transition <↑↑0,0,2,↑↓> will also introduce required cubes <0–0,0,ϵ,01> and <–00,0,ϵ,01>.

Now, let us consider the state variables. After an input transition, a current state transition might occur. However, since the states of the machine at this stage are still symbolic, we don't know how many state variables there are, and which state assignment has been chosen. In order to guarantee a hazard–free instantiation of each symbolic implicant, we introduce a symbolic privileged cube for each input transition that is followed by a current state transition. Consider for example input transition <↑↑0,0,2,↑↓>. For this transition we introduce privileged cube (<––0,0,0,00>,<000,0,0,00>). This privileged cube will make sure that during the input transition no symbolic implicant is introduced that can temporarily turn on.

During current state transitions, the output values and the implied next state must remain stable. Therefore current state transitions will not introduce a set of privileged cubes. Current state transitions will however introduce a set of required cubes. For example for current state transition <110,{0,2},2,10> two required cubes are introduced, namely <110,{0,2},2,00> and <110,{0,2},ϵ,10>.

After all symbolic required and privileged cubes have been generated, a symbolic hazard–free two–level solution is generated. For this purpose we make use of a modified version of Dill/Nowick's algorithm [Now92], proposed by Fuhrer [Fuhr95]. In this algorithm, the set of all symbolic prime implicants is generated, which is rendered hazard–free by an a–posteriori hazard–removal filter. Efficient exact and heuristic techniques do no yet exist for this purpose, as is the case for binary hazard–free minimizers, although due to its multi–valued nature it is possible to use the exact and heuristic algorithms described in chapter 4. Some modifications are however necessary. In this thesis, we have applied the classic exact method.

8.4 Critical races

The now minimized symbolic description of a BMM does put constraints on the encoding, just like the minimized symbolic description of a synchronous machine. However, these constraints are not enough in order to avoid *critical races* [Ung69].

A race does occur if during a current state transition two or more state variables change in value.

DEFINITION 8.8
A race is called critical if the behavior of the machine depends on the order in which the state variables change in value.

As an example, consider the flow table in figure 8.6a. Observe that for input column 000 a current state transition is specified: <000,{0,3},0,01>. If we would use the state encoding depicted in figure 8.6b, a critical race can occur in this current state transition. If the change of the first state variable is observed by the BMM, before the change of the second state variable, the machine might end up in the wrong stable state, namely state 1. In order to avoid this anomalous behavior, we must introduce additional constraints, to be obeyed by the encoding. These constraints are again expressed with the help of dichotomies. In this case, adding dichotomy ($\{0, 3\}, \{1\}$) will make sure that during the current state transition state 1 is not entered. This observation can be generalized as follows:

THEOREM 8.1

A necessary constraint for a critical–race–free encoding is that the state encoding satisfies the following dichotomies: For each current state transition T occurring in an input column i, a dichotomy (CS(T), s) is generated, if entry (i, s) is specified in the flow table, and s $\notin$ CS(T) [Trac66].

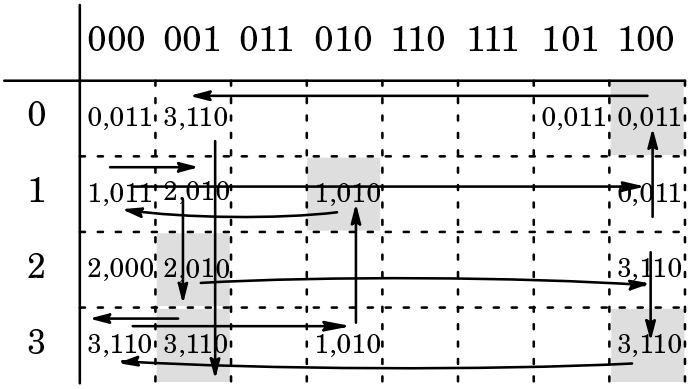


FIGURE 8.8. Flow table that demonstrates the shortcoming of theorem 8.1

The constraints described by theorem 8.1 are necessary, but not sufficient as can be demonstrated by the flow table in figure 8.8. Consider input column 100 in this flow table. According to theorem 8.1, dichotomies ({0, 1}, {2}), ({0, 1}, {3}), ({2, 3}, {0}), and ({2, 3}, {1}) must be covered. The encoding in figure 8.9 does obey all these constraints.

S	Code
0	0010
1	0001
2	1000
3	0100

FIGURE 8.9. An encoding that leads to critical–race behavior of the flow table in figure 8.8

The encoding in figure 8.9 does avoid that during one of the following two current state transitions, <100,{0,1},0,011> and <100,{2,3},3,110> any other

stable state than one from the set of states $\{0, 1\}$ and $\{2, 3\}$ respectively is entered. However, the encoding shown can not avoid that the two current state transitions share common *intermediate states*. An intermediate state is a state that is entered during a current state transition, but which does not correspond with any encoded symbolic state. In this case, both current state transitions share intermediate state 0000. For this intermediate state conflicting specifications exists. The first current state transition specifies a different next state and different output values than those specified by the other current state transition. Clearly, additional constraints are needed. The sharing of intermediate states by two current state transitions can be avoided by adding an additional dichotomy that explicitly distinguishes the two current state transitions. In this case, adding dichotomy $(\{0, 1\}, \{2, 3\})$ to the set of constraints will guarantee a critical–race–free encoding for the two current state transitions. This observation can be generalized into the following theorem:

THEOREM 8.2

A critical–race–free encoding must cover all dichotomies (CS(T), CS(U)), where T and U are two current state transitions in the same column of a flow table specification, both of which imply two different next states [Trac66].

All the constraints necessary to ensure a critical–race–free implementation of the flow table in figure 8.8 can be described by the following set of dichotomies:

- Due to theorem 8.1:

For column 001: $(\{0, 3\}, \{1\})$, $(\{0, 3\}, \{2\})$, $(\{1, 2\}, \{0\})$ and $(\{1, 2\}, \{3\})$.
For column 100: $(\{0, 1\}, \{2\})$, $(\{0, 1\}, \{3\})$, $(\{2, 3\}, \{0\})$, and $(\{2, 3\}, \{1\})$

- Due to theorem 8.2:

For column 001: $(\{0, 3\}, \{1, 2\})$
For column 100: $(\{0, 1\}, \{2, 3\})$

The other input columns do not give rise to the introduction of additional constraints in the form of dichotomies. A minimum encoding that covers all the depicted dichotomies is given by the two dichotomies $(\{0, 3\}, \{1, 2\})$ and $(\{0, 1\}, \{2, 3\})$. Theorems 8.1 and 8.2 are also known as the *Tracey constraints* [Trac66]. Any critical–race–free encoding cannot be smaller than the smallest encoding that obeys all Tracey constraints.

8.5 Critical–race–free encoding of a symbolic minimized BMM

In the previous section we described the minimal constraints necessary to avoid critical races. In addition to these constraints, we would also like to take

advantage of the result obtained by our hazard–free symbolic minimization step. It is tempting to just add the constraints generated by this step. However, this approach does invalidate the symbolic minimized solution as is demonstrated by figure 8.10.

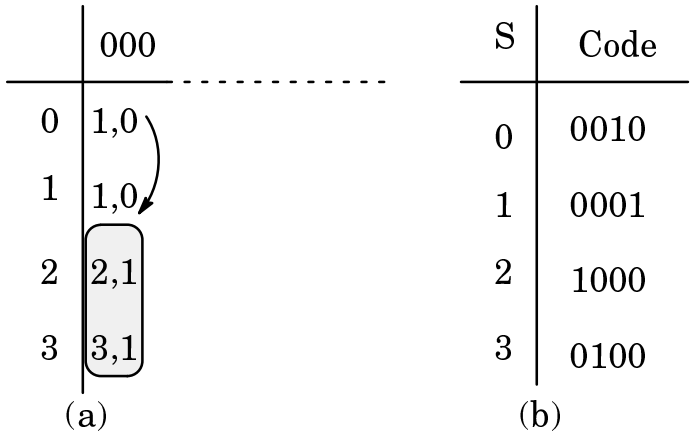


FIGURE 8.10. Flow table demonstrating the need for additional constraints after symbolic minimization

In figure 8.10a part of a flow table is depicted. In this flow table a current state transition is shown: <000,{0,1},1,0>. The shaded ellipsoid represents a symbolic implicant, <000,{2,3},ϵ,1>, generated by our hazard–free minimizer. According to theorems 8.1 and 8.2, we have to add dichotomies ({0, 1}, {2}) and ({0, 1}, {3}) to the set of constraints. The constraints due to symbolic minimization are represented by dichotomies: ({2, 3}, {0}) and ({2, 3}, {1}). The encoding in figure 8.10b does obey all these constraints. If we use this encoding, and if we instantiate the symbolic implicant, we get implicant <000,––00,0000,1>. This implicant might turn on during the current state transition since both the current state transition and the symbolic implicant share intermediate state 0000. Therefore, in this example the output might temporarily turn on, and thus might show a hazardous behavior.

Based on this observation we conclude that we want to avoid that current state transitions and incompatible symbolic implicants share common intermediate states. In order to do so, we must first define incompatibility.

DEFINITION 8.9

A symbolic implicant A is incompatible with a current state transition T if:

$$I(A) \cdot I(T) \neq 0 \wedge$$
$$((NS(A) \neq \epsilon) \wedge (NS(A) \neq NS(T)) \vee (\exists k : O(A)_k = 1 \wedge O(T)_k = 0)$$

So, if an implicant A implies a different next state, or enables an output not enabled by transition T, while sharing the same input entries with this transition it is said to be incompatible with transition T.

With the help of definition 8.9, we can formulate the extra constraints necessary as follows:

THEOREM 8.3

A critical–race–free encoding must cover each dichotomy (CS(T), CS(A)), where T is a current state transition, and A is an incompatible symbolic implicant [Fuhr97].

The constraints represented by theorem 8.3 cover all the constraints represented by theorem 8.2. This is because every current state transition must be covered by at least one symbolic implicant in the solution. Therefore, each constraint that distinguishes between two current state transitions is covered by at least one constraint generated according to theorem 8.3. Also, the constraints imposed by the symbolic solution itself cover all the constraints necessary according to theorem 8.1 for a similar reason. For a detailed proof, see [Fuhr97].

The combination of symbolic constraints in combination with the constraints represented by theorem 8.3 will guarantee a critical–race–free solution. As an example, consider again the flow table specification in figure 8.8. A (hazard–free) symbolic minimized version of this machine is shown in figure 8.11.

–1–	{0,1,2,3}	{1}	010
1—	{0,1}	{0}	011
1—	{2,3}	{3}	110
——	{0,1,3}	{ϵ}	010
—0	{0}	{0}	011
0–1	{0,3}	{3}	110
–0–	{3}	{3}	110
—1	{1,2}	{2}	010
–00	{0,1}	{ϵ}	011
0–0	{1}	{1}	010
0—	{2}	{2}	000

FIGURE 8.11. Symbolic hazard–free minimized description of the flow table in figure 8.8

The constraints due to the symbolic minimization process itself are: $(\{0,1\},\{2\})$, $(\{0,1\},\{3\})$, $(\{2,3\},\{0\})$, $(\{2,3\},\{1\})$, $(\{0,1,3\},\{2\})$, $(\{0,3\},\{1\})$, $(\{0,3\},\{2\})$, $(\{1,2\},\{0\})$ and $(\{1,2\},\{3\})$.
The constraints due to theorem 8.3 are: $(\{0,1\},\{2,3\})$ and $(\{1,2\},\{0,3\})$.

The resulting minimum solution consists of four dichotomies: $(\{1,2\},\{0,3\})$, $(\{2,3\},\{0,1\})$, $(\{0,2\},\{1,3\})$ and $(2,\{0,1,3\})$. A valid encoding, based on these dichotomies, is depicted in figure 8.12.

S	Code
0	1101
1	0111
2	0000
3	1011

FIGURE 8.12. A critical–race–free encoding of the machine in figure 8.8 that also obeys symbolic constraints

8.6 Binary instantiation

Now that we have a symbolic minimized description of our BMM, and a critical–race–free encoding, we can instantiate the BMM. We can use e.g. the encoding in figure 8.12 to instantiate the symbolic description of the machine in figure 8.11. Unfortunately, this will in general not result in a hazard–free implementation, despite the fact that each symbolic implicant is instantiated into a dhf–implicant.

Consider an input transition after which a current state transition takes place. After state assignment, it might turn out that some state variables will remain static at one during the input transition. After instantiation, static hazards could occur in these variables during the input transition, because we could not take into account these static state variables at the symbolic level. I.e., no symbolic required cube could be introduced to support these static variables, for it would introduce a conflict in the specification. The instantiated symbolic solution can be used only if a set of required cubes is added, after instantiation, covering these *static–1 transitions* [Fuhr97].

An unfortunate consequence of having to add these additional required cubes is that the symbolic solution itself is no longer an upper bound on the cardinality of the instantiated machine. The new upper bound can be estimated as the cardinality of the symbolic solution plus the number of additional cubes. However, in practice it turns out that the number of additional cubes can be quite large.

After recalculating the set of required and privileged cubes, based on the set of instantiated transitions, an additional binary minimization step can be performed in two ways:

- We can use the instantiated symbolic description as our initial binary description of the machine, or:
- We can use the encoding obtained, and use the set of required and privileged cubes as our initial description of the machine.

In table 8.1 we compare these two approaches for a set of well–known asynchronous benchmark circuits. We also compare the found solutions with an encoding that only obeys the Tracey constraints discussed in section 8.4.

TABLE 8.1 Symbolic minimization versus Tracey's method

		Symbolic		Encoding		Tracey	
circuit	I/S/O	#b	#c	#b	#c	#b	#c
dramc	7/3/6	2	22	2	22	2	22
ircv	4/4/3	4	13	4	13	2	11
isend	4/6/3	7	22	7	21	3	19
isend–bm	5/5/4	7	23	7	22	3	21
isend–csm	5/3/4	2	12	2	12	2	12
pe–send–ifc	5/5/3	6	26	6	26	3	21
pscsi	10/10/5	19	–	19	–	4	69
sbuf–read–ctl	3/3/3	3	7	3	7	2	7
sbuf–send–ctl	3/4/3	4	12	4	11	2	11
stetson–p2	8/13/12	10	–	10	–	4	35
stetson–p1	13/11/14	19	–	19	–	4	56
trcv–bm	4/4/4	4	17	4	15	2	13
trcv–csm	5/3/4	2	12	2	12	2	12
tsend	4/7/3	7	23	7	21	3	20
tsend–bm	4/5/3	8	19	8	19	3	18
tsend–csm	5/4/4	5	16	5	15	2	14

The first column in table 8.1 corresponds with the name of the asynchronous benchmark. In the second column the number of input variables (I), the

number of states (S) (after state minimization), and the number of output variables (O) is shown. In the third and fourth column, annotated by 'Symbolic' and 'Encoding' respectively, the results of symbolic hazard–free minimization are depicted. In these columns, #b represents the number of state variables, #c represents the number of cubes . In column "Symbolic", the symbolic solution itself is instantiated. After the set of required cubes necessary to cover the mentioned static–1 transitions have been added, the instantiated solution is minimized by a binary hazard–free minimizer. In column "Encoding" we only use the found critical–race–free state encoding derived by application of theorem 8.3. Based on this encoding the set of required and privileged cubes is derived, by converting all symbolic transitions to binary transitions. both sets are then provided to a binary hazard–free minimizer. The dashes, "–", in the table indicate that either symbolic or binary minimization of the circuit was cancelled after many hours of calculation. The binary minimizer that was used for the results in table 8.2 is the exact method of Dill/Nowick described in chapter 4.

The table shows that by only using the derived encoding, and by not instantiating the symbolic machine, we obtain better results in the number of cubes. The reason for this is that by instantiating the symbolic implicants, we fill in a lot of don't–care entries, which reduces the freedom of the binary minimizer, compared to the case where we only provide the minimizer with a set of required and privileged cubes.

The table also shows that symbolic minimization does not lead to a significant reduction in the number of cubes. In fact, for every depicted benchmark the circuit implementation, whose encoding only obeys the conventional Tracey constraints, represented in column "Tracey", is equal or smaller both in the number of cubes and the number of state variables.

8.7 Optimization strategies

As we can observe in table 8.1, the symbolic method introduces many state variables compared to the Tracey method. For example, for benchmark circuit "stetson–p1", the symbolic method does need 19 state variables, whereas 4 state variables are sufficient to cover all Tracey constraints. It seems that the symbolic method is not capable of generating better solutions in the number of cubes, because of the many state variables needed. Remember that the cardinality of the symbolic solution does not represent an upper bound on the binary solution. Due to the introduction of extra cubes, necessary to cover all the mentioned static–1 transitions, the binary solution derived from the symbolic solution might be degraded.

8.7.1 Using the Reached Current State Set

So the question is why does the symbolic method generate so many state variables? The answer to this question is shown in figure 8.13.

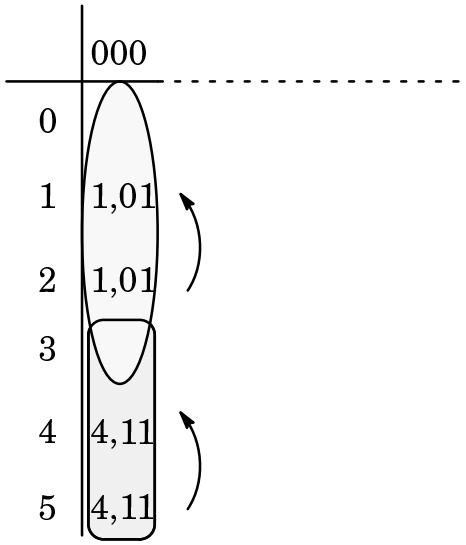


FIGURE 8.13. A 'weakness' in the symbolic method

In figure 8.13 part of a flow table is depicted. In this flow table two current state transitions are depicted, together with two implicants which cover these current state transitions. If the two symbolic implicants are part of the solution, then, according to theorem 8.3, the following dichotomies must be covered in order to obtain a valid critical–race–free encoding: $(\{1,2\},\{3,4,5\})$ and $(\{4,5\},\{0,1,2,3\})$. According to definition 8.3, both dichotomies are incompatible. Therefore, at least two state variables are needed to satisfy both constraints.

By examining the two implicants in figure 8.13, we can observe that both implicants can be reduced in size, so that each of these implicants exactly covers one current state transition. The reduction of the implicants does not increase the solution in the number of cubes. According to theorem 8.3, the two constraints now become $(\{1,2\},\{4,5\})$ and $(\{4,5\},\{1,2\})$. These dichotomies are compatible, and can be represented by one state variable.

The key observation of the reduction in the example in figure 8.13 is that it is possible that a symbolic implicant covers entries in the flow table which are unspecified for a given state. The subset of the CS–set of an implicant in which each state that has at least one specified entry is called the *Reached Current State Set* (RCS–set).

DEFINITION 8.10

The Reached Current State Set is a subset of the Current State Set (CS) of an implicant A, and is defined as:

$$RCS(A) = \{s \mid (s \in CS(A)) \wedge (\exists i : (i \Rightarrow I(A)) \wedge iFs)\}$$

Here, i is a minterm, representing a column in flow table F. The relation iFs is true when entry (i, s) in flowtable F has been specified.

The two implicants in figure 8.13 are defined as: <000,{0,1,2,3},1,01> and <000,{3,4,5},4,11>. The RCS–set of the first implicant is {1, 2}, the RCS–set of the second implicant is {4, 5}.

We can use the RCS–sets of all symbolic implicants to reduce the number of dichotomy constraints.

THEOREM 8.4

Consider a current state transition T, and an incompatible implicant A. In case the RCS–set and the CS–set of implicant A are not equal, the constraint stated by theorem 8.3 can be reduced to (CS(T), RCS(A)), by replacing the CS–set of implicant A by its RCS–set.

PROOF

We can safely replace all CS–sets of all implicants by their RCS–sets, because by doing so we just remove a set of unspecified entries from the coverage of each of these implicants. So by renaming the RCS–set into CS–set, and by applying theorem 8.3, theorem 8.4 is proven. □

Replacing a CS–set of an implicant by its RCS–set is very similar to the use of so–called dichotomy don't cares [Vill90]. For each symbolic implicant A, the face constraint becomes (RCS(A), s), where $s \in C \backslash CS(A)$. Reducing the number of states in the current state part of all implicants this way has no consequences for the cardinality of the solution in the number of implicants. The number of state variables necessary will however be reduced, because the probability that two or more dichotomies are compatible is increased. As we will see in the next section, this reduction can be quite impressive.

The calculation of the RCS–set of an implicant A is simply performed by checking which required cubes are covered by implicant A. For those cubes all states belonging to the CS–set are marked as belonging to the RCS–set of implicant A. The following algorithm shows this more clearly:

ALGORITHM 8.1. Calculating the RCS–set of each implicant

```
Foreach(A ∈ Implicants)
{
   RCS(A) := ∅;

   Foreach(J ∈ Req)
      if((I(J) ⇒ I(A)) ∧ (CS(J) ⊆ CS(A)))
         RCS(A) := RCS(A) ∪ CS(J);
}
```

In algorithm 8.1, Req corresponds with the set of required cubes.

8.7.2 Removing mutual dc–entries

For the following discussion, we need the following definition:

DEFINITION 8.11
An implicant A covers transition T if implicant A and transition T are compatible as defined in definition 8.9, if A implies a next state, and if $(I(T) \Rightarrow I(A)) \wedge (CS(T) \subseteq CS(A))$, which expresses the fact that T is covered by A.

Suppose a current state transition T is covered by some implicant A. Suppose that implicant B is incompatible with transition T, then, according to theorem 8.4, we should generate dichotomy $(CS(T), RCS(B))$. We can remove this constraint if implicant A and B intersect.

THEOREM 8.5
If current state transition T is covered by implicant A, and if transition T is incompatible with implicant B, and implicant A and B intersect each other, then the intersection belongs to the don't care set of the flow table.

PROOF
T is covered by A. So, A implies the same next state as current state transition T. B is incompatible with T, so for the intersecting part it could imply a different next state, or enable outputs not enabled by T. Suppose it implies a different next state. In that case two next states are defined for the intersection of A and B. If the intersection didn't belong to the don't care set this obviously couldn't occur. For the same reason B cannot enable outputs not enabled by T in case the intersection is specified. The conclusion must therefore be that the intersection belongs to the don't care set. □

Since, according to theorem 8.5, an intersection between implicants A and B belongs to the don't care set, we can make sure that A and B no longer intersect by sharpening implicant B. By doing this, we no longer have to generate dichotomy constraint (CS(T), RCS(B)). Unfortunately, checking this kind of intersection for all current state transitions, and all symbolic prime implicants is too expensive. We limit ourselves to an already found solution. That is: we use the set of current state transitions to check if we can detect implicant pairs A and B in the generated solution.

When such a pair is found, we remove the intersection between A and B by sharpening implicant B. Sharpening implicant B will in general lead to several new implicants. Therefore, after this sharpening operation we calculate a new minimum symbolic solution. It turns out that this procedure converges quickly.

If a solution is found, then we know that each implicant C intersecting implicant A is necessarily compatible with A. This means that the binary two–level minimizer has the possibility to instantiate each symbolic implicant by taking its CS–set. This might reduce the number of literals of each implicant, reducing the final implementation of the machine.

An example is shown in figure 8.14.

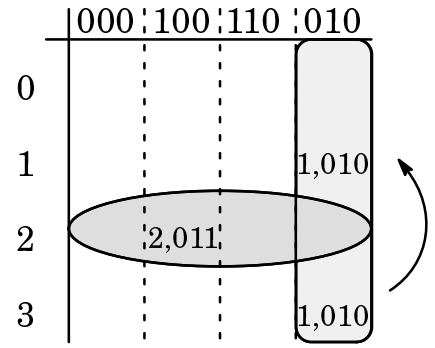


FIGURE 8.14. Mutual dc–entries between implicants I and J

In figure 8.14, part of a flow table is depicted. In the flow table we can observe a transition T: <010,{1,3},1,010>. Transition T is covered by implicant A <010,{0,1,2,3},1,010>. The RCS–set of implicant A is $\{1, 3\}$. Now, consider an incompatible prime implicant B defined as: <––0,{2},2,011>. According to theorem 8.4 we would have to generate dichotomy constraint $(\{1, 3\}, \{2\})$. However, by sharpening implicant B into implicants <–00,{2},2,011> and <1–0,{2},2,011>, we no longer need to generate this dichotomy constraint. Furthermore, during instantiation of implicant A, we can take the CS–set, that is set $\{0, 1, 2, 3\}$ instead of the RCS–set.

8.8 Results

In table 8.2 we compare the several possibilities to perform state encoding described in the previous sections.

TABLE 8.2 Comparison of several state encoding algorithms

		S		S+RCS				S+RCS+RMD				Tracey	
circuit	I/S/O	#b	#c	#b	#c	#b	#c	#b	#c	#b	#c	#b	#c
dramc	7/3/6	2	22	2	22	2	22	2	22	2	22	2	22
ircv	4/4/3	4	13	3	12	3	12	3	11	3	11	2	11
isend	4/6/3	7	22	4	22	4	21	4	20	4	20	3	19
isend–bm	5/5/4	7	23	5	23	5	21	4	19	4	17	3	21
isend–csm	5/3/4	2	12	2	12	2	12	2	12	2	12	2	12
pe–send–ifc	5/5/3	6	26	4	23	4	23	3	21	3	21	3	21
pscsi	10/10/5	19	–	7	84	7	74	6	83	6	68	4	69
sbuf–read–ctl	3/3/3	3	7	3	7	3	7	2	7	2	7	2	7
sbuf–send–ctl	3/4/3	4	12	4	12	4	11	3	11	3	11	2	11
stetson–p2	8/13/12	10	–	6	37	6	32	5	35	5	28	4	35
stetson–p1	13/11/14	19	–	9	–	9	–	7	61	7	57	4	56
trcv–bm	4/4/4	4	17	3	16	3	15	3	14	3	14	2	13
trcv–csm	5/3/4	2	12	2	12	2	12	2	12	2	12	2	12
tsend	4/7/3	7	23	4	21	4	21	4	21	4	20	3	20
tsend–bm	4/5/3	8	19	5	19	5	18	4	22	4	17	3	18
tsend–csm	5/4/4	5	16	4	16	4	15	4	16	4	15	2	14
scsi	9/8/5	21	–			10	90			9	90	4	91
TOTAL											442		452

Several results are depicted. The column indicated by "S" corresponds with column "Symbolic" in table 8.1. In column "S+RCS", we use the RCS–set of each prime implicant to derive the necessary encoding constraints. For each benchmark circuit two results are mentioned. The first result is obtained by instantiating the symbolic implicants. The second result shows the result of using the found state encoding, where the input to the hazard–free binary minimizer consists of an instantiated set of required cubes/privileged cubes. As can be observed in table 8.2, using the RCS–set has a big impact on the number of state variables required. For example, the symbolic method requires 19 state variables for benchmark circuit "pscsi". By using the RCS–set this number can be reduced to 7. The dashes, "–", in table 8.2 indicate that either symbolic or binary minimization of the circuit was cancelled after many hours of calculation. The binary minimizer that was

used for the results in table 8.2 is the exact method of Dill/Nowick described in chapter 4.

In the results mentioned in column "S+RCS+RMD", we use both the RCS–set of each selected implicant and the optimization strategy discussed in section 8.7.2. Again, two results are mentioned for each benchmark circuit. We observe that the number of state variables is reduced even more, compared to the number of state variables produced by the plain symbolic method, and the number of state variables produced by the improved method in column "S+RCS". Also the number of cubes is reduced. If we compare the results produced by this method with the corresponding numbers of the circuit implementations whose encoding only obeys the Tracey constraints, depicted in column "Tracey", we can see a slight improvement in the overall number of cubes.

8.9 A simulated annealing approach

In [Fuhr95] the simulated annealing mode of Nova [Vill90] is used to derive a critical–race–free encoding. In this approach, the number of state variables is bound by the minimum number of state variables necessary, which corresponds with the cardinality of the critical–race–free encoding resulting from theorems 8.1 and 8.2: the Tracey encoding. The set of constraints are partitioned into *compulsory* and *non–compulsory* constraints. The former constraints are necessary in order to avoid critical races, the latter are constraints that result from the symbolic minimization process. The simulated annealing algorithm is used to derive an encoding satisfying all of the compulsory constraints, and as many of the non–compulsory constraints as possible. Compared to a machine instantiated with a Tracey encoding a slight improvement can be observed. This method has as an additional advantage that the number of state variables is equivalent to the number of state variables needed by Tracey's method. We think that it is likely that these results can also be improved since Fuhrer does not incorporate the improvements described in section 8.7. Especially the use of the RCS–set instead of the CS–set will probably lead to more compulsory constraints being compatible, which might improve the number of non–compulsory constraints satisfied. The latter might lead to a reduction in the number of cubes in the final solution.

8.10 Conclusions

In this chapter we have examined the possibility of applying symbolic minimization during the synthesis of BMMs. Symbolic minimization for

BMMs, unfortunately, does not result in an upper bound estimation of the binary implementation as is the case for synchronous machines. In [Fuhr95] it is shown that by separating the next state logic from the output logic, it is possible to get an upper bound estimation on the number of cubes necessary to implement the output logic.

As we can observe from table 8.1 an encoding resulting from symbolic minimization in general does also not lead to better results compared to a plain Tracey encoding. Although our improvements described in section 8.7 have a big impact on the symbolic minimization approach, the results are only slightly better than those obtained with a Tracey encoding.

In future work it could be interesting to build in the improvements we suggested in section 8.7 in the simulated annealing approach by Fuhrer, discussed in section 8.9.

Chapter 9 Concluding remarks

9.1 General conclusions

In this thesis we have presented an almost complete synthesis trajectory for asynchronous Burst–Mode machines. We showed that the synthesis of Burst–Mode machines consists of the steps: state reduction, state assignment and logic synthesis/technology mapping. These steps are equivalent to those taken in the synthesis of synchronous finite state machines. This means that the algorithms, developed to achieve each of these steps for synchronous machines, can be taken as a starting point to develop algorithms that achieve the same steps for Burst–Mode machines.

In the past this approach resulted in the first complete synthesis trajectory developed by Nowick [Now93]. Compared to Nowick's synthesis system a lot has been improved over the last five years. A few of these improvements, which have been discussed and proposed in this thesis are:

- Better algorithms for state assignment, resulting in smaller implementations of the same Burst–Mode machine. In chapter 8 we have discussed some of these algorithms, and we have shown that some improvements are possible, resulting in even smaller machines.
- Logic synthesis has been dramatically improved. Logic synthesis in Nowick's system was targeted towards PLA–based implementations. Due to its exact nature, logic synthesis in Nowick's system was only applicable to relative small Burst–Mode machine specifications. This has changed radically. Not only have new fast exact algorithms been developed, discussed in chapter 4 and e.g. in [Theo98b], but also efficient heuristic algorithms have been developed that are capable of synthesizing the biggest Burst–Mode machines available today, also discussed in chapter 4 and [Theo98b]. In chapter 6 we also proposed a multi–level logic synthesis step that allows for smaller Burst–Mode machine implementations, compared to plain PLA–based implementations.
- In Nowick's system, verification of the generated Burst–Mode machine was not possible. In this thesis we have shown how a Burst–Mode machine can be verified at the level of transitions.

We mentioned that we presented an almost complete synthesis trajectory. It is incomplete in the sense that we did not consider the last step: generating

the actual layout of the BMM to be synthesized. This last step is, of course, a very important step. However, the viability of our synthesis trajectory does not crucially depend on this last step, like e.g. STG–based methods, because of our assumed unbounded wire delay model. Therefore, we can guarantee a correct implementation of the BMM specified.

9.2 Future work

Regarding the synthesis of individual Burst–Mode machines, it is, of course, possible to improve every step described above. If we, for example, look at the logic synthesis step we must conclude that multi–level synthesis for Burst–Mode machines is still somewhat immature. Work is also possible on specific layout, or dedicated logic for Burst–Mode machines, as was done in e.g. [Kud95], [Yun96].

There are also other areas, relevant to the synthesis of Burst–Mode machines, which have not yet been addressed very thoroughly. One of these areas is the sensitivity to deviations in signals due to e.g. noise or EMC. Asynchronous circuits are not driven by a global clock, and because of this they tend to generate less interference for their neighboring digital and analog circuits. However, the sensitivity of Burst–Mode machines themselves, and asynchronous circuits in general, to these interferences has never been examined.

These interferences tend to disturb the behavior of a circuit in two ways: they cause variations in delays and they cause glitches. Asynchronous circuits are probably more sensitive to glitches since a glitch corresponds to a number of false signal transitions to which the circuit might respond. In synchronous circuits, chances are very good for glitches to be filtered out by the clock–controlled latches/flipflops, especially when they are edge–triggered. Synchronous circuits on the other hand probably are more sensitive to variations in delays, while asynchronous circuits might be less sensitive. Delay variations will have the biggest impact on long wires, which are used in the interconnection of a set of asynchronous circuits. By using a delay–insensitive communication protocol between the circuits in this set, any variation in the delay can be tolerated.

We do not expect a future designer to partition his or her design in a set of communicating Burst–Mode machines. This should be the job of a system operating at a higher level of specification. In the literature, several examples are given. One is the highly successful DICY–system of Philips [Ber92a]. One weak point of the DICY–system is that it is based on a syntax–directed translation. This means that only local ”peep–hole” optimizations are

possible. Other approaches are possible and should be researched, like in [Kud95], where a design is originally specified as a petri–net with well–defined rules that can easily be verified by a designer, in contrast to STG–based synthesis methods.

If we look at the possible application areas of asynchronous systems, we see two major possibilities: low–power applications and data–flow applications. Due to their implicit low–power mode, asynchronous circuits are natural candidates for low–power applications. Data–flow applications are usually specified as data–flow graphs. Operations in data–flow graphs, like multiplication and addition, are mapped onto a selected set of execution units. These units are bound to vary in latency. So, some units will have some slack compared to the slowest unit, even if they are pipelined. In asynchronous systems, other computations can start within this slack. Even more interesting is the fact that in asynchronous designs, parts of an execution unit can start operating on partially available data. This is difficult to obtain in a synchronous design.

References

[Akel92] V. Akella, G. Gopalakrishnan: "SHILPA: A High–level Synthesis System for Self–Timed Circuits", International Conference on Computer Aided Design (ICCAD), pp. 587–591, 1992

[Ash92] P. Ashar, S. Devadas, A.R. Newton: "Sequential Logic Synthesis", Kluwer Academic Publishers, 1992

[Beer92] P.A. Beerel, T.H.–Y. Meng: "Automatic Gate–Level Synthesis of Speed–Independent Circuits", International Conference on Computer Aided Design (ICCAD), pp. 581–586, 1992

[Beer93] P.A. Beerel, T.H.–Y. Meng: "Logic Transformations and Observability Don't Cares in Speed–Independent Circuits", Proc. International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU), 1993

[Beis74] J. Beister: "A unified approach to combinational hazards.", IEEE transactions on Computers, Vol. C–23, No. 6, pp. 566–575, June 1974

[Ber92a] K. v. Berkel: "Handshake circuits: an intermediary between communicating processes and VLSI", Ph.D. thesis, Eindhoven University of Technology, 1992

[Ber92b] K. v. Berkel: "Beware the isochronic fork", Integration, the VLSI journal, No 13, pp. 103–128, June 1992

[Berk94] K.v. Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schalij: "A fully–asynchronous low–power error corrector for the DCC player", International Solid State Circuits Conference, pp. 88–89, February 1994

[Blaar92] M.R.C.M. Berkelaar: "Area–Power–Delay Trade–Off in Logic Synthesis", Ph.D. thesis, Eindhoven University of Technology, 1992

[Bred72] J.G. Bredeson, P.T. Hulina: "Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits", Information and Control, Vol. 20, pp. 114–224, 1972

[Brow90] F.M. Brown: "Boolean reasoning", Kluwer Academic Publishers, 1990

[Brzo95] J.A. Brzozowski, C–J.H. Seger: "Asynchronous Networks", Springer–Verlag, 1995

[Chak97] S. Chakraborty, D.L. Dill, K.–Y. Chang, K.Y. Yun: "Timing Analysis for Extended Burst–Mode Circuits", Advanced Research in Asynchronous Circuits and Systems (ASYNC), pp. 101–111, 1997

[Chan78] A.K. Chandra, G. Markowsky: "On the number of prime implicants", Discrete Mathematics, Vol. 24, pp. 7–11, 1978

[Chu87] T. Chu: "Synthesis of Self–timed VLSI Circuits from Graph–theoretic Specifications", International Conference on Computer Design (ICCD), pp. 220–223, 1987

[Coat93] B. Coates, A. Davis, K. Stevens: "The Post Office experience: designing a large asynchronous chip", Integration the VLSI journal, Vol. 15, No. 3, pp. 341–366, 1993

[Cort97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, A. Yakovlev: "Decomposition and Technology Mapping of Speed–Independent Circuits Using Boolean Relations", International Conference on Computer Aided Design (ICCAD), pp. 220–227, 1997

[Coud94] O. Coudert: "Two–level logic minimization: an overview", Integration the VLSI journal, Vol. 17, No. 2, pp. 97–140, 1994

[Davi97] A. Davis, S.M. Nowick: "An Introduction to Asynchronous Circuit Design", Technical Report UUCS–97–013, Columbia University

[Eich65] E.B. Eichelberger: "Hazard Detection in Combinational and Sequential Switching Circuits", IBM Journal of Research and Development, No. 9, pp. 90–99, March 1965

[Frac74] J. Frackowiak: "Methoden der analyse und synthese von hasardarmen schaltnetzen mit minimalen kosten I.", Elektronische Informationsverarbeitung und Kybernetik, Vol. 10, No 2/3, pp. 149–187, 1974

[Fried68] A.D. Friedman, P.R. Menon: "Synthesis of Asynchronous Sequential Circuits with Multiple–Input Changes", IEEE Transactions on Computers, Vol. C–17, No 6, pp. 559–566, June 1968

[Furb97] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day, N.C. Paver: "AMULET2e: An Asynchronous Embedded Controller", Advanced Research in Asynchronous Circuits and Systems (ASYNC), pp. 290–299, 1997

[Fuhr95] R.M. Fuhrer, B. Lin, S.M. Nowick: "Symbolic Hazard–Free Minimization and Encoding of Asynchronous Finite State Machines", International Conference on Computer Aided Design (ICCAD), pp. 604–611, 1995

[Fuhr97] R. Fuhrer, S.M. Nowick: "OPTIMIST: State Minimization for Optimal 2–Level Logic Implementation", International Conference on Computer Aided Design (ICCAD), pp. 308–315, 1997

[Gage98] H. v. Gageldonk: "An Asynchronous Low–Power 80C51 Microcontroller", Ph.D. thesis, Eindhoven University of Technology, 1998

[Gar79] M.R. Garey, D.S. Johnson: "Computers and Intractability. A Guide to the Theory of NP–Completeness", W. H. Freeman and Company, 1979

[Gras65] A. Graselli, F. Luccio: "A method for minimizing the number of internal states in incompletely specified sequential networks", IRE Transactions on Electronic Computers, EC–14, pp. 350–359, June 1965

[Hach96] G.D. Hachtel, F. Somenzi: "Logic synthesis and verification algorithms", Kluwer Academic Publishers, 1996

[Hauc95] S. Hauck: "Asynchronous Design Methodologies: An Overview", Proceedings of the IEEE, Vol. 83, No. 1, pp. 69–93, January 1995

[Hill93] F.J. Hill, G.R. Peterson: "Computer Aided Logical Design with Emphasis on VLSI", 4th edition, Wiley, 1993

[Kess97] J. Kessels, P. Marston: "Designing Asynchronous Standby Circuits for a Low–Power Pager", Advanced Research in Asynchronous Circuits and Systems (ASYNC), pp. 268–278, 1997

[Kond94] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, A. Yakovlev: "Basic Gate Implementation of Speed–Independent Circuits", Design Automation Conference (DAC), pp. 56–62, 1994

[Kud95] P. Kudva, G. Gopalakrishnan: "Techniques for Synthesizing Efficient Burst–mode Circuits", Proc. International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU), 1995

[Kung92] D.S. Kung: "Hazard–non–increasing gate–level optimization algorithms", International Conference on Computer Aided Design (ICCAD), pp. 631–634, 1992

[Lava92] L. Lavagno: "Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs", Ph.D. thesis, University of California at Berkeley, 1992

[Mago71] G. Mago: "Realization Methods for Asynchronous Sequential Circuits", IEEE Transactions on Computers, Vol. C–20, No. 3, pp. 290–297, March 1971

[Mart90] A.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, pp. 1–64. Addison–Wesley, 1990

[McCl86] E.J. McCluskey: "Logic design principles with emphasis on testable semi–custom circuits", Prentice Hall, 1986

[Meng91] T.H. Meng: "Synchronisation Design for Digital Systems", Kluwer Academic Publishers, 1991

[deM85] G. De Micheli, R.K. Brayton, A. Sangiovanni–Vincentelli: "Optimal State Assignment for Finite State Machines", IEEE Transactions on Computer–Aided Design, Vol. CAD–4, No. 3, pp. 269–285, July 1985

[deM86] G. De Micheli: "Symbolic Design of Combinational and Sequential Logic Circuits Implemented by Two–Level Logic Macros", IEEE Transactions on Computer–Aided Design, Vol. CAD–5, No. 4, pp. 597–610, October 1986

[deM94] G. De Micheli: "Synthesis and Optimization of Digital Circuits", McGraw–Hill, 1994

[Moon92] C.W. Moon: "Synthesis and Verification of Asynchronous Circuits from Graphical Specifications", Ph.D. thesis, University of California at Berkeley, 1992

[Now91] S.M. Nowick, D.L. Dill: "Synthesis of Asynchronous State Machines Using a Local Clock", International Conference on Computer Design (ICCD), pp. 192–197, 1991

[Now92] S.M. Nowick, D.L. Dill: "Exact Two–Level Minimization of Hazard–Free Logic with Multiple–Input Changes", International Conference on Computer Aided Design (ICCAD), pp. 626–630, 1992

[Now93] S.M. Nowick: "Automatic Synthesis of Burst–Mode Asynchronous Controllers", Ph.D. thesis, Stanford University, 1993

[Now94] S.M. Nowick, B. Coates: "UCLOCK: Automated Design of High–Performance Unclocked State Machines", International Conference on Computer Design (ICCD), pp. 434–441, 1994

[Now97] S.M. Nowick, N.K. Jha, F.–C. Cheng: "Synthesis of asynchronous circuits for stuck–at and robust path delay fault testability",

IEEE Transactions on Computer–Aided Design, Vol. 16, No. 12, pp. 1514–1521, December 1997

[Rud87] R.L. Rudell, A. Sangiovanni–Vincentelli: "Multiple–Valued Minimization for PLA Optimization", IEEE Transactions on Computer–Aided Design, Vol. CAD–6, No. 5, pp. 727–750, September 1987

[Rut96] J. Rutten: "Asynchronous Burst Mode Machines", Proc. ProRISC/IEEE–Benelux Workshop on Circuits, Systems & Signal Processing, Mierlo, The Netherlands, pp. 273–278, 1996

[Rut97a] J.W.J.M. Rutten, C.A.J. van Eijk: "Asynchronous Counters for Low Power Applications", Proc. ProRISC/IEEE–Benelux Workshop on Circuits, Systems & Signal Processing, Mierlo, The Netherlands, pp. 513–518, 1997

[Rut97b] J.W.J.M. Rutten, M.A.J. Kolsteren: "A Divide and Conquer Strategy for Hazard Free 2–Level Logic Synthesis", ACM/IEEE International Workshop on Logic Synthesis, Lake Tahoe, CA, May 1997.

[Rut97c] J.W.J.M. Rutten, M.R.C.M. Berkelaar: "Improved State Assignment for Burst Mode Finite State Machines", Advanced Research in Asynchronous Circuits and Systems (ASYNC), pp. 228–239, April 1997

[Rut98a] J.W.J.M. Rutten, M.R.C.M. Berkelaar, C.A.J. van Eijk, M.A.J. Kolsteren: "An Efficient Divide and Conquer Algorithm for Exact Hazard Free Logic Minimization", Proc. Conf. on Design, Automation and Test in Europe (DATE), pp. 749–754, 1998

[Rut98b] J.Rutten, M. Berkelaar: "Towards Multi–level Synthesis for Asynchronous Logic", Proceedings of the International Workshop on Logic Synthesis, Lake Tahoe, CA, pp. 77–82, June 1998

[Rut98c] J.W.J.M. Rutten, M.R.C.M. Berkelaar: "Efficient Exact and Heuristic Minimization of Hazard–Free Logic", International Conference on Computer Design (ICCD), pp. 152–159, 1998

[Rut98d] J.W.J.M. Rutten, M.R.C.M. Berkelaar: "Multi–level Logic Synthesis for Asynchronous Logic", Proc. ProRISC/IEEE–Benelux Workshop on Circuits, Systems & Signal Processing, Mierlo, The Netherlands, pp. 467–471, 1998

[Sald91] A. Saldanha, T. Villa, R. K. Brayton, A. L. Sangiovanni–Vincentelli: "A Framework for Satisfying Input and Output Encoding Constraints", Design Automation Conference (DAC), pp. 170–175, 1991

[Sawa95] M. Sawasaki, C. Ykman–Couvreur, B. Lin: "Externally Hazard–Free Implementations of Asynchronous Circuits", Design Automation Conference (DAC), pp. 718–724, 1995

[Sema97] Sematech Roadmap: "The national roadmap for semiconductors", Semiconductor Industry Association, 1997 edition

[Sent92] E. M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, A. Sangiovanni–Vincentelli: "SIS: A System for Sequential Circuit synthesis, University of California", Berkeley, Memorandum No. UCB/ERL M92/41

[Sylv98] D. Sylvester, K. Keutzer: "Getting to the Bottom of Deep Submicron", International Conference on Computer Aided Design (ICCAD), pp. 203–211, 1998

[Thee96] F. Theeuwen, E. Seelen: "Power Reduction through Clock Gating by Symbolic Manipulation, VLSI": Integrated systems on silicon, Gramado, RS Brazil, pp. 389–400, August 1997

[Theo96] M. Theobald, S.M. Nowick, T. Wu: "Espresso–HF: A Heuristic Hazard–Free Minimizer for Two–Level Logic", Design Automation Conference (DAC), pp. 71–76, 1996

[Theo98a] M. Theobald, S.M. Nowick: "An implicit method for hazard–free two–level minimization", Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), pp. 58–69, 1998

[Theo98b] M. Theobald, S.M. Nowick: "Fast Heuristic and Exact Algorithms for Two–level Hazard–Free Logic Minimization", To appear in IEEE Transactions on Computer–Aided Design

[Trac66] J.H. Tracey: "Internal State Assignment for Asynchronous Sequential Machines", Transactions on Electronic Computers, Vol. EC–15, No. 4, pp. 551–560, August 1966

[Ung69] S.H. Unger: "Asynchronous Sequential Switching Circuits", Wiley, 1969

[Vill90] T. Villa, A. Sangiovanni–Vincentelli: "NOVA: State Assignment of Finite State Machines for Optimal Two–Level Logic Implementation", IEEE Transactions on Computer–Aided Design, Vol. 9, No. 9, pp. 905–924, September 1990

[Will91] T.E. Williams, M.A. Horowitz: "A Zero–Overhead Self–Timed 160–ns 54–b CMOS Divider", IEEE Journal of Solid–State Circuits, Vol. 26, No. 11, pp. 1651–1661, November 1991

[Wu91] S–F. Wu, D. Fisher: ”Automating the Design of Asynchronous Sequential Logic Circuits”, IEEE Journal of Solid–State Circuits, Vol. 26, No. 3, pp. 364–370, March 1991

[Yako96] A.V. Yakovlev, A.M. Koelmans, A. Semenov, D.J. Kinniment: ”Modelling, analysis and synthesis of asynchronous control circuits using Petri nets”, Integration the VLSI journal, Vol. 21, No. 3, pp. 143–170, 1996

[Yun94] K. Yun: ”Synthesis of Asynchronous Controllers for Heterogeneous Systems”, Ph.D. thesis, Stanford University, 1994

[Yun96] K.Y. Yun: ”Automatic Synthesis of Extended Burst–Mode Circuits Using Generalized C–elements”, European Design Automation Conference (EURO–DAC), pp. 290–295, 1996

Index

Numbers

A

B

C

D

E

P

Q

R

S

T

U

V

W

Z