Technical Report

UCAM-CL-TR-468
ISSN 1476-2986

Number 468

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Synthesis of asynchronous circuits

Stephen Paul Wilcox

July 1999

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 1998 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Queens' College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

The majority of integrated circuits today are synchronous: every part of the chip times its operation with reference to a single global clock. As circuits become larger and faster, it becomes progressively more difficult to coordinate all actions of the chip to the clock. Asynchronous circuits do not suffer from this problem, because they do not require global synchronization; they also offer other benefits, such as modularity, lower power and automatic adaptation to physical conditions.

The main disadvantage of asynchronous circuits is that techniques for their design are less well understood than for synchronous circuits, and there are few tools to help with the design process. This dissertation proposes an approach to the design of asynchronous modules, and a new synthesis tool which combines a number of novel ideas with existing methods for finite state machine synthesis. Connections between modules are assumed to have unbounded finite delays on all wires, but fundamental mode is used inside modules, rather than the pessimistic speed-independent or quasi-delay-insensitive models. Accurate technology-specific verification is performed to check that circuits work correctly.

Circuits are described using a language based upon the Signal Transition Graph, which is a well-known method for specifying asynchronous circuits. Concurrency reduction techniques are used to produce a large number of circuits that conform to a given specification. Circuits are verified using a bi-bounded simulation algorithm, and then performance estimations are obtained by a gate-level simulator utilising a new estimation of waveform slopes. Circuits can be ranked in terms of high speed, low power dissipation or small size, and then the best circuit for a particular task chosen.

Results are presented that show significant improvements over most circuits produced by other synthesis tools. Some circuits are twice as fast and dissipate half the power of equivalent speed-independent circuits. Examples of the specification language are provided which show that it is easier to use than current specification approaches. The price that must be paid for the improved performance is decreased reliability, technology dependence of the circuits produced, and increased runtime compared to other tools.

Preface

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

This dissertation is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other University. No part of this dissertation has already been or is concurrently being submitted for any such degree, diploma or other qualification.

I believe that this dissertation is 59 861 words in length, including bibliography and footnotes but excluding diagrams, and hence complies with the limit of 60,000 words put forward by the Board.

Acknowledgements

I would like to thank Simon Moore and Peter Robinson for their advice and comments, the EPSRC for their funding, and George and Paul for spotting mistakes in various parts of this thesis. I would especially like to thank Judie for putting up with me, and my parents for their support and for getting me to the stage where I could attempt this.

This dissertation was typeset in LaTeX 2ϵ, and all diagrams produced using xfig 3.2.0, both from the Red Hat Linux 5.0 distribution. The body text is 10pt Bitstream Benguiat with headings set in Benguiat Gothic. Programs L2b, b2ps, prune and synth were written in C++ and compiled using GNU g++ 2.8.1. When execution times are given in the text, these refer to the time taken to run the program on a 210 MHz AMD K6 with 64MB memory running Linux kernel 2.0.32.

Contents

List of Figures

Chapter 1: Introduction

Chapter 2: Previous Work

Chapter 3: Overview and Motivations

Chapter 4: Specification

Chapter 5: Concurrency Reduction

Chapter 6: Synthesis

Chapter 7: Timing and Verification

Chapter 8: Results

List of Tables

Chapter 2: Previous Work

Chapter 4: Specification

Chapter 5: Concurrency Reduction

Chapter 6: Synthesis

Chapter 7: Timing and Verification

Chapter 8: Results

Introduction 1

1.1 Why Asynchrony?

The transistor has gone a long way since its discovery by Bardeen and Brattain in 1947 [16]. In the early 50s, integrated circuits with as many as ten transistors were available; by the 80s, hundreds or even thousands of transistors could be integrated on a single die. In 1998, barely fifty years on from the first transistor, microprocessors costing under $100 contain almost ten million transistors, and the scale of integration seems likely to rise even further.

Initially, circuits were largely designed in an ad-hoc manner without requiring global synchronization. Consequently, many early computers were asynchronous, such as ORDVAC at the University of Illinois and IAS at Princeton. It was soon found that a global timing signal would allow smaller and faster circuits to be produced, such as the later Illinois machines, ILLIAC II, III and IV. The introduction of a global clock allowed systems to be decomposed into subsystems, each of which was a finite state machine with its outputs synchronized to one edge of the clock. Design correctness was simply a matter of determining the delays in the combinational logic within each subsystem, and checking that latch setup and hold times were not violated. Checking that an asynchronous circuit was correct required removing hazards, critical races and, at a higher level, checking for deadlock possibilities.

Synchronous circuits soon began to dominate digital design. The simplifying assumption that time is discrete, partitioned by clock pulses, permitted progressively larger and more complex designs to be created, with a good degree of confidence that the design will operate correctly. As circuits grew, synchronous design techniques and CAD tools became more widespread, and asynchronous design was mostly forgotten.

As lithography became more advanced, feature sizes became smaller and clock speeds rose. Constant field scaling [189] implies that wire delays for a particular circuit will scale down proportionally to feature size as gate delays do, but the maximum economic die size has remained fairly constant at about 200–400 mm^2. Wires are therefore increasing in length relative to other features at the same rate that transistors are becoming faster. In an effort to keep wire resistance low, wires have become taller than they are wide, but this has adverse effects on inter-wire capacitance and, recently, inductance [165].

Significant delays in wires cause *clock skew*, where the clock edge is not seen simultaneously at all points on the die. Optical injection of the clock is possible, but this will not solve clock skew problems for clock speeds much above 1 GHz. For example, the permissible clock skew on the 500MHz Alpha 21164 was 90ps, a time in which light can scarcely cross the chip, so optical clocking would only just cope with current clock speeds. With the current roadmaps predicting 0.1 μm feature sizes giving clock speeds in excess of 4 GHz by 2010 [188], it can be seen that the assumption of a single global clock will fail within the next ten or fifteen years. Even today, clock distribution is difficult. For the last few years, Digital's Alpha design team have had to find increasingly esoteric ways to reduce clock skew. The 21064 had a massive 35cm wide clock driver in the centre of the die [66], the 21164 had a pair of drivers totalling 58cm to reduce the distance from the clock driver to any point on the circuit [14], whereas the 21264 has a distributed network of conditional clocks with known skew. Even if the clock can be distributed successfully, data signals still travel at sublight speeds on-chip, a fact that required two register files in the 21264 to reduce the distance data had to travel in a single clock cycle.

As synchronous circuits begin to hit these fundamental technology barriers, asynchronous circuits look to be poised for a comeback. Asynchronous circuits are any that do not have a global synchronisation signal; they can range from locally-clocked modules connected in a clock-free way to fully delay-insensitive circuits. Asynchronous circuits have a number of advantages:

- They automatically adapt their speed to suit their physical conditions:
  - Temperature: Martin's asynchronous microprocessor functioned correctly, and much faster, when placed in liquid nitrogen [120]
  - Age of components: hot-carrier effects [54] cause degradation in short-channel transistors over time, causing a synchronous circuit to fail to meet timing margins
  - RF interference: individual gate delays can vary –50% to +100% due to low-level EMI [25]
- Lower power:
  - Only parts of the circuit that are being used take power, however newer synchronous processors use conditional clocking to achieve the same goal [66].
  - Dynamic supply voltage variation can cut power, e.g. by a factor of 20 for an asynchronous DCC player [86], although dual supplies have also recently been used for low power in synchronous circuits [181].
- Infrequently used subcircuits can be left unoptimised, at very little performance penalty.
- Better technology migration potential. Because asynchronous circuits do not use global timing assumptions, it is possible to implement a circuit using a different gate library or possibly a completely different logic family, as Tierno et al. [175] showed when they ported the Caltech microprocessor to Gallium Arsenide. Basic delay-insensitive building blocks [145] and asychronous

pipelining schemes [82] have even been demonstrated for rapid single-flux quantum (RSFQ) superconducting devices, which are still in their infancy.

- The outside world is asynchronous; in particular, metastability (see Chaney and Molnar [24]) is not a problem when the circuit can wait for its components to stabilise.

It is also often said that asynchronous circuits give average case performance, rather than the worst case performance which must be accepted for synchronous circuits. This statement requires some qualification. Bundled-data approaches require overestimating the worst-case datapath delay by typically 100% to allow for process variations [57], whereas a synchronous circuit may be clocked only 10–20% slower than the speed at which it fails. Handshaking overheads also increase the time to do any operation on data, although Martin [115] believes that this overhead is roughly the same as the clock skew penalty in todays CMOS circuits.

Papers which state that average delays can be substantially less than worst-case delays usually use a ripple-carry adder as an example, but the worst-case for a ripple carry happens surprisingly often in microprocessors [87]. It is also the case that carry select and carry skip adders are reasonably simple, so ripple carry adders will not be used in real designs. Achieving average-case performance requires completion detection, which takes a time overhead that is not present in synchronous circuits, although this can be taken off the critical path. Pipelines that are built out of elements that have large delay variances tend to perform worse than pipelines with a more uniform delay per stage, unless additional decoupling is used [84]. To summarise, the only fast asynchronous circuits are likely to be ones using pipelined completion detection with carefully prepared pipeline structures, such as proposed by Martin [121].

On the other hand, there are some major disadvantages to asynchronous circuits:

- Many of the techniques that make it easier to design synchronous circuits cannot be used for self-timed design. Inputs to asynchronous circuits are active all the time, whereas in synchronous circuits they are only sampled at well-defined intervals. This leads to problems with hazards [180] when reducing Boolean expressions using algorithms designed for synchronous circuits.

- It is not possible to put latches round all the parts of an asynchronous circuit and run the circuit slower for testing purposes. In particular, scan paths and design-for-test will have to be modified for use in asynchronous circuits, but much effort is being expended here. It has often been said that stuck-at faults in certain classes of asynchronous circuits cause them to stop rather than give an incorrect answer, so testing is in some sense built-in, but this has been disputed [20].

- Some global timing issues return and are difficult to solve, such as deadlock or livelock in systems composed of many concurrent parts.

- There are few proven CAD tools to help with design.

Although asynchronous circuits may not show speed improvements over equivalent synchronous circuits, it may be possible to develop asynchronous architectures that simply have no synchronous counterparts. An example is Sproull and Sutherland's Counterflow Pipeline Processor [170]; this can be built in a clocked way, but can take advantage of an asynchronous framework in a way that a clocked version could not. Another example is the Rotary Pipeline processor of Moore, Robinson and Wilcox [128], which is a generalisation of Williams' self-timed ring structures. Data flows round a ring of ALUs without having to wait for control or clock overheads until it reaches the register file. Certain specific areas, such as DSPs, have been showing the advantages of asynchronous circuits for some time [79].

1.2 Aims

The work in this dissertation was inspired by Furber and Day's paper on latch controllers [58]. They specified a circuit to operate the latches in an asynchronous pipeline by giving orderings between rising and falling transitions of the inputs and outputs of the latch controller circuit. These orderings are better known as *Signal Transition Graph (STG) fragments*. Implementations were produced by hand, and relied upon the skill of Furber and Day to produce fast circuits.

Orderings between transitions are an intuitive way to specify the behaviour of a circuit, but not all circuits can be described in this way; consider a circuit where the choice between two transitions depends on the state of a third level-sensitive input. To be useful as a specification, transition orderings must be augmented with other constructions.

One of the interesting features of Furber and Day's paper [58] is that three implementations were produced which allowed varying degrees of concurrency between adjacent pipeline stages. Chapter 3 introduces an intermediate representation of the interface behaviour of a circuit, which makes it easy to change the amount of concurrency in a similar way. A fast concurrency-reducing transformation can be defined on this intermediate form, which allows a large number of possible implementations to be investigated.

The aim of this dissertation is to describe the development of a synthesis tool for asynchronous circuits, which starts with STG fragments, performs concurrency reduction on intermediate forms, and synthesizes these forms into verified modules. In detail, the aims are:

1. To create a front-end description, based upon STG fragments, that is powerful enough for almost all real-world circuits and is simple to use.

2. To compile this specification into the intermediate form mentioned above.

3. To show that exhaustive enumeration of concurrency-reduced intermediate forms is possible within a reasonable time.

4. To show that the concurrency-reduced intermediate forms can be synthesized into circuit modules and verified as correct given bounds on the environment reponse times.

5. To show that circuits produced tend to be superior to current asynchronous tools, in terms of the scoring function given by the designer.

1.3 Structure of this dissertation

A pictorial overview of the synthesis tool described in this dissertation is given in Figure 1.1.

Chapter 2 relates previous work in asynchronous circuits, concentrating on specification styles and fundamental mode synthesis techniques. Literature on timing and verification will be left until Chapter 7.

Chapter 3 gives the observations that prompted the work described in this dissertation. It can be viewed as a roadmap for the dissertation.

Chapter 4 describes the design of a specification language, based upon STG fragments, and the way in which this language is translated first to a Petri net, and then into an intermediate form called a blue diagram. This translation is performed by the program L2b. Some example specifications are given, from a number of sources including the standard set of SIS STG benchmarks [101].

The concurrency reduction operation is described in Chapter 5, and comparisons made with other approaches to the problem. The concurrency reduction algorithm was implemented in the program prune.

Chapter 6 explains the synthesis algorithms that were used in the synthesis program synth. Most of the methods are based upon existing work, but with some modifications to improve the results.

Chapter 7 gives the gate-level timing algorithms that were used, and describes a verification algorithm that uses the gate-level timing analysis.

Chapter 8 lists the results of the whole synthesis procedure for the example circuits that were considered in Chapter 4. Results are also given for the different state assignment algorithms and implementations considered in Chapter 6.

Chapter 9 gives an summary of the work presented in this dissertation, along with conclusions that can be drawn and possible areas for future work.

Typographic conventions

Anything that would be expected to occur in a text file will be set in a `typewriter` font, such as signal names in a specification, and transitions of those signals, and keywords such as `module` and `arbitrate`. Letters that are being used to stand for one out of a number of possible transitions or signals will be set in *italics*, as will the names of well-known asynchronous synthesis examples such as *alloc-outbound*. Program names such as L2b and prune will be set in sans serif. L2b actually has a lower case "L", but this tends to read as "twelve-b", so it has been changed so an upper case letter in this dissertation.

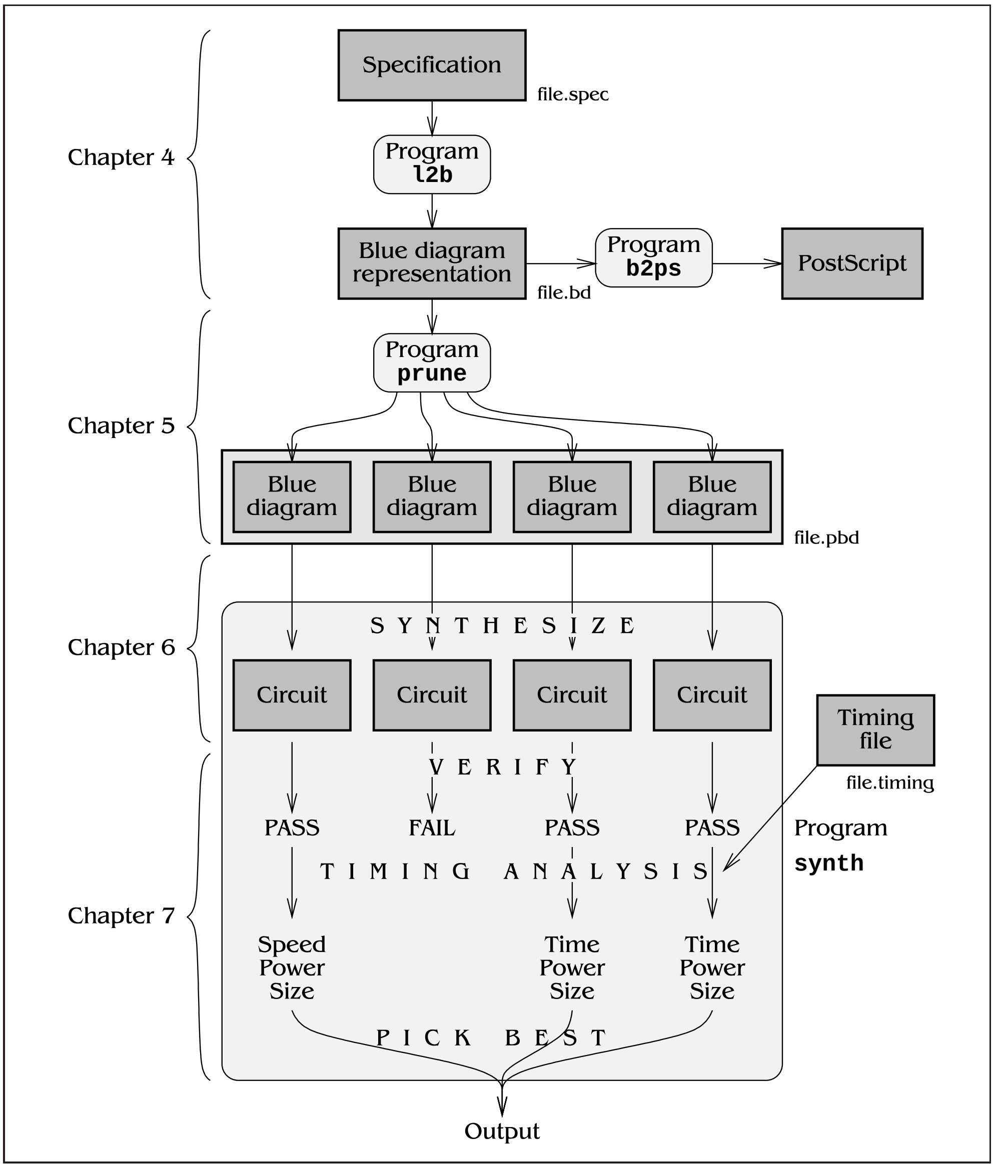


Figure 1.1: An overview of the synthesis tool presented in this dissertation

Previous Work 2

He who cannot draw on
three thousand years [of knowledge]
is living from hand to mouth
– Goethe

2.1 Delay assumptions

An important early work in asynchronous circuit synthesis is the book by Unger [177], which collected a number of results and methods into a definitive reference work for the early seventies. At that time, there were two main types of circuit, which were distinguished by what they assumed about the delays that were present in circuits:

- **Fundamental mode** or **Huffman** circuits, due to D. A. Huffman [77]. The delay assumption is that upper and lower bounds are known for all gate and wire delays. When a combination of inputs has been given to a Huffman circuit, these known bounds can be used to determine when the circuit will become stable, and the environment must wait for the circuit to stabilize before providing another input. Formally, for any circuit there exist real numbers $\delta_2 > \delta_1 > 0$ such that two input transitions less than δ_1 apart are treated as a single change, and two transitions greater than δ_2 apart are treated as two sequential inputs. If the delay between two inputs is between δ_1 and δ_2, then the behaviour of the circuit will be undefined. Hazard removal for Huffman circuits is difficult, and is often avoided by either imposing the restriction that only one input changes at a time, which limits concurrency and impacts performance, or adding explicit inertial delays on outputs, which also reduces performance.

- **Speed Independent** or **Muller** circuits, after D. E. Muller [132]. The delay assumption used is that gate delays are unbounded but finite whereas wires have no delay. The only way to find out whether a Muller circuit has finished a computation is to have it return a completion signal, which indicates that the circuit is ready to receive another input.

Muller's 1955 technical report [132] defined a notion of asynchronous circuit correctness that would be useful during design. One of his desirable properties for a circuit (Condition 3 from [132]) states that if a circuit, complete with its environment, is broken at a set of one or more nodes, then the final equilibrium state of the circuit (if one exists) should not depend on the relative speeds of the active elements. In subsequent reports [133, 134] he defines *speed-independence* and *semi-modularity* and proves some results connecting these concepts. Let a gate be *excited* if a change to it inputs has just occurred which will cause a change in its output, but the output change has not happened yet. The gate *fires* when the output change happens.

Speed Independence A circuit is speed independent with respect to a particular initial state if all behaviours of the circuit starting in that initial state end up in one equivalence class of states. In the circuits considered in this dissertation, there will not be any oscillating internal states of a control circuit; in this case, the condition for speed-independence can be restated as "The final state of the circuit does not depend on the relative delays of gates".

Semi-Modularity A circuit is semi-modular if, from any state $\underline{b}$ reachable from the initial state, and for any successor state $\underline{c}$ of $\underline{b}$, then any excited gates that do not fire in the transition from $\underline{b}$ to $\underline{c}$ are still excited in $\underline{c}$. Equivalently, "An excited gate can only become stable through firing", which is the usual definition of semi-modularity.

Muller proved that a semi-modular circuit is also speed-independent, although the reverse is not true, and that speed-independence implies that condition 3, which was mentioned above, is satisfied. A good account of the work of Muller can be found in Miller's book [124].

Both the Fundamental mode and Speed-Independent models have their problems. Fundamental mode relies upon knowing the delays of circuit elements, but Chappel and Zaky [25] states that gate delays may be affected by as much as factor of two by low-level electromagnetic interference. Speed independent circuits do not take account of wire delays, which dominate gate delays in submicron CMOS [36]. Some speed-independent approaches [3, 94] assume that zero-delay input inverters are available on all gates, which is a violation of true speed-independence but is fairly safe in practice.

Other delay models include:

- **Delay-Insensitive** or **DI** design assumes that the delays in both wires and gates are finite but unbounded, and is the most robust assumption that can be made. The term "Delay-Insensitive" was coined by Molnar and Clark in the Macromodules project [32]; they also defined the *Foam Rubber Wrapper Property* as a test for delay-insensitivity, which states that if arbitrarily delaying the input and output transitions of a circuit cannot cause a hazard or a change in its behaviour, then its interface is delay-insensitive.

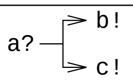
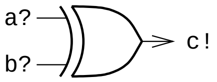
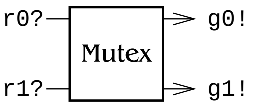
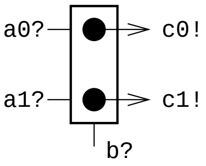
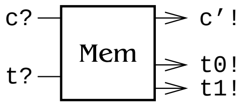
<table>
<tr><th>Component</th><th>Symbol</th><th>Trace expression</th></tr>
<tr><td>Fork</td><td> </td><td>pref [a?;(b!||c!)]*</td></tr>
<tr><td>Merge</td><td> </td><td>pref [(a?|b?);c!]*</td></tr>
<tr><td>Mutex</td><td> </td><td>pref{(r0?g0!r0?g0!)*
||(r1?g1!r1?g1!)*
||[(g0?g0?)|(g1?g1)]*}</td></tr>
<tr><td>2×1-Join</td><td> </td><td>pref{[(a0?||b?)c0!]
[(a1?||b?)c1!]}*</td></tr>
<tr><td>Mem</td><td> </td><td>pref [(t?t0!)*c?c'!
(t?t1!)*c?c'!]*</td></tr>
</table>

Figure 2.1: DI circuit modules from Patra and Fussel [144]

Unfortunately, very few circuits fall into this class, so some weakening assumptions have to be made—Martin [119] showed that a DI circuit composed of only single-output gates can contain only NOT gates and C-elements, which do not allow enough flexibility to build most circuits. Because DI design is so restrictive, synthesis algorithms usually just connect a set of predefined low-level modules, where the modules themselves are designed using a different delay model. These modules are usually built out of simpler gates, such as AND, OR and NOT, but it has recently been found that certain modules can be efficiently built directly in some superconducting technologies, giving modules that are substantially smaller than AND or OR gates [145]. Effort has gone into finding the best set of DI components; Patra and Fussel [144] say that the five modules shown in Figure 2.1 are minimal and optimal in a sense defined by Keller [85].

- **Quasi Delay Insensitivity** or **QDI** modifies DI by introducing *isochronic forks* [117], structures that allow delay matching over a limited area. An isochronic fork is shown in Figure 2.2. When a signal starts at p and travels down the forked wire, it will be seen to arrive at different times at q and r. If the difference between the arrival times of the signal at q and r is less than the propagation delay of either of the gates driven by the fork, labelled f and g, then the fork is deemed to be isochronic. This assumption can be upheld by making the two prongs of the fork almost equal in length, and ensuring that the thresholds of f and g are not widely different. Asymmetric forks were

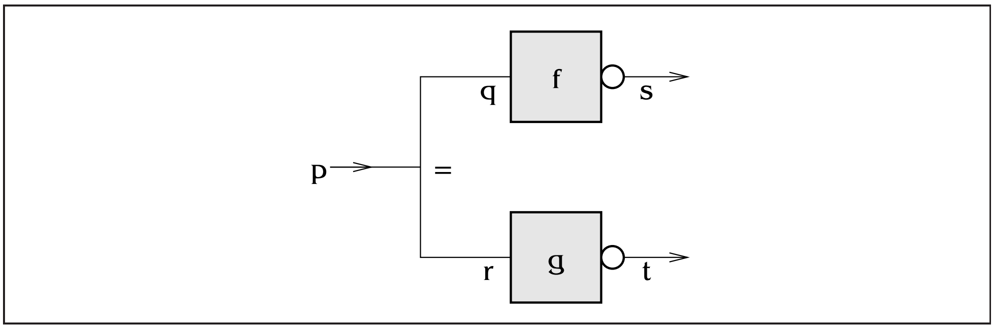


Figure 2.2: An isochronic fork

also proposed, where the signal is guaranteed to arrive at one end before the other. Using isochronic forks enables a wider variety of circuits to be designed, but they must be treated with care, as pointed out by van Berkel [9]. A pair of CMOS gates can often have significantly different input thresholds, even if the gates are identical, which means that a slow ramp voltage caused by a long wire could trigger two gates at very different times. Current automatic place-and-route tools may not honour the isochronic forks in a design, which is problematic. Some research teams, such as the TITAC team [139], have found that the QDI assumption is overly pessimistic and leads to low performance, but Martin's work disputes this.

- **Quasi-QDI** or **Q^2DI** is a further relaxation of QDI. QDI circuits can be quite large when built out of standard cells. Van Berkel proposed *extended isochronic forks* [12] as a way to design more compact circuits with better performance, at the expense of using a more risky delay assumption. The assumption used, with reference to Figure 2.2, is that the difference in delay between p and s and between p and t is less than the propagation delays of gates driven by s and t. Extended isochronic forks may require post-layout verification to make sure the assumption holds.

- **Field Forks:** The problems with DI prompted Kishinevsky et al. to consider *Field Forks* [89]. A CMOS gate has two contacts, one on each side of the active area, so a signal that should be forked to a number of gates can instead be chained through the necessary transistors. The transistors should change state in the order of the chaining, and given this knowledge, the circuit can be designed to work correctly. Forking a signal to the P and N transistors of a gate is allowed, for example in a NOT gate. Although elegant, field forks have been largely ignored.

Armstrong et al. [1] attempted to bridge the gap between the two delay models in 1969 by using unbounded gate delay and bounded wire delay, but this had similar properties to speed-independent circuits. Other recent delay assumptions are the *unbounded complex-gate* assumption used by Chu in [28, 30], *bounded simple-gate* assumption as used by Lavagno et al. in SIS [105], and the *bounded complex-*

gate assumption used by Moon et al. [127]. There is a rapidly growing number of delay models, each having their own strengths and weaknesses, with no one model being the best for all occasions.

2.2 Signalling and data conventions

In synchronous circuits, data transfer is simply a matter of making sure that the sender observes setup and hold times on the data lines, and that the receiver samples the data lines when the clock edge arrives. Data transfers in asynchronous circuits do not have a clock edge to synchronize to, so other ways of coordinating the sending and receipt of data must be found. Events passed from one asynchronous module to another can be considered as data transfers of a null value, so all inter-module communication can be treated as data transfers.

2.2.1 Two-phase versus four-phase protocols

When passing events and data between modules, it is usually not known exactly what the wire delays are between the modules. In such cases, it is important to use methods which do not assume precise delays, such as request/acknowledge handshaking. When just events are being sent, there are two ways to organize the request and acknowledge wires: *two phase* and *four phase* signalling. Two phase signalling, also called *transition* signalling, treats both rising and falling edges of signals identically. Four phase, or *level* signalling, assigns no meaning to falling edges, using only rising edges to convey information. Figure 2.3 shows this graphically.

The choice between two phase and four phase design is not easy to make, as pointed out by Sutherland et al. [171]. Two phase signalling maps well to formalisms such as trace algebra, because there are no unnecessary transitions; this also can theoretically reduce power and increase speed. Unfortunately, two phase circuit elements, such as XOR gates and C-elements, tend to be larger and slower than level-sensitive gates such as AND and OR. The link with trace theory makes two-phase design clean and elegant, but this is not preserved in the resulting circuits, which often have duplicated circuit blocks. CMOS is fundamentally a level-sensitive technology, so four-phase signalling maps onto the hardware better; it also is a more familiar model to most circuit designers. The disadvantage of four-phase control is that the falling edges introduce useless concurrency and extra power and delay, which in turn complicates formal analysis.

2.2.2 Bundled data versus delay-insensitive schemes

When several bits of data are to be sent rather than just a single event, the schemes above need to be generalized. One way is the *bundled data* approach, where it is assumed that the time taken for the data on a bus to travel from the sender to the receiver is almost the same as the time that a request transition takes to do the same journey; this is a good approximation if the data lines and request wire are

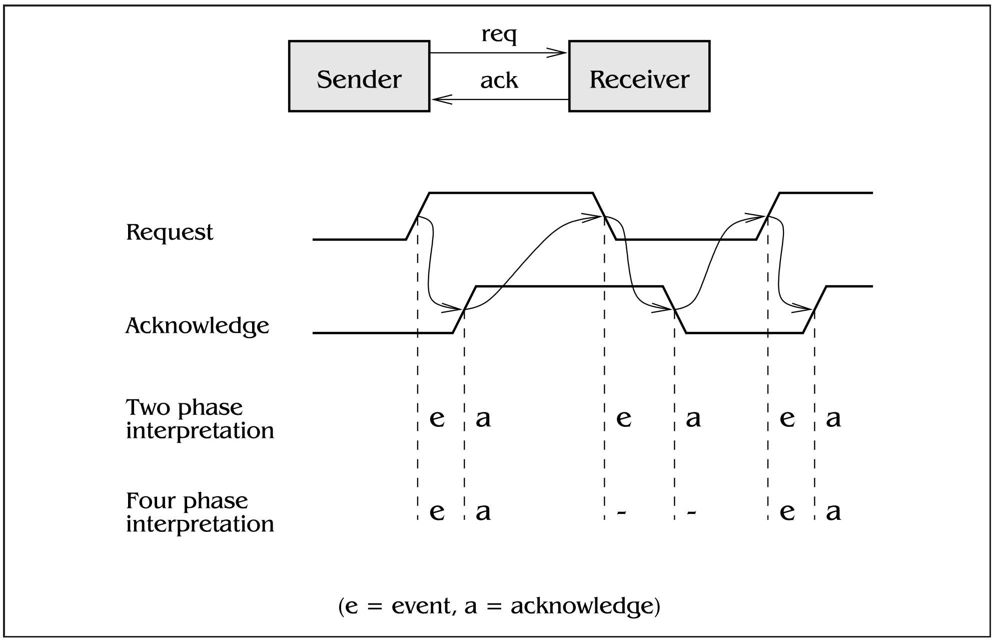


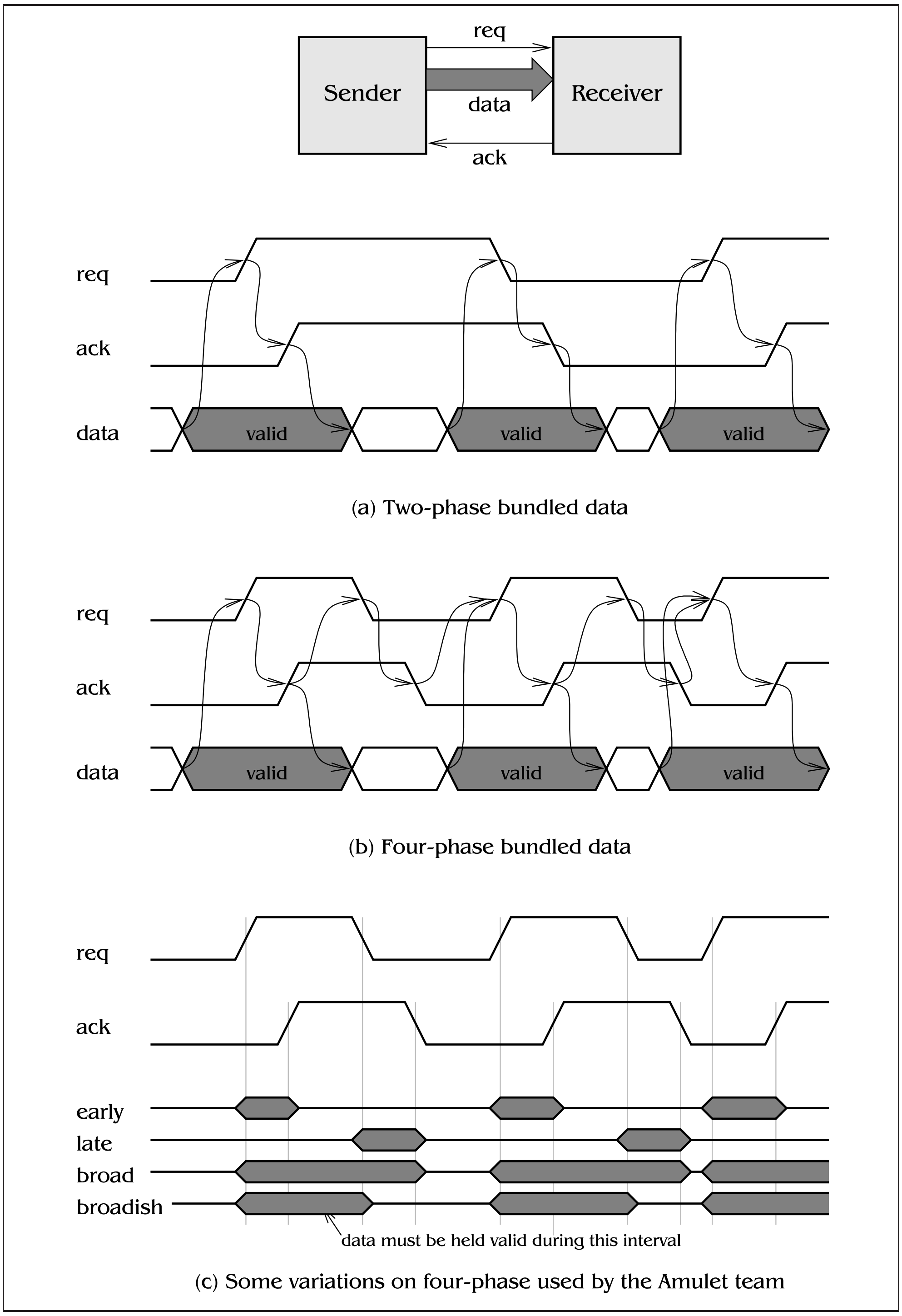
Figure 2.3: Two phase and four phase events

all routed very close to one another. Bundled data can be used with either two-phase or four-phase control signalling, as in Figure 2.4. The data wires are driven shortly before a request event is sent. When the receiver sees the request event, it can be assumed that the data is stable at the receiver. The useless transitions in the four-phase protocol can happen in parallel with the settle time for the next data, so that four-phase bundled data is not intrinsically slower than two-phase. Figure 2.4 (c) shows some modifications to four-phase timings that have been used by the Amulet group [59, 108].

A variation on four-phase bundled data is the asP* protocol, presented by Molnar et al. [126]. The falling edge of the acknowledge signal is timed, and occurs three gate delays after the rising acknowledge. This has been demonstrated to give good performance in a FIFO with no processing logic.

Delay-insensitive schemes do not introduce timing and routing constraints, but require more circuitry. Brunvand [19] has classified DI schemes as follows. A pair of request/acknowledge handshakes R_0/A_0 and R_1/A_1 can be used to pass a single bit of data, by performing a handshake on R_0/A_0 for a binary 0 and R_1/A_1 for a binary 1. When used to send a number of bits between modules, this is termed a *four-wire* scheme. The A_0 and A_1 wires can be combined together on a per-bit basis, yielding a *three-wire* scheme where each bit has R_0, R_1 and Ack wires, or the acknowledges can be combined into a single wire for the whole bus, which is called a *two-plus-wire* scheme. Any of these approaches may be used with either two-phase or four-phase signalling, but the most common combination is four-phase signalling with two-plus-wire data, more commonly referred to as *dual rail*.

Dual rail data requires two wires per bit from sender to receiver, $R_0[0]/R_1[0]$,



(a) Two-phase bundled data

(b) Four-phase bundled data

(c) Some variations on four-phase used by the Amulet team

Figure 2.4: Two phase and four phase data

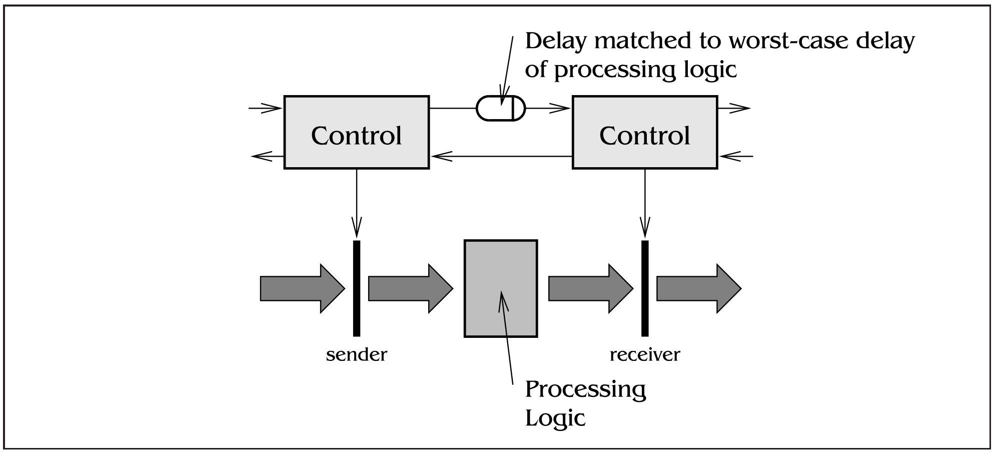


Figure 2.5: Bundled data with processing delay

...$R_0[n\text{-}1]/R_1[n\text{-}1]$, and an acknowledge wire from the receiver to the sender. All wires start at logic 0. A transaction consists of raising one of each pair of wires $R_0[i]$ and $R_1[i]$, raising the acknowledge when the all data bits have been received, dropping all data lines when the acknowledge has been received, and then dropping the acknowledge.

NULL Convention Logic is a proprietary dual-rail methodology using neuron-like gates, described by Fant and Brandt [55] and based partially on the work of Seitz [161]. It is a more structured approach than dual rail, a fact which permits substantial gate-level optimization of circuits. Results were given in [167] for an asynchronous 2-D DCT chip, but because bit-serial addition was used and their router was not tailored for use with this logic, the results were poor compared to other designs.

2.2.3 Comparisons

Two-phase bundled data was first proposed by Sutherland [172], and has been widely used, for example in Amulet 1 [57]. The Amulet team found that two-phase latches are much larger than pass-transistor or Yuan and Svensson [200] latches, which both use four-phase control. Two-phase to four-phase conversion was found to be too expensive, so four-phase control was used throughout Amulet 2e. Day and Woods [47] state that their four-phase pipeline design is smaller, faster and more energy-efficient than a two-phase design.

Bundled data has the advantage that a standard synchronous datapath can be used, but this is also a disadvantage. Figure 2.5 shows how to insert processing logic between two stages of an asynchronous pipeline: a delay must be added to the control path so that the bundled data assumption will still hold at the receiver. This delay must be chosen so that it is longer than the worst-case delay through the processing logic, plus a safety margin of typically 100% [57], which substantially affects performance. It is no surprise that the Amulet team, who use bundled data throughout their designs, have shifted their focus from high speed to low power.

Dual rail datapaths consume more silicon area than synchronous or bundled data circuits, because there are twice the number of wires involved, but they can be quite efficiently implemented with Cascade[1] Voltage Switch Logic (CVSL) gates [189, page 170]. The request signal is embedded in the data, so no additional delays need to be added; the receiver simply waits until it sees valid data, and then it knows that the data processing is complete. Unfortunately, determining whether the data is valid requires looking at each bit in the datapath, which takes a large tree of gates. This *completion tree* takes time to produce a result, although this is unlikely to be as long as the additional delay margins imposed on bounded delay circuits. Cunning design, such as that used by Williams [190] and Martin [121], can take the completion delay off the critical path and allow a pipeline to run as fast as the data processing circuits will allow, which is simply not possible with either synchronous or bundled data design.

Hybrid approaches have been proposed that keep the small size of bundled data, but have performance near that of dual rail. Garside proposed an ALU for use in the Amulet processor that used a ripple carry adder with a dual-rail carry path and an external bundled data interface [61]. Completion of the addition is signalled a short time after all carry bits have been calculated. This gives reasonable performance while keeping size and power low. Another technique is Current-Sensing Completion Detection (CSCD), which uses the fact that CMOS gates only take power while they are switching. A circuit was suggested by Izosimov [78] which uses a resistor and an analogue amplifier as a current sensor to determine when processing has finished, but this causes a voltage drop to the rest of the logic that results in a 35% delay penalty. Grass and Jones gave a faster BiCMOS circuit [62]. Activity Monitoring Completion Detection is a more promising approach, given by Grass et al. in [63], where small circuits inspect the output of datapath gates and signal when the outputs are stable.

2.3 Graph-based specification approaches

Many ways have been proposed to specify the behaviour of asynchronous circuits; Figure 2.6 gives an overview of the more common styles. The specification type depends on the delay assumption, and affects the synthesis algorithms used. This section looks at specifications that are essentially graph-based, while text-based specifications are covered in the next section. Only deterministic circuits are considered.

2.3.1 Petri nets (PNs)

Petri nets, invented by C. A. Petri, are a graphical specification of processes that naturally depict causality, concurrency and choice. Figure 2.7 shows two examples of Petri nets. The open circles are *places*, the black circle is a *token*, and `x+`, `y+`,

[1] Often erroneously called Casc**o**de Voltage Switch Logic, probably because it sounds better. The *cascode configuration* is actually an analogue circuit designed to nullify the Miller effect (C_{cb}) in high-frequency bipolar amplifiers [76, page 103].

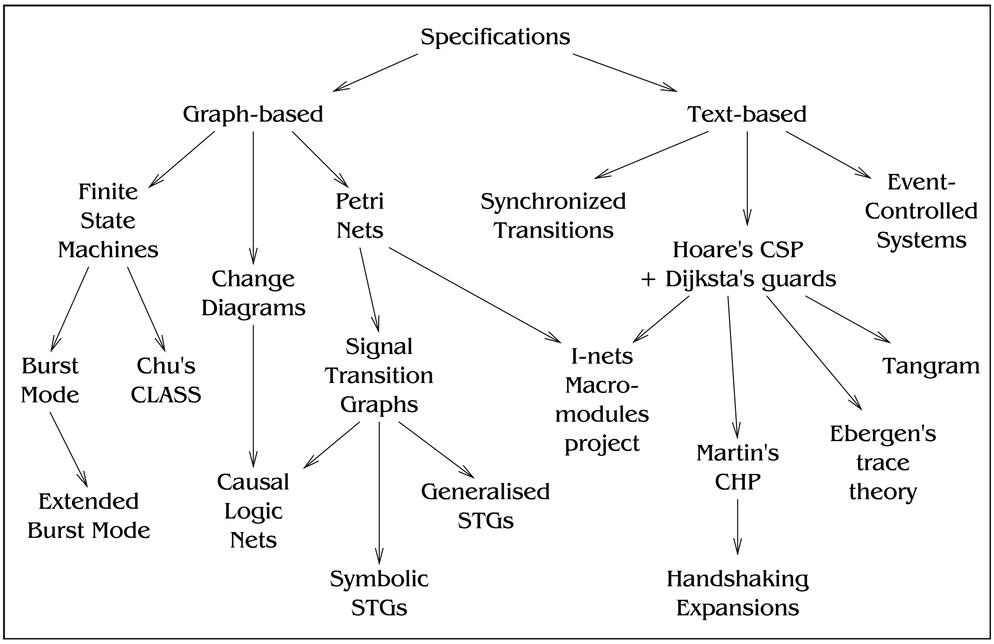
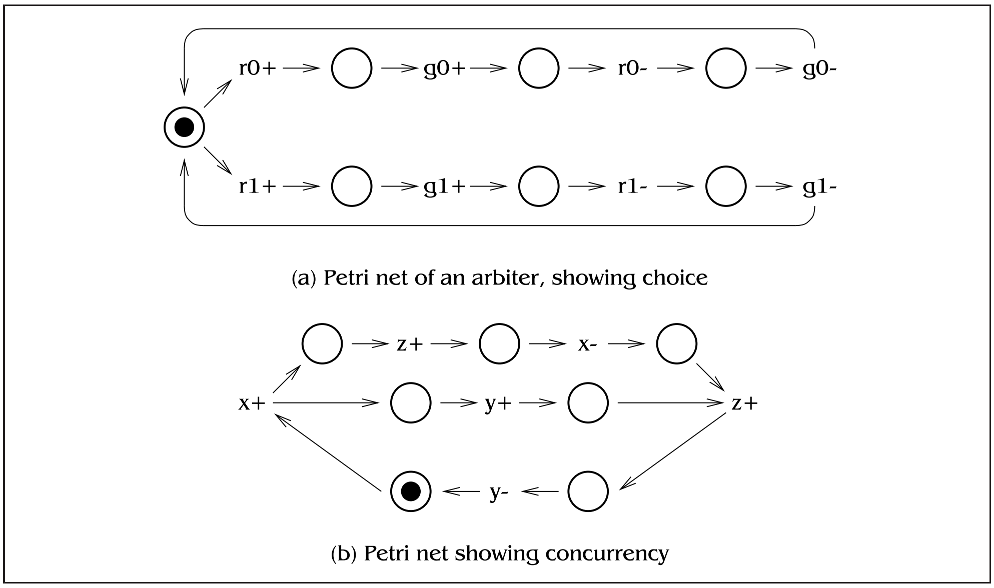


Figure 2.6: Overview of specification styles



(a) Petri net of an arbiter, showing choice

(b) Petri net showing concurrency

Figure 2.7: Petri net examples

z+, x-, y- and z- are transitions. The arrangement of tokens in the net is called its *marking*. The set •t of all places with arrows to a particular transition t is known as the set of *predecessor* places of that transition, similarly the *successor* places t• are those with an arrow from the transition. When a transition has at least one token in all its predecessor places, it can *fire*, removing one token from each predecessor place and adding one token to each successor place. A highly concurrent circuit with many internal states may correspond to a small Petri net. A good introduction to Petri nets can be found in Reisig [151].

Petri nets that are used to design four-phase circuits usually have their transitions labelled with + or -, but two-phase nets, such as I-nets, do not. Many known results about Petri nets were collected by Murata [135], from which the following definitions are taken.

A *Petri net* is a 4-tuple (P, T, F, M_0), where

$\mathrm{P} = \{p_1, p_2, \ldots p_m\}$ are the places
$\mathrm{T} = \{t_1, t_2, \ldots t_m\}$ are the transitions
with $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$
$\mathrm{F} \subset (\mathrm{P} \times \mathrm{T}) \cup (\mathrm{T} \times \mathrm{P})$ is the flow relation
$\mathrm{M}_0 : \mathrm{P} \rightarrow \{0, 1, 2, \ldots\}$ is the initial marking.

A *pure* net is one with no self-loops, i.e. no t and p such that $(t, p) \in \mathrm{F}$ and $(p, t) \in \mathrm{F}$. Self-loops can be turned into two-transition loops by adding a dummy transition, if required. A net is *k-bounded* if no place can contain more than k tokens during any sequence of transition firings from the initial marking; a 1-bounded net is also called *safe*. Nets which are not bounded can not necessarily be implemented as a finite circuit. A net is *live* if every transition can be fired infinitely often from the initial marking.

Direct structural synthesis of a Petri net is possible, translating either places or transitions into circuit constructs. In general, structural methods make large and slow circuits, but they can be useful for rapid prototyping. Direct place translation has an SR flip-flop for each place in the net. An AND gate connected to all the predecessor places of a transition goes high when the transition is enabled, and then is used to set all the successor places and reset the predecessors [184]. A better approach is to use a circuit element for each transition, as used by Patil and Dennis [48] and later refined by Kishinevsky et al. [89]. This can only be used for two-phase circuits, and places a number of restrictions on the net.

Petri nets are a general specification style with few restrictions, so not all nets can be turned into actual circuits. Certain conditions need to be met by a net before it can be synthesized, such as persistency and consistent state assignment. A non-input transition t is *non-persistent* if there is a reachable marking in which it and another transition u are enabled, but firing u disables t. This behaviour might cause a hazard if t is part-way through firing when u fires. A *persistent* net is one with no non-persistent transitions; note that this is not the same as Chu's definition of STG persistency [28], even though STGs are a restricted class of Petri nets. Consistent state assignment means that a particular signal a must alternate a+, a-, a+, a-, ...

There are two ways to determine whether these conditions hold. Kondratyev et al. have used net unfoldings [95], and also conducted an implicit state space search by using binary decision diagrams [92]. Usually, when one algorithm takes a long time, the other will be much more efficient for a particular net; they are essentially complementary techniques.

Because Petri nets are such a general specification, other forms can be translated into a Petri net and then synthesized. Kishinevsky et al. [88] have translated *transition systems*, a superset of state graphs, to Petri nets; this is a generalisation of earlier work by Cortadella et al. [40]. Translation of circuits into *circuit Petri nets* and resynthesis to optimize the circuit was discussed by Kondratyev et al. [92].

Many Petri net transformation and synthesis algorithms have been implemented in the tool `petrify` [37], which produces speed-independent circuits. Synthesis using `petrify` is similar to the STG synthesis that will be presented shortly. A state graph is formed, which has CSC violations removed by adding state variables based on the theory of regions [38]. A region is a constrained set of states in the state graph that will preserve speed-independence if it is used as a rising or falling condition for a state variable. CSC violations are removed iteratively, then standard logic synthesis algorithms are used to produce CMOS complex gates. The gates produced tend to be reasonably small, and are usually in a gate library [92]. A good overview of Petri net methods is given by Kondratyev et al. [92].

I-nets

It has already been said that useful delay-insensitive circuits cannot be built out of single-output gates, and that a set of basic multi-output modules is required. I-nets are a specification, very similar to Petri nets, that are designed to specify these modules. The modules are small, which allows exponential synthesis algorithms to be used, such as the traditional FSM algorithms of Tracey and Unger. Typically, two-phase signalling is used. I-nets were used in the Macromodules project by Clark and Molnar in the late 60s and early 70s. The aim of the project was to create "building blocks . . . from which it is possible for the electronically naive to construct arbitrarily large and complex circuits that work" [32]. Modules were boxes with about 80 MECL-II chips in each, plugged into a power and cooling backplane and attached to each other with data cables.

Synthesis from an I-net proceeds by exhaustively simulating all transitions to get a two-phase interface state graph (ISG), converting this to an equivalent four-phase diagram called an encoded interface state graph (EISG), and then using standard logic synthesis techniques, such as Karnaugh maps. State variable insertion, if required, is done by hand. A full description can be found in Sproull [169]. The circuits produced may not be delay-insensitive, but by analysing hazards and adding inertial delays if necessary, the circuits can be made to compose correctly in a DI setting.

In 1994, Sutherland et al. [171] described the synthesis of a number of pipeline latch controllers, using fragments of Petri nets which they called *snippets*. Figure 2.8 shows the snippets used, and Figure 2.9 gives the resulting circuit. It can be seen that this is a combined two-phase and four-phase methodology, using XOR

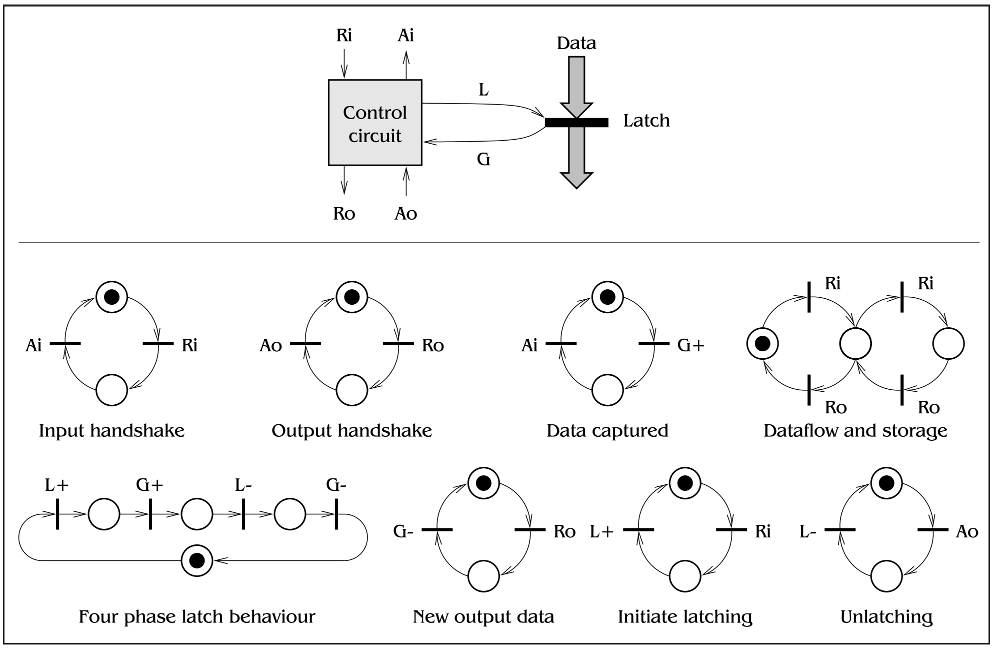


Figure 2.8: Snippets specifying the medium capability latch controller of [171]

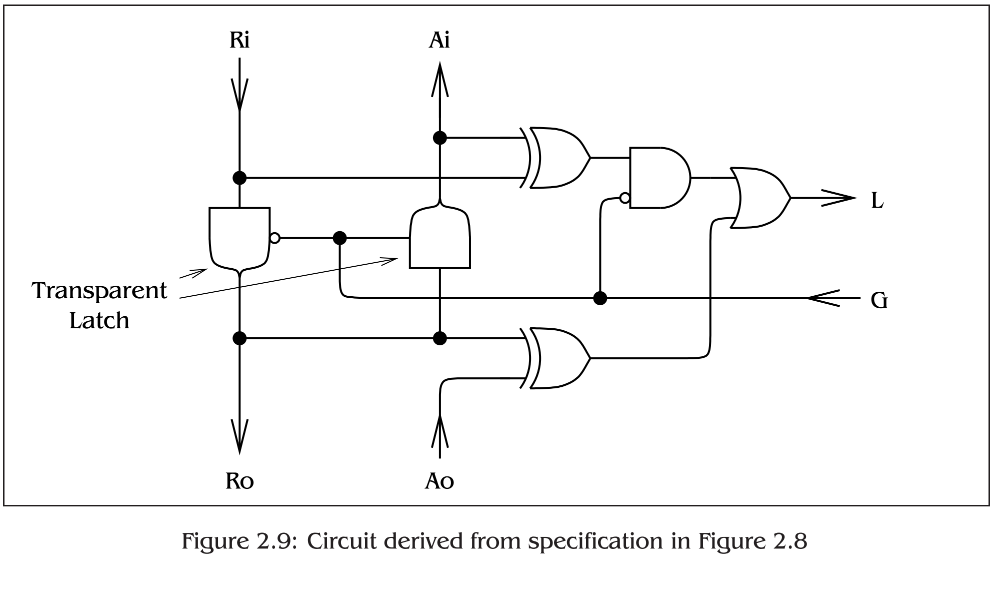


Figure 2.9: Circuit derived from specification in Figure 2.8

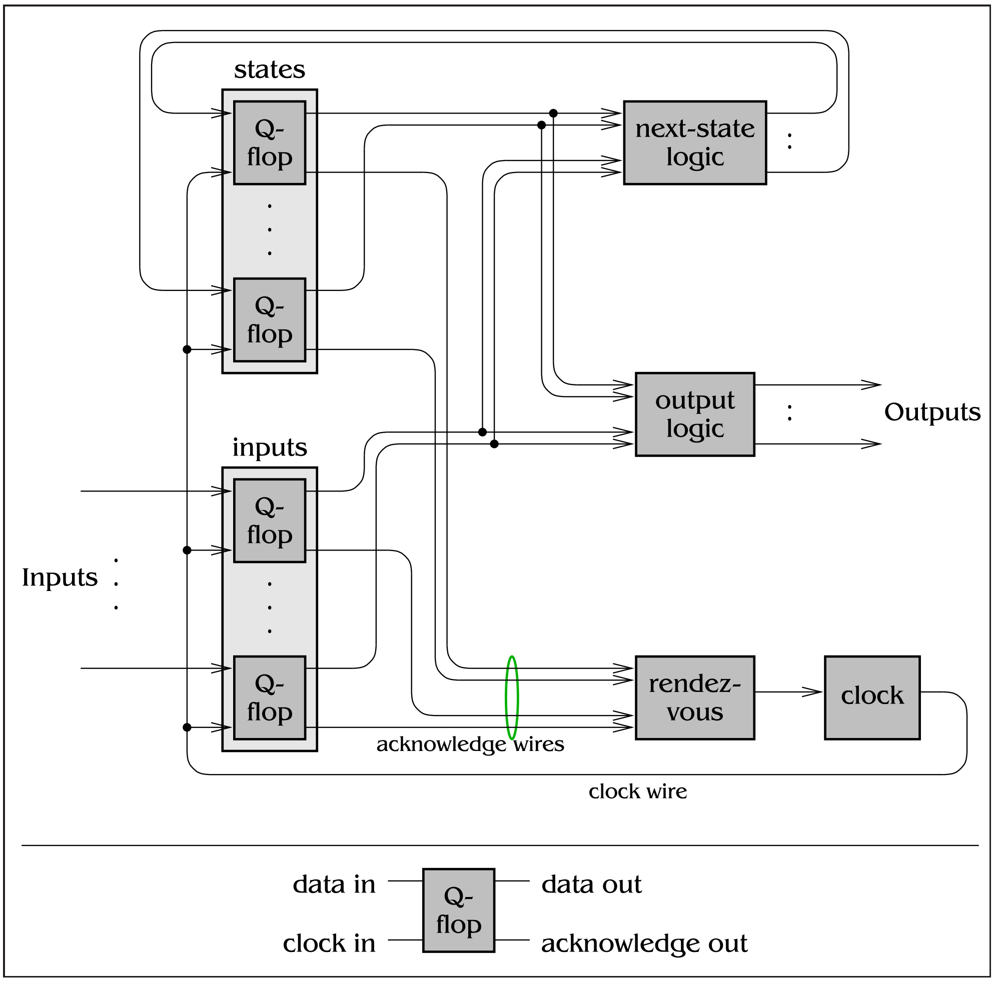


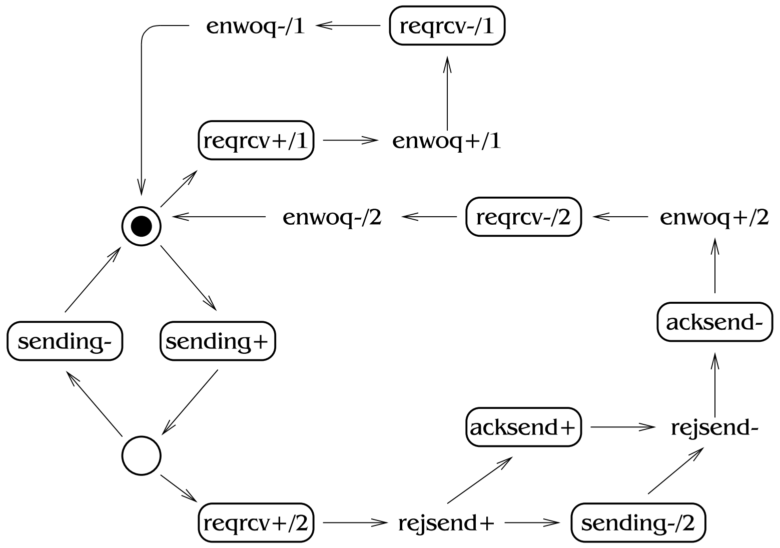
Figure 2.10: Q-module implementation style

gates to convert one way and wait-ons, also called transparent latches, to convert the other way.

Rosenberger et al. [154] implemented I-nets as Q-modules, which are internally clocked state machines with the ability to stretch the clock period if a circuit element goes metastable. Figure 2.10 shows the Q-module implementation style. A Q-flop is a clocked data latch, with a built-in arbitration circuit, that will only send an acknowledge when its output is stable. Q-modules can provide a compact way to implement large specifications, but the clock is always cycling, so they take a fair amount of power and have an indeterminate latency.

Time Petri nets (TPNs)

Time Petri nets are Petri nets that have rational earliest and latest firing times $(t{+}e_{\mathtt{x}})$ and $(t + l_{\mathtt{x}})$ associated with every transition x. When x becomes enabled, it must wait until time $(t + e_{\mathtt{x}})$ before firing, but must have fired by time $(t + l_{\mathtt{x}})$. TPNs



According to the usual STG convention, two different transitions of the same wire in the same direction are distinguished by appending `/1` and `/2`. This STG is live, safe, free-choice and has the USC, CSC and CSA properties, but does not have single-cycle transitions, because of `reqrcv+/1` and `reqrcv+/2`.

Figure 2.11: Example of an STG: `rcv-setup`

were used by Semenov [163] to reduce the size of the state graph corresponding to the Petri net, and hence make a simpler circuit. The state graph can still be large, especially for highly concurrent specifications. This problem was addressed by Verlind et al. [186] by allowing tokens in the state graph to have negative ages. This has the effect of folding many different possible orderings of concurrent signals into one ordering, but the negative ages mean that any one transition can actually have fired before the others.

2.3.2 Signal transition graphs (STGs)

STGs were proposed by Chu [28] as interpreted live safe free-choice Petri nets. Liveness and safety of Petri nets has been covered already; an interpreted net is one that has signal names associated with all its transitions, and a net is free-choice if for any two transitions $\mathtt{s}$ and $\mathtt{t}$ such that $(\bullet\mathtt{s} \cup \bullet\mathtt{t}) \neq \emptyset$, then $\bullet\mathtt{s}$ and $\bullet\mathtt{t}$ are both a single place p. When STGs are drawn, any places that have one arc in and one arc out are removed, making the representation more compact. An example STG showing concurrency and choice is shown in Figure 2.11. Input transitions are usually distinguished; here they are ringed. STGs are designed to be used to specify small modules, and are not useful for system-level design.

An equivalent specification, the *signal graph*, was proposed by Rosenblum and Yakovlev [155]. Signal graphs are less constrained than STGs—all graphs that can be meaningfully interpreted are regarded as correct—and allow timing information to be included in the specification, with constructs like a+ ⟶ T(50ns) ⟶ b+. However, no synthesis algorithms were presented in [155], so Chu's STGs came to dominate.

The advantage of STGs is that Chu provided polynomial-time synthesis algorithms, in contrast to the exponential time taken for general Petri net synthesis. His STG *contraction* algorithm means that the logic for a signal can be synthesized by looking at only a small part of the original STG. The disadvantage of these and other fast methods is that the STG must obey certain conditions before they can be used, such as liveness, safety, consistent state assignment (CSA), unique or complete state coding (USC/CSC), STG persistence, and single-cycle transitions. An STG with *consistent state assignment* has its signals alternating up and down, ie. a+,a-,a+. A graph with *unique state coding* does not have two reachable markings with the same value of all signals in both. A graph with *complete state coding* is allowed to have two markings with the same values on all signals, as long as the enabled non-input transitions in both markings are the same; i.e. if the circuit does not know what state it is in, it does not need to know. A *persistent* STG is one where, for every arc between transitions t* ⟶ u* where u is a non-input signal, there is a sequence of arcs ensuring that u* happens before the next transition of t. Finally, a net with *single-cycle transitions* has, for each signal, only a single rising and single falling transition of that signal.

Several of Chu's conditions have been criticised for being over-restrictive. Persistency was shown to be unnecessary by Lavagno et al. [104], who gave an example of a non-persistent specification that is clearly implementable. STG persistency, which implies PN persistency, was shown by Puri and Gu [149] to be related to CSC rather than implementability. Yakovlev [195] outlined several features of STGs that he considered to be too limiting. An example of an unsafe STG was given that was obviously implementable, and other reasons were given why free-choice was too restrictive, and why coloured tokens and non-binary signals should be introduced. Multi-valued signals are allowed in *symbolic STGs* [193], which are said to be useful for high-level specification, but they need to be translated into binary STGs before synthesis.

Generalized STGs are a specification used by the synthesis tool ASSASSIN [199], and are a superset of STGs. Boolean guards are allowed on arcs, which have the meaning that a token can only flow along an arc if the guard is true. Additional transition types are x~, meaning x toggles state, x&, meaning x becomes stable, x^0 and x^1 mean x goes to 0 and 1 respectively, and x* means that anything can happen to x.

Timing constraints were added to STGs by Myers and Meng [137], for the same reasons that time Petri nets were later considered. An example timed STG is shown in Figure 2.12. This approach requires post-layout verification of delays to make sure that the original timed STG had its internal delays correct. Note that in Figure 2.12, the delay constraints turn out to be equivalent to the fundamental mode assumption: all output and internal signals are faster than all input signals. This work was later extended to allow nondeterministic environment behaviour [7].

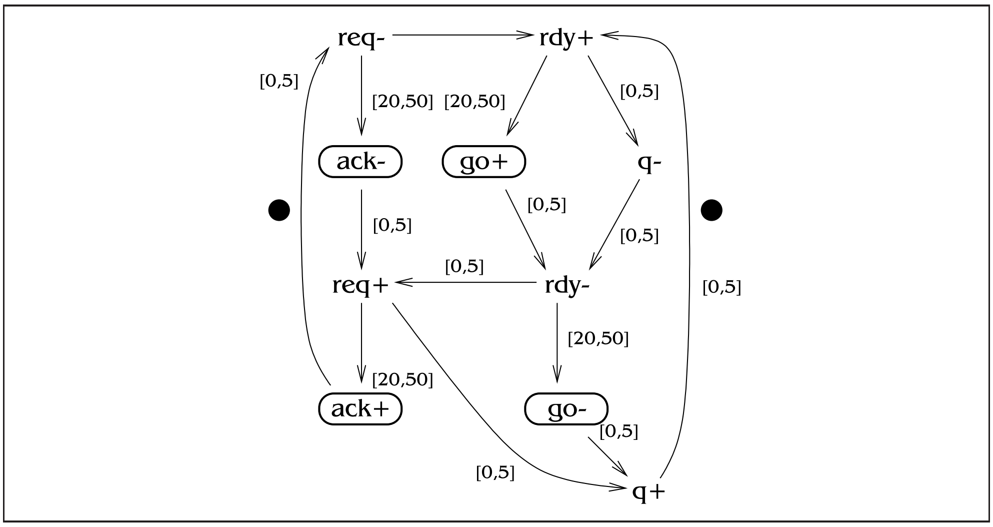


Figure 2.12: An example timed STG from Myers and Meng [137]

When an STG specification contains a choice between two output or internal signals, called an *internal conflict*, then metastability may arise in the circuit produced. This can either be dealt with by adding special analogue components, as suggested by Chung and Kleeman [31], or by factoring out an arbitration module from the original STG producing a new STG without conflicts, as proposed by Cortadella et al. [44, 43].

STGs have also been used for verification; Yakovlev et al. [191] gave a framework that treats specifications and circuits as instances of the same kind of object. Conditions that are required for a correct implementation to exist can then be checked on the STG, and then on the synthesized circuit to verify that the properties were preserved during synthesis.

State variable insertion

Unlike I-nets, STGs aimed to automate the synthesis process. If the USC or CSC conditions do not hold for an STG, state variables must be added to the specification to distinguish between markings where all signals have the same value. Several methods for doing this have been proposed.

Vanbekbergen et al. gave the first totally general STG state assignment algorithm [183], but it was often prohibitively time-consuming. Their first algorithm, using *generalized lock classes* or GLCs, ran in time $O(n^5)$. The second algorithm used graphs to determine stronger conditions, which would imply that all signals were in a GLC and hence that USC was satisfied, and was implemented in ASSASSIN. This algorithm would often add arcs in the STG that would reduce concurrency, so steps were taken to ensure that the circuit performance was not reduced by too much.

Tracey's algorithm is one of the first state assignment algorithms for asynchronous circuits, but it works on a finite state machine (FSM) representation. Lavagno et al. [102] formed an FSM from an STG specification, used Tracey's algorithm to insert state variables, and then went back to the STG and inserted rising and falling transitions of the new state variables where appropriate.

Recent work centres of the concept of *regions*, described by Cortadella et al. [42]. This technique works on the state graph, where the addition of a new signal requires finding sets of states where the variable will be stable at 0, switching to 1, stable at 1 and switching to 0. A region is a set of states which has the property that a new variable using this set as its switching-to-0 or switching-to-1 set will preserve the speed-independence of the specification. Regions are based on earlier work by Ykman-Couvreur and Lin [198], but have a finer granularity which allows better solutions at the expense of a longer computation time.

STG synthesis methods

One of the main contributions of Chu's Ph.D. thesis [28] was his contraction algorithm. Using this, synthesis from STGs proceeded in three stages. For each output x,

- The net was contracted to only include transitions that directly affect the switching of x,
- A state graph was formed from the contracted net,
- A Karnaugh map was drawn from the state graph and logic derived.

The delay assumption used was that arbitrarily large CMOS gates could be built that behave as if there is a single lumped delay at their outputs, the *unbounded complex gate* assumption.

SIS is an early STG synthesis tool which was formed by collecting together a number of results in asynchronous circuit synthesis [103, 164]. The authors of SIS felt that delay-insensitive design was too pessimistic and restricted, speed-independent design was not realistic, and fundamental mode had significant problems with multiple input changes. SIS starts from an STG specification, which is required to be live, safe and free-choice for the benefit of algorithms known at the time. The circuits produced function under either the unbounded gate or bounded wire delay models. SM flip-flops, equivalent to SR flip-flops, are used as the state-holding elements with a pair of set/reset gates that are static-hazard-free by construction. The flip-flops are assumed to be relatively immune to dynamic hazards. Hazard analysis is performed, and as a final step, delays can be added on wires to remove any remaining hazards. The addition of delays in the circuit can reduce performance; Beerel [6] reported that his speed-independent circuits were typically 25% faster than SIS.

The aims of *ASSASSIN* [199] were to build a unified synthesis environment on top of SIS, where different specification types can be synthesized using the same methods. The high-level specification is the generalized STG, which is then translated into the lower level state graph before synthesis. Other specifications could also be converted into a state graph. Much of the synthesis was inherited from SIS,

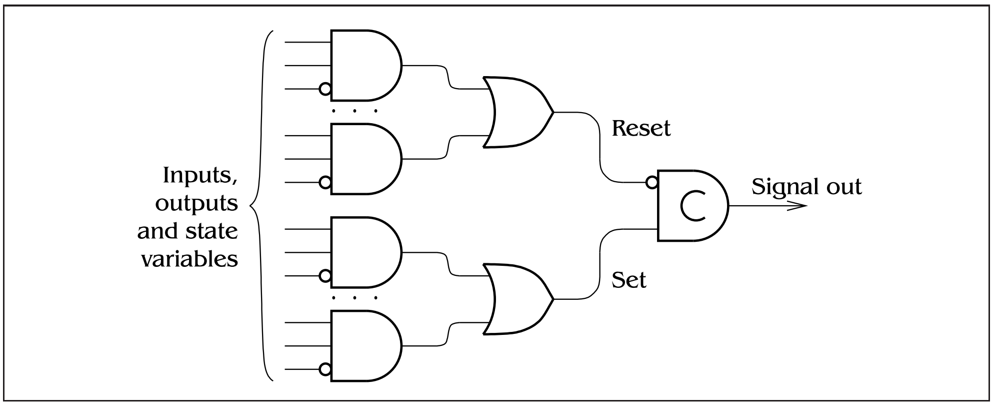


Figure 2.13: Implementation style used by Beerel [6] and Kondratyev et al. [94]

although there were several additions, such as concurrency reduction and new state assignment algorithms. A new state-holding element called the *MHS flip-flop* was used, which has an internal hazard filter to remove delays on outputs.

Beerel's synthesis tool *SYN* [6] takes determinate state graphs and produces almost speed-independent circuits. The determinate condition on state graphs is very similar to persistency on Petri nets. Beerel's Standard-C implementation strategy is shown in Figure 2.13. State holding C-elements are used at primary outputs. Each of the AND gates on the left of Figure 2.13 corresponds to a connected set of states in the state graph; if these sets are all disjoint, then there are no delay hazards in the logic shown. The input bubbles on the C-elements mean that the circuits produced are not quite speed-independent, but if it is assumed that inverters are fast, then the circuits will not fail. Kondratyev et al. [94] described a similar synthesis style at the same time as Beerel, but also developed a style using an SR flip-flop as the state-holding element. This has the advantage that no bubble is required on the reset input, and complemented outputs are available which reduces the number of bubbles required on the AND gates, but loses the advantage of the C-element, which is that the output is the last moving point.

Recently, interest has been shown in synthesis methods that can be used for very large STGs. Semenov et al. [162] described an algorithm that uses unfoldings to find relationships on the places that must be occupied in the STG before a given signal can fire. Using this localised condition for synthesis produces an algorithm that is only singly exponential in the size of the STG, rather than doubly exponential as for most other algorithms. The algorithm is implemented in the synthesis tool *PUNT*. This builds on earlier work on unfoldings by Pastor et al. [143], which gave polynomial-time algorithms for the synthesis of free-choice STGs, apart from one stage of the algorithm which was exponential but empirically fast. Unfoldings were also used for fast synthesis by Miyamoto and Kumagai [125], but were called Occurrence Nets.

PNIT is a TclTk framework that integrates the tools `petrify`, SIS and PUNT. It supports a number of file formats and provides a common interchange format, PNIF. It is capable of both asynchronous and synchronous design.

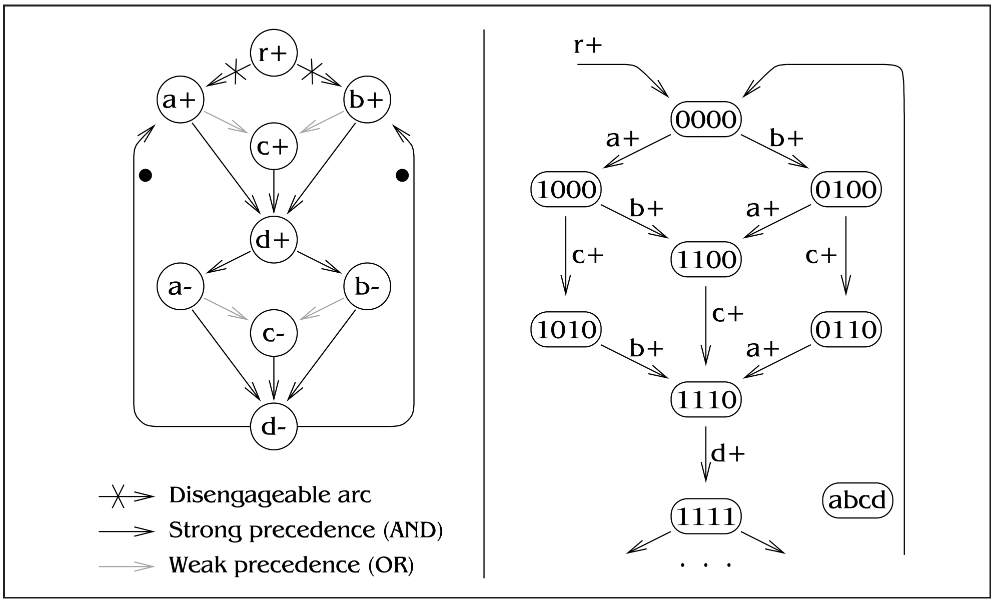


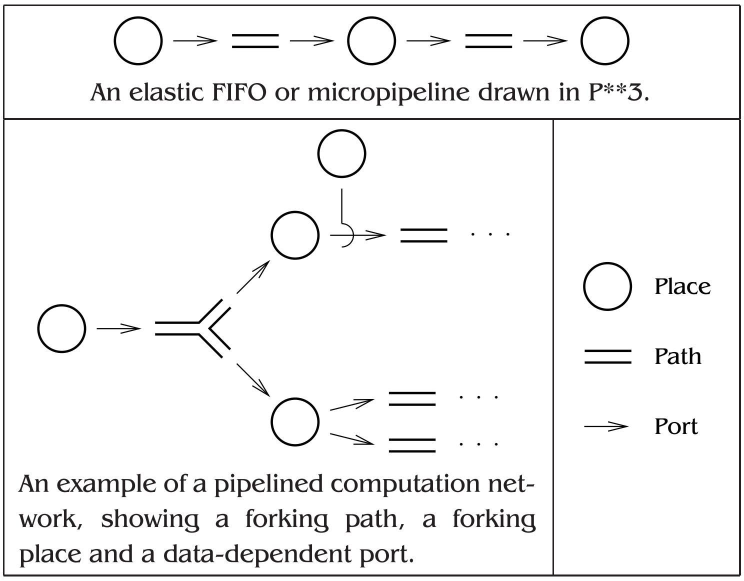
Figure 2.14: An example change diagram from Hauck [69] with part of its state graph

2.3.3 Change diagrams

Change diagrams (CDs) [89] are a similar specification to STGs, but where STGs use places to represent OR behaviour, change diagrams have another kind of arc, called a *weak precedence* arc. There are no places in a change diagram, so they cannot describe choice behaviour. Their restrictions do however permit polynomial time synthesis algorithms.

An example of a change diagram is shown in Figure 2.14, where the weak precedence arcs are drawn grey, and the blacks arcs function as in STGs. When either a+ or b+ have fired, then c+ can fire. If a+ fires and then c+ fires, a token gets taken from both input arcs to c+, leaving a token debt on the arc from b+ to c+. This debt will be cancelled out when b+ fires, unlike the OR operation in STGs, when b+ firing would cause another c+ to fire. An effect of this is that change diagrams cannot specify an XOR gate, but STGs can. The crossed arcs in Figure 2.14 are disengageable arcs, which fire exactly once and are then removed. They are used for non-repeating initialisation behaviour. Although OR arcs appear to add flexibility to the model, a result by Kishinevsky et al. [89] reduces diagrams with OR arcs to ones without them to simplify synthesis, so they do not add anything significant.

Causal Logic Nets were introduced by Yakovlev et al. [192] as a specification combining the best features of STGs and change diagrams. CLNs have places and transitions, as STGs and Petri nets do, but the firing rule is a boolean function of input places rather than the simple AND rule of STGs. A result in [192] states that an STG that is observation equivalent to a change diagram with OR behaviour must be non-free-choice and either non-persistent or unsafe, although an earlier report by Yakovlev [195] stated that safety and free-choice were overly restrictive STG characteristics. CLNs look to be overly powerful for current specification needs.



An elastic FIFO or micropipeline drawn in P**3.

An example of a pipelined computation network, showing a forking path, a forking place and a data-dependent port.

Figure 2.15: The P**3 primitives and an example of their use

2.3.4 P**3

P**3 (pronounced "P cubed"), described by Coates et al. [35], is a development of Sutherland's earlier micropipeline work [172] to allow data-dependent pipelined computation. P**3 is a graphical specification of networks of pipelines which can be used to design complete systems, unlike the other graphical specifications here which deal with small control circuits. P**3 takes its name from the three types of module used in circuits: *places*, where data resides between computations, *paths*, where computations happen unconditionally, and *ports*, which are connecting structures that can be used to steer the data conditional on other data values. A few P**3 structures are drawn in Figure 2.15. Synthesis of a P**3 specification consists of translating each construct into a circuit module. Examples of these modules were given by Coates [35] for a two-phase implementation with the emphasis on speed rather than robustness, although a wide variety of formalisms could be used at the gate level. For example, the four-phase asP* protocol has also been used to implement P**3 specifications.

2.3.5 Burst mode

Burst mode was developed by Davis et al. as a way to avoid the limitations of fundamental mode. Fundamental mode circuits require a settling period after every input, and as stated in Section 2.1, there are two values δ_1 and δ_2 such that two inputs separated by a time more than δ_1 but less than δ_2 will cause the behaviour of the circuit to be undefined. If two inputs are intended to arrive concurrently from two different sources, it is impossible to guarantee that there will not be this critical time difference between them, so fundamental mode cannot be used.

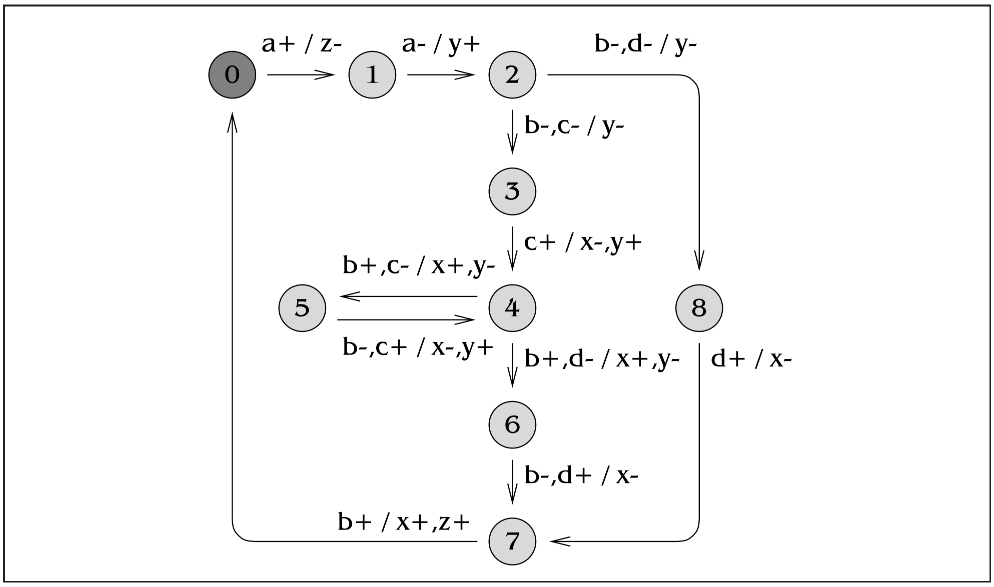


Figure 2.16: Example burst-mode diagram: `isend`, from Yun [202]

Burst mode sidesteps this problem by making sure that, during the receipt of one or more out of a number of concurrent input transitions, no internal state change will occur. The settling time is therefore effectively zero for these transitions, so $\delta_1 = \delta_2 = 0$ and no failures can occur. State changes are only allowed after *all* transitions in the concurrent set of inputs has been received. A set of input transitions that will occur concurrently is called an *input burst*; similarly, outputs are grouped together in an *output burst*. The burst-mode assumption is that, after providing an input burst, the environment must wait until it has received the whole of the corresponding output burst before it can send another input burst. Hauck [70] believes that it may be difficult to meet the burst mode constraints on the environment in all cases, which seems plausible, although no groups have reported this.

An example burst-mode diagram is shown in Figure 2.16. This represents a finite state machine, with 0 as the initial state. Arrows between states are labelled as "input burst / output burst". When an input burst is received, the corresponding output burst is fired and the machine moves along the arrow. For example, in state 2, the machine waits for `b-` and one out of `c-` and `d-`; if `b-` and `c-` were received, `y-` is fired and the machine moves to state 3, similarly `b-` and `d-` cause `y-` and a move to state 8. The machine must always be able to tell which arc to take out of a state, so the arc from state 2 to state 8 could not be labelled simply `b-/y-`; after receiving `b-`, the machine would not be able to decide whether state 3 or state 8 is next.

Burst mode was designed to allow the creation of large systems, and also to give fast circuits [34]. The Post Office, a large communications chip for the Mayfly project at HP Labs, showed that it achieved both of these goals. However, burst mode limits concurrency, because the only allowable concurrency is within an input

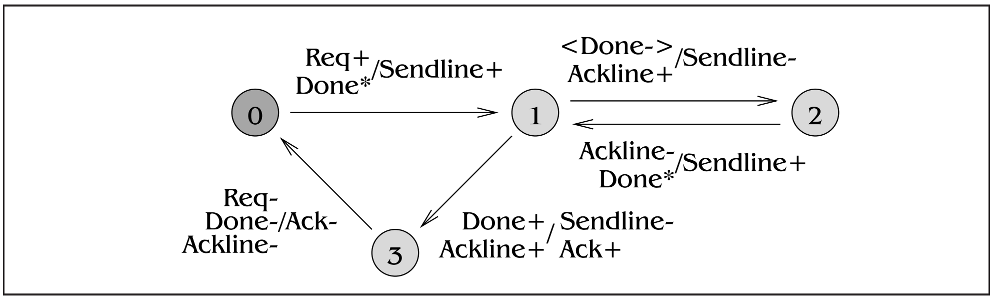


Figure 2.17: Example extended burst-mode diagram: `sbuf-send-pkt2-core`

burst. Yun [203] stated that a moderate degree of concurrency is essential, but that fine-grained concurrency is not necessary in most applications. Extended burst mode (XBM) adds a construct called the *directed don't-care*, written `x*` for an input `x`, which allows certain inputs to change concurrently with outputs. When `x*` is written, its meaning depends on the next up-going or down-going transition of `x` that can be reached along a path in the XBM specification. If the next transition is `x+`, then `x*` means that `x` can remain stable or make a low-to-high transition, and the reverse for `x-`. The value of a signal that might or might not have changed at a particular time can be read by using ⟨`x+`⟩ for "`x` is high" and ⟨`x-`⟩ for "`x` is low". In text files, ⟨`x+`⟩ is written as `x#` and ⟨`x-`⟩ as `x~`. An example XBM specification from Ykman-Couvreur et al. [199][2] is given in Figure 2.17.

Large burst mode machines may be partitioned using the work of Kudva et al. [97], although one of the strengths of burst mode is that large machines can be built when local clocking is used (see Section 2.3.5), and partitioning does not appear to guarantee that the burst mode environment constraints are still met.

Synthesis from burst-mode specifications

The first burst-mode synthesis tool was *MEAT*, written in LISP by Davis, Coates and Stevens at HP Labs [33, 45]. The burst-mode specification is first converted to a flow table, then semi-automated state minimization carried out, followed by state assignment using Tracey's algorithm, and finally a circuit produced for each state assignment using a modified Quine-McCluskey algorithm and the best circuit picked. The state-holding elements are CMOS complex gates with a large output inverter to give a good drive capability. All wire forks are kept away from module interfaces to ensure modularity. Useful additions to the tool would have been automatic layout of gates on silicon and the use of standard cells, according to Davis [45].

MEAT made no efforts to avoid hazards; it just used a verifier on the resulting circuit and added delays by hand to remove any hazards found. Nowick [140] looked at changing the MEAT synthesis strategy to make circuits that were hazard-free by

[2] This diagram appeared on page 23 of [199], but with `Done+` replaced with ⟨`Done+`⟩, which I can only assume is an error, because that does not agree with the XBM rules that are given on the same page.

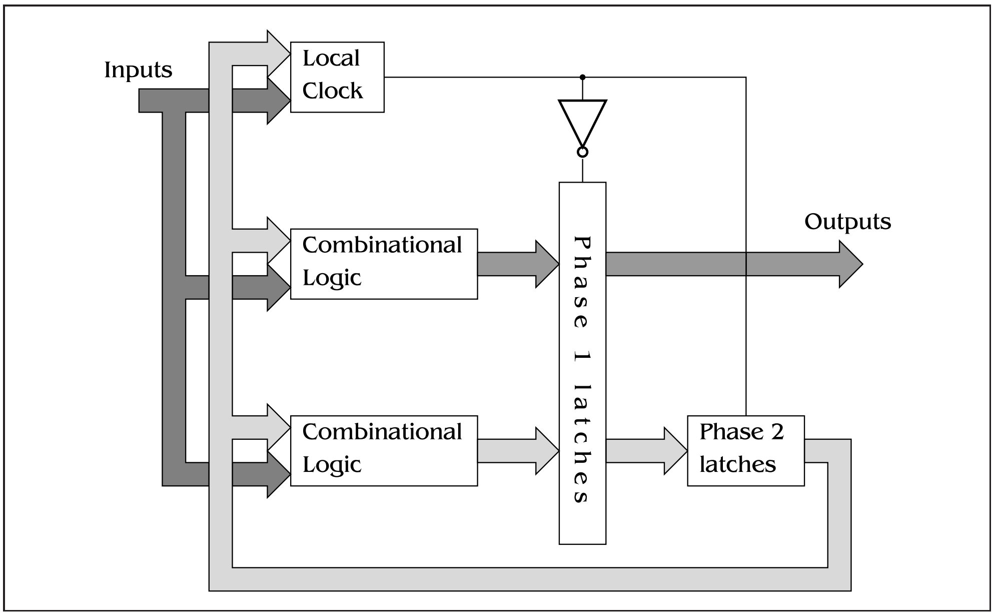


Figure 2.18: Local Clocking synthesis style

construction. If state minimization is not done, it is always possible to avoid hazards at the synthesis step, but hazards may be introduced during minimization if *state splitting* occurs, when one row of the original flow table is included in more than one row of the reduced table (see Section 2.6.1 for details of the minimization procedure). State splitting may violate the *stability* condition, which says that a burst-mode machine must remain stable in a particular state until the whole of an input burst is received. Coates later modified the minimization algorithm to be stable. Nowick's approach to minimization was to repeatedly merge two states that would not cause a hazard, until no more states could be merged.

Nowick et al. [141] also proposed the synthesis of burst-mode machines using local clocks. This work builds on the concept of self-synchronizing circuits due to Rey and Vaucher [152] and later Unger [179]. Figure 2.18 shows the architecture used. The internal clock can be used to embed a synchronous state machine in an asynchronous framework, which allows the use of fewer state variables and partially avoids hazard considerations. Hazards must still be avoided on the clock line. A cache controller was built in the local clocking style, and found to have a latency about half that of a comparable synchronous design.

Yun's *3D* synthesis tool [203] improved on earlier work in two areas: it used the XBM specification to get increased concurrency, and used a new state assignment algorithm. 3D circuits are of the classic Huffman type; they are a block of combinational logic with delays on feedback paths. No latches or C-elements are used, and no delays are placed on primary outputs. Next-state values for output and state variables are placed in a 3-dimensional K-map, with inputs and outputs labelling the x and y axes, and state variables on the z axis. A *layer* is a set of states in the x-y plane. Synthesis starts at state 0 and proceeds by filling in entries in the K-map

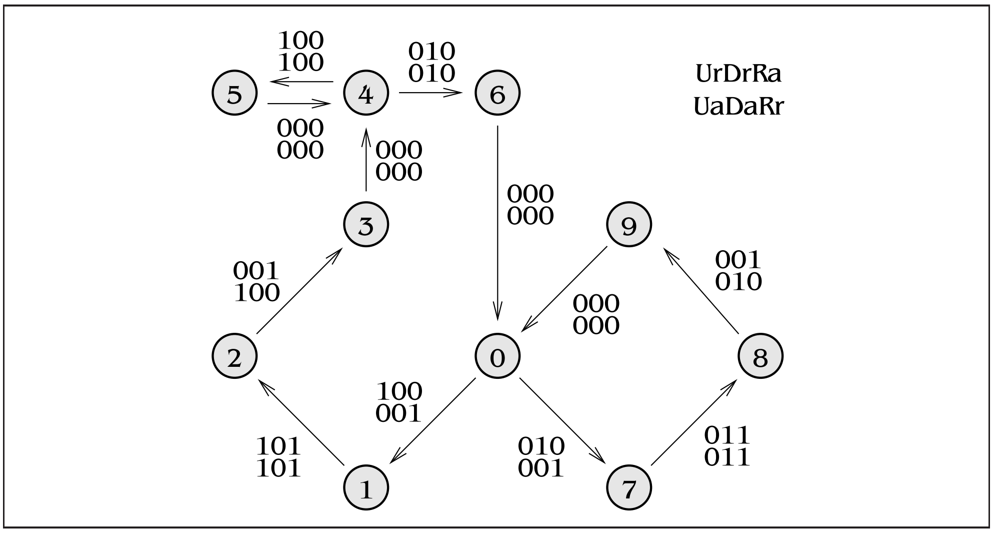


Figure 2.19: AFSM synthesis style used by Chu's CLASS [29]

corresponding to the transitions in the XBM specification, keeping within a single layer. If this causes a contradiction when moving from state p to state q in the specification, then the algorithm backs up to the point that state p was entered, and makes a change in the state variables so that the machine moves to a free layer. The transition from p to q can now be filled in on the new layer. State changes can either occur after output bursts or concurrently with them; in occasional circumstances, hazard avoidance may require the state change to come before the output burst. The hazard avoidance and improved specification style effectively render the other burst mode tools presented here obsolete.

2.3.6 Other FSM-based methods

Hollaar gave an implementation strategy [74] that used a 1-hot encoding in an attempt to remove the fundamental mode restriction. It was only partially successful, as pointed out by Hauck [70]; time is still required for a Hollaar circuit to stabilise, although that time is two or three times less than the fundamental mode assumption would indicate.

Chu gave a list of reasons why FSMs are bad for synthesis in [27]; in particular there are problems specifying concurrency so it is difficult to compose two FSMs to get another FSM, and state assignment is needed which can causes races and failures if done badly. However, synchronous designers are familiar with FSM specifications, so Chu designed a synthesis tool using an FSM front-end to his earlier STG work [29]. An example of one of Chu's asynchronous FSMs, or AFSMs, is given in Figure 2.19. It can be seen that there is no concurrency in Figure 2.19; no pair of transitions can fire simultaneously. This sequentiality is likely to give a large performance loss in the resulting circuits. It is interesting to note that this specification is almost identical to *blue diagrams* which are a main topic of this dissertation.

Operator	Written as	Notes
Input	a?	
Output	a!	
Concatenation	a;b	Do a then b
Union	a|b	Do a or b
Repetition	*a	ϵ, a, aa, ...
Prefix-Closure	pref(ab)	ϵ, a, ab
Projection	abc$\downarrow$\{a,c\}	ac
Weave	abd||acd	abcd or acbd

Figure 2.20: Permissible operations in Ebergen's Trace Theory

Peyton Jones also used an AFSM as a specification in [147], with a fast provably correct synthesis procedure targeted at PLAs. The design process was only partly automated, and no results were given.

2.4 Text-based specification approaches

The alternative to graph-based specification is text-based specification, where the proposed circuit behaviour is coded in a particular formal language. Text-based specifications tend to be more cryptic and have stricter delay models, such as DI or QDI, but it is often easier to prove results about the synthesized circuit. Several approaches are based on a combination of Hoare's Communicating Sequential Processes (CSP) [73] and Dijkstra's guarded commands [51], although many languages have been subverted for the purpose—even C++ has been used [67] as a hardware design and simulation language, using a thread library to get concurrent behaviour.

2.4.1 Ebergen's trace theory

This was an early approach to DI circuit synthesis, using a language similar to regular expressions. Circuits were built from a number of basic modules. The behaviour of these modules was written down in the trace language, then syntax-directed translation used to implement a given specification as an interconnection of basic modules, a process called *decomposition*. It can be proven that the circuit formed is equivalent to the given specification. A condition called *progress* ensures that all possible traces will occur, so that deadlock is impossible. The result is a clean, theoretical basis for DI design. Problems with the C-element later led to the inclusion of an isochronic fork in the basic module set, making the delay assumption QDI.

Table 2.20 gives the operations in Ebergen's trace theory. Prefix-closure is used to indicate that a circuit will always have fired a finite number of times, even though the set of possible traces is infinite. The projection operation is used for data hiding, when composing a number of modules. The weave operation introduces concurrent behaviour, but can often make a specification much harder to read. Some of the basic modules are shown in Figure 2.21.

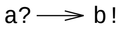
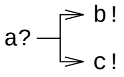
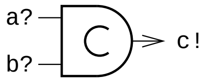
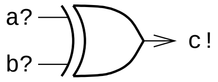
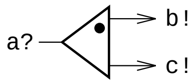
Component	Circuit	Symbol	Trace expression
Wire	Wire		pref∗[a?;b!]
Fork	Fork		pref∗[a?;(b!‖c!)]
Join	C-element		pref∗[(a?‖b?);c!]
Merge	Xor		pref∗[(a?|b?);c!]
Toggle	Toggle		pref∗[a?;b!;a?;c!]

Figure 2.21: A few examples of trace theory circuit primitives

Operator	Written as	Meaning
Assignment	$x\uparrow$, $x\downarrow$	x set to logic 0, 1 resp.
Sequencing	$\langle cmd\rangle;\langle cmd\rangle$ $\langle cmd\rangle,\langle cmd\rangle$	Commands in sequence Commands in parallel
Selection	$[G_1 \rightarrow \langle cmd_1\rangle [\!]\ldots [\!] G_n \rightarrow \langle cmd_n\rangle]$	Wait for a G_i to be true, execute $\langle cmd_i\rangle$
Repetition	$*\langle selection\rangle$	Repeat $\langle selection\rangle$; if all G_i false, exit
Interconnection	channel($\langle p\rangle,\langle q\rangle$)	Create channel between ports $\langle p\rangle$ and $\langle q\rangle$.
Synchronization	Process 1: $\langle p\rangle$ Process 2: $\langle q\rangle$	Each process waits for the other, then proceeds
Communication	Process 1: $\langle p\rangle!\langle x\rangle$ Process 2: $\langle q\rangle?\langle y\rangle$	Synchronization, then $\langle y\rangle{:=}\langle x\rangle$
Probe	$\overline{\langle p\rangle}$	**true** if another process is blocked on port $\langle p\rangle$

Figure 2.22: Operations in Martin's CHP

2.4.2 Martin's CHP

Communicating Hardware Processes (CHP) [114, 115] is based on a subset of CSP [73] with the additions of Dijkstra's guarded commands [51]. A CHP specification has a number of *processes*, operating in parallel, connected by *channels*, over which all synchronization and data transfer takes place. Compared to CSP, only the boolean data types are supported, and there is no dynamic creation of resources. Some additions to CSP are the probe operation, which can tell whether data is ready on a channel without blocking, and multiple channels or buses. The CHP operations are listed in Figure 2.22.

Using an automated procedure, a CHP specification is translated step-by-step into production rules for each signal. An example production rule for a C-element

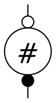
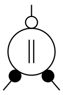
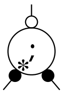
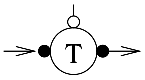
Name	Symbol	Function
Repeater		After handshake on upper port, gives unbounded number of handshakes on lower port
Parallel		After handshake on upper port, complete handshakes simultaneously on both lower ports and then acknowledge above
Sequencer		After handshake on upper port, complete handshake on starred port, then other port, then acknowledge above
Transferrer		After handshake on upper port, transfer one item of data from left to right, then acknowledge

Figure 2.23: A few examples of Tangram circuit primitives

is $\mathtt{a} \land \mathtt{b} \rightarrow \mathtt{q}\uparrow$, $\neg\mathtt{a} \land \neg\mathtt{b} \rightarrow \mathtt{q}\downarrow$. The procedure often requires expert assistance to achieve the best circuit for a particular use. Martin's approach appears to create the fastest circuits out of all the asynchronous design styles.

Timed Handshaking Expansions, derived from CSP by adding timing information, is a language developed by Myers [136] to interface to an existing timed graph-based specification due to Myers and Meng [137]. Alterations were required so that timing makes sense, for example, guards were changed to events rather than levels. The graph-based specification was then translated to a timed state graph before synthesis. Circuits that are 50% smaller and 50% faster than those produced by SIS were reported.

2.4.3 Tangram

Tangram was developed by van Berkel at Philips [10], to address some of the problems with other specification languages. Ebergen's language was too low level, and the circuits produced were composed from a small number of modules, which meant many instantiations of these modules were required resulting in a large circuit. Martin's synthesis style, although giving good performance, was not suitable for the standard cell work that Philips were doing. The solution adopted was to have a combined CSP and guarded command language like Martin's, but to translate to circuit primitives that are closely connected with the language constructs. The translation was therefore "highly transparent", so a good idea of circuit complexity can be gained from looking at the specification.

A few circuit primitives are given in Figure 2.23; there are too many to list here. Lines connecting two modules are actually a request/acknowledge handshake pair, with the request going from the solid circle to the open one, and the acknowledge the other way. Either two-phase or four-phase implementations of modules are possible.

Tangram's focus on handshaking and composability can lead to inefficient circuits, but its big advantage is low power as was demonstrated by an impressive reimplementation of the Philips DCC error correction circuitry [11]. To increase performance, some peephole optimizations were proposed by Kessels [86] that can remove useless circuit elements inside modules.

2.4.4 Others

Brunvand-style compilation to Macromodules

The precursor to Tangram was a language used by Brunvand and Sproull [19] to specify the interconnection of Macromodules, themselves designed by I-net synthesis. The language was a subset of Occam, itself based on CSP. Macromodules were designed for each of the language constructs, and then a syntax-directed translation carried out from the high-level language.

Resynthesis using Petrify

A text-based specification was used as an interface to a graph-based synthesis algorithm by Peña and Cortadella [146]. Circuits were built out of a set of modules, but these modules were actually defined as Petri nets. The composition of these nets could then be optimized and resynthesized using `petrify`. When a number of circuits were synthesized in this way, and the area of the resulting circuit compared to the area that would result if the modules were implemented separately, a 30% drop was observed; however, 60% of the improvement came from a single example, and many circuits were not smaller at all. Kolks et al. [91] have also tried this approach with ASSASSIN.

Event Controlled Systems

Event Controlled Systems (ECS) [130] are a way to integrate two-phase control signals with four-phase data signals. Two-phase signals are translated from the *voltage domain* into the *temporal domain*, where they become essentially four-phase signals, then module behaviours are specified in terms of these four-phase pseudo-signals. Bounded delay models are used, with bundled data. ECS has been used to design a microprocessor, ECSTAC [131], but it has not caught on widely, perhaps because of the counterintuitive translation between domains.

Synchronized transitions

Synchronized transitions, developed by Staunstrup, have been used as a high-level specification for modular verification of speed-independence [90]. The behaviour of a circuit can be specified, then semi-modularity and hence speed-independence mechanically checked. An example specification in synchronized transitions for a C-element is $\langle\langle \mathrm{a}{=}\mathrm{b} \rightarrow \mathrm{c}{:=}\mathrm{a} \rangle\rangle$, which means that whenever the two inputs a and b are equal, the output c should be assigned to the value of one of them. Synchronized transitions have also been used as an interface to an ML-like functional programming language FL [106], again for property verification.

2.5 Concurrency Reduction

Concurrency reduction has been done previously at the STG level, by Vanbekbergen et al. [183], and at the SG level, by Ykman-Couvreur et al. [197] and later Cortadella et al. [39]. Vanbekbergen used concurrency reduction to solve the USC problem, which is a sufficient but not necessary condition for STG synthesis. By adding arcs to the STG, the number of state variables required can be reduced, possibly to zero. However, this method was limited to live-safe marked graphs with single transitions of any signal.

Concurrency reduction was used by Ykman-Couvreur et al. [197] to reduce the number of state variables needed to implement a given state graph. The concurrency reduction operation was to pick a pair of transitions $\mathtt{t}_0$ and $\mathtt{t}_1$, then to ensure that these two transitions must alternate $\mathtt{t}_0, \mathtt{t}_1, \mathtt{t}_0, \mathtt{t}_1$ in the state graph by deleting all states that do not fit this pattern. All pairs of states were examined, and a pair of cost functions attached to each: cost 1 was the number of states in the state graph that were deleted, and cost 2 was the number of CSC violations left in the state graph. The pair with the lowest cost 1 was chosen, using the lowest value of cost 2 to break ties. This method tried to reduce concurrency as little as possible while maximally reducing the number of states with CSC violations. Fewer CSC violations typically means less state variables will be required, and a smaller circuit will result. State variables were added using the work of Vanbekbergen et al. [182], if any were needed. Results were promising in terms of area, but speed measures such as cycle time were not given.

The recent work of Cortadella et al. [39] introduces two new concurrency reducing operations. The first is an extension of the work of Ykman-Couvreur, including some new correctness constraints. The second is more general, but was not implemented because it was difficult to see what the operation actually represented at a Petri net or STG level.

2.6 FSM synthesis algorithms

Synthesis of finite state machines will play a central role in this dissertation. In this section relevant algorithms are collected together, along with other related ideas.

2.6.1 ISSM minimization

The starting point for finite state machine synthesis is typically a flow table representing an incompletely specified sequential machine, or ISSM. An example flow table for an ISSM from Miller [124] is shown in Table 2.1; it is a synchronous machine, but the same principles apply to asynchronous machines.

Inputs to the circuit are labelled A–D horizontally, and internal states 1–5 vertically. An entry of *a*/*b* in column *i* and row *s* means that in state *s*, if the input given is *i*, the machine should output symbol *b* and move into state *a* on the next clock edge. Dashes represent don't-cares and are used to fill in entries that will not be

	A	B	C	D
1	2/0	-/1	3/-	2/0
2	3/0	5/1	2/0	-
3	3/0	4/1	-	5/0
4	-	1/1	2/-	-
5	-	-	1/1	-

Table 2.1: Flow table example from Miller [124] and Unger [177]

used; after moving to a don't-care state, all outputs of the machine are don't-cares from then on.

If this machine was implemented as it stands, the five states would require $\lceil \log_2 5 \rceil = 3$ binary state variables to encode the present state. State machine minimization attempts to combine the rows of the table into a smaller number of new rows, not necessarily disjoint, such that the behaviour of the machine is preserved. In Table 2.1, an attempt could be made to create a new table with just two rows $a = \{1, 2\}$ and $b = \{3, 4, 5\}$, but then the next-state entry for state a under input A would be $\{2, 3\}$, which corresponds to neither a nor b. The reduced table cannot be filled in to give the behaviour of the original, so that minimization is incorrect.

Minimization of flow tables usually, but not always, gives a smaller circuit. A *minimal* solution is one that has the smallest number of rows in the reduced machine. Minimization of a completely specified machine with n rows is known to take time $O(n \log n)$, but the problem for incompletely specified machines has been proven to be NP-complete [60].

Early state minimization algorithms were described by Unger [177]. A *compatible pair* is two states that, when given the same sequence of inputs, will produce output sequences that agree with each other where they are defined. Don't-care values in the output sequences are taken to agree with whatever the other sequence has in that place. For example in Table 2.1, states 1 and 2 form a compatible pair, which can be shown by applying any input combination; e.g. ABCD gives an output of 0110 from state 1 and 01–– from state 2, which agree with each other. States 1 and 3 do not form a compatible pair, because when given the input DC, they give outputs 00 and 01 respectively.

Compatibles are sets of states such that any two members are a compatible pair, and *maximal compatibles* or *MCs* are compatibles that cannot be made larger by adding another member. In Table 2.1 the maximal compatibles are $\{1, 2\}$, $\{1, 4\}$, $\{2, 3\}$ and $\{3, 4, 5\}$. The set of all maximal compatibles is always a solution to the minimization problem, although this often produces more rows than the original table had. Here, a new flow table with four rows can be created, with each row corresponding to one of the MCs, as shown in Table 2.2. Because some states of the original table belong to more than one state of the reduced table, some next-state entries have a choice between two states.

Usually, a reduced machine can be found using a proper subset of the maximal

New state	A	B	C	D
a = $\{1, 2\}$	c/0	d/1	c/0	ac/0
b = $\{1, 4\}$	ac/0	ab/1	c/-	ac/0
c = $\{2, 3\}$	cd/0	d/1	ac/0	d/0
d = $\{3, 4, 5\}$	cd/0	b/1	a/1	d/0

Table 2.2: Flow table reduced using maximal compatibles

prime	Φ(prime)	Notes
{1,2}	{2,3},{4,5}	Max compatible
{1,4}	{1,2},{2,3},{4,5}	Max compatible
{2,3}	{1,2},{4,5}	Max compatible
{3,4,5}	{1,2},{2,3},{1,4}	Max compatible
{3,5}	none	
{4,5}	{1,2},{2,3}	
{3,4}	{1,2},{2,3},{1,4},{4,5}	Deleted by {3,4,5}

Table 2.3: Primes from Table 2.1

compatibles. This is not true in the example given, as can be seen by noting that in Table 2.2 every state appears at least once on its own as a next-state entry, so no row can be omitted. When minimizing completely-specified tables, it can be proven that there is a minimal solution using only maximal compatibles, but this result does not hold for incompletely specified tables. More sophisticated methods are required.

Prime classes were introduced by Grasselli and Luccio as a replacement for maximal compatibles. They proved that at least one minimal solution of ISSM minimization is composed of prime classes only [64], and later gave algorithms to solve the problem [65].

A compatible P is said to *imply* another compatible Q if including P as a state in the reduced machine means that there must be another state in the reduced machine that contains Q. In Table 2.1, having a row $\{1, 2\}$ in the reduced machine means that $\{2, 3\}$ has to be included as well by considering the next-state entry of the new state $\{1, 2\}$ under the input A, so $\{1, 2\}$ implies $\{2, 3\}$. The set $\{2, 3\}$ itself implies other sets. Let the transitive closure of the implies set for set S be denoted $\Phi(S)$. Then the *prime classes* are all compatible sets S such that there is no set T with $S \subset T$ and $\Phi(T) \subseteq \Phi(S)$. Intuitively, prime classes are all sets that might be useful in a reduced machine; if there is a set T that includes S but also implies less than S, then S can be replaced by T in any reduced machine, so there is no point considering S. Suitable candidates for prime classes are the maximal compatibles, and subsets of them. An improved method to find prime classes was given by Bennetts [8]. The prime classes of Table 2.1 are given in Table 2.3.

When picking prime classes to include in a solution, every row of the original table must be included in some row of the reduced table, and every set implied by a set in the solution must also be in the solution, called *covering* and *closure*

New state	A	B	C	D
a = $\{1, 2\}$	b/0	c/1	b/0	ab/0
b = $\{2, 3\}$	b/0	c/1	ab/0	c/0
c = $\{4, 5\}$	-	a/1	a/1	-

Table 2.4: Flow table reduced using prime classes

constraints respectively. It is easy to see from the Table 2.3 that $\{1, 2\}$, $\{2, 3\}$ and $\{4, 5\}$ can all be chosen and only imply each other, so satisfy the closure constraint. They also obviously cover the original table, so they give a minimized table with three rows, shown in Table 2.4. It can also be seen that no other three-member solution exists formed from prime classes.

DeSarker [49] proposed a faster solution by splitting the problem into two steps: first, recursively combining prime classes with their implied classes to create *prime closed sets*, and secondly using these to cover the original table.

One of the best-known tools for state minimization is STAMINA, described by Rho et al. [153]. This combines two previously known algorithms with two new heuristic ones. The first new algorithm is good at machines with a particular structure, so may not be much use for general machines. The second algorithm tries to reduce the number of prime classes that need to be considered by suboptimally solving the problem using only maximal compatibles. Prime classes that are contained in maximal compatibles that appeared in the solution are likely to be the most useful in solving the problem, so only these prime classes are added to the set of maximal compatibles and the problem solved again. Although this is not guaranteed to give a minimal solution, it did well on all examples that were tried.

A novel approach was described by Puri and Gu [148]. The search for a solution is depicted as a tree, and then pruning criteria are used to reduce the search space. Heuristics are used to concentrate on areas of the tree that are likely to give the best solutions. It was reported to be a little faster than STAMINA, but Puri and Gu's algorithm will always give a minimal solution. The algorithm will be described in more detail later in the dissertation.

2.6.2 State assignment

State assignment is the action of allocating one or more binary codes to each row of a flow table. These binary codes will typically be held in a flip-flop of some kind, and then next-state logic equations derived using a logic synthesis algorithm. A *unicode* assignment is one where every row has a single binary code; a *multicode* assignment has one or more rows with several binary codes. A *single transition time* assignment is one where, on a state change, any state variables that must change do so concurrently. Non-STT assignments are called *multistep* assignments. The abbreviations USTT and MSTT for unicode and multicode single transition time assignments are often seen. The best USTT assignment for an n-row machine has the number of state variables s proportional to $\lceil \log_2 n \rceil^2$; this is higher than the MSTT assignment given in Kuhl [99] which has $s \leq 2\lceil \log_2 n \rceil$, but multicode assignments tend to have more logic per state variable.

State assignment for synchronous circuits does not affect correctness, only the size of the circuit. State changes always happen on clock edges, so any number of state variables can change simultaneously. Without a global clock, if several state variables attempt to change at the same time, a *race* will occur; the changes occur in some order, so the machine passes through several intermediate states. If one of these intermediate states is also part of another transition to a different final state, then the machine cannot tell which final state is the correct one. This is called a *critical race*, and should be avoided.

One of the earliest asynchronous state assignment algorithms was given by Tracey in 1966 [176], based upon earlier work by Liu [110]. Tracey's algorithm looks at all situations where a critical race might occur, and makes sure that no intermediate state can ever occur as part of two transitions to different states. When it was proposed, it was excessively time-consuming, and two other approximations of the algorithm were also given by Tracey [176] to reduce computation time. An even more approximate and fast version was given by Smith [168], which keeps the problem small by partially solving it whenever a certain size is reached. Computers are now around a million times faster than in 1966, so Tracey's original algorithm is feasible. Tracey's algorithm also can be used to add state variables to an existing assignment, rather than creating a whole new state assignment. Unger [178] gave an extension to Tracey's algorithm, which allowed certain combinations of inputs to change concurrently, even though the fundamental mode assumption forbids this. His algorithm will be described later.

Tan's algorithm [173] gives small implementations on PLAs by arranging that the next-state equations for different variables have many product terms in common. The assignments, which have more state variables than Tracey assignments, do not appear to be smaller on CMOS. Another PLA-targeted algorithm was given by Rutten [158], a development of an earlier algorithm by Fuhrer. Using a particular measure of PLA size, their algorithm gave a 3% size advantage over Tracey's algorithm, although over three-quarters of that improvement came from a single example, and out of the given examples, four were better with Tracey's algorithm and four were worse.

Multistep algorithms tend to produce circuits that are small but slow, because each state change can take several variable transitions. Maki and Tracey [113] gave an algorithm that starts with an assignment with the minimum number of state variables, then adds in new variables until all transitions can be completed. The algorithm attempts to minimise the number of steps needed for each transition, so the circuit is not too slow. Fisher and Wu [56] gave an algorithm that embeds a graphical representation of the flow table into a hypercube with vertices corresponding to the 2^n states of the n state variables. No comparison was made with the earlier work by Maki and Tracey, and it is not possible to add to an existing assignment.

Multicode assignments can be created that have only one state variable changing between rows in the flow table, as done by Kantabutra and Andreou [83]. The advantage of this is low power; they found their circuits took 58% of the power of a

Tracey assignment, although the state logic was 23% larger. The size and power of the output logic was not considered, and no indications of speed were given.

2.6.3 Logic synthesis

Several ways exist of implementing circuits in CMOS. The trade-off that has to be made is ease of design and low cost against performance, such as high speed and low power. Highest speed is achieved by using complex gates and full-custom design, but this is time-consuming and error-prone, and costly to fabricate. Standard cell synthesis makes full-custom design easier and faster, but chip production is just as expensive and the circuits produced are not quite as fast. Sea-of-gates design allows complex gates at relatively low cost, but has fallen out of favour. PLAs and FPGAs are low-cost approaches which give noticeably inferior results, but are good for rapid prototyping and proof-of-concept.

Complex gates in full custom CMOS

Two forms of CMOS complex gates are commonly found in the literature:

- Pull-up/pull-down trees with a weak keeper inverter, also called a feedback inverter or staticizer, such as those used by Martin [118] and the Amulet group [59]. These are usually specified as conditions on the rising and falling transitions of the output. I will call these *dynamic gates with keepers*, although they are sometimes known as dynamic or pseudo-static gates.

- Fully static gates that are specified by an on-set and an off-set, where there are no states of the input that can cause the output to float. These are used by *petrify*, MEAT and the later Amulet work [108]. I will refer to these as *fully static* gates.

A complex function with several inputs will typically be smaller and faster when built as a dynamic gate with a keeper, but small functions such as 2-input C-elements are faster as the fully static form [46, 108].

Figure 2.24 shows a standard CMOS And-Or-Invert or AOI complex gate, where the P-tree has been implemented as the dual of the N-tree. If $b = c = 1$ and a changes from 1 to 0, a static hazard can occur on the output. During the time that a and $\overline{a}$ are 0, the P tree erroneously conducts, because there is a conducting path through the series a and $\overline{a}$ transistors. It can be seen that there will be a problem by multiplying out the expression from the P-tree into $a.\overline{a} + a.\overline{b} + \overline{a}.\overline{c} + \overline{b}.\overline{c}$; the $a.\overline{a}$ term is obviously to blame.

If the P-tree of a gate was also implemented as a Sum-of-Products (SOP) expression as well as the N-tree, then there would never be any product terms including a signal and its complement. This is shown in Figure 2.25; it can be seen that instead of producing a hazard, the P-tree simply lets the output of the gate float, which presents no problems in CMOS. Such gates are termed *SOP/SOP* gates because they have sum-of-product expressions for both the N tree and P tree, and were discovered independently by several groups.

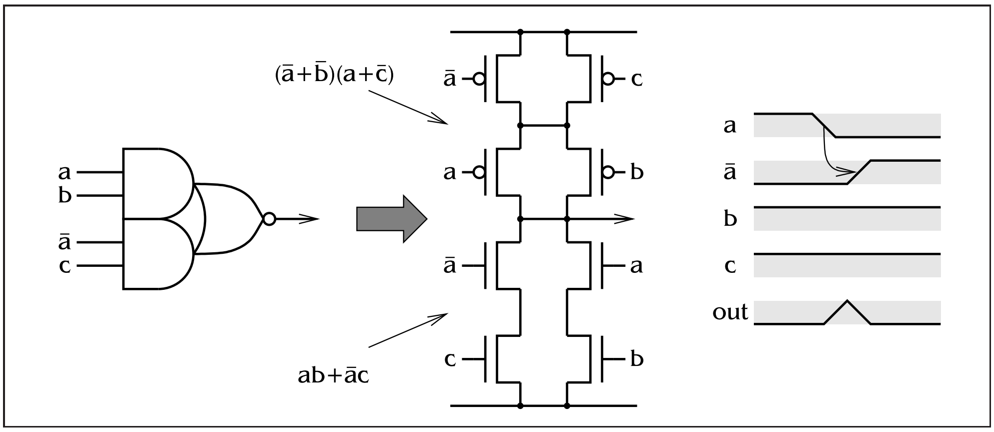


Figure 2.24: A gate with a single-input-change static hazard

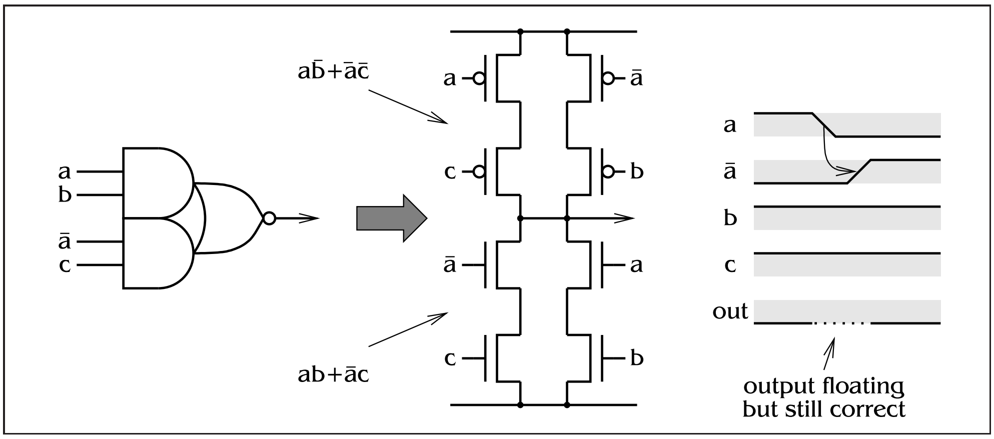


Figure 2.25: Gate with single-input-change hazard removed

There is some debate as to whether complex gates are smaller than standard cell synthesis. The results of Kudva et al. [98] showed that complex gates were well under half the size of library solutions, and typically had only half the delay. Beerel [5] believes that there are more opportunities for logic sharing with simple gates from a library, so that complex gates may well be larger. The truth is probably somewhere in the middle.

CMOS technology does not look to be infinitely shrinkable, so there will come a time soon when process optimizations will run out of steam. At that time, complex gate solutions may have a hard time migrating to other technologies. When Martin ported his asynchronous processor to GaAs, the complex gates required mapping to a standard cell library, because P transistors are particularly poor conductors on GaAs. Recently, Brown et al. [18] described a technology called *Complementary GaAs* or *CGaAs*, where P transistors have gains within a small factor of the N transistors and gates are semi-isolated, so CMOS complex gates can be directly ported. CGaAs processing currently costs about five times as much as silicon, but may give complex gates a new lease of life when the CMOS end-point is reached.

Standard cell synthesis

Standard cell synthesis in asynchronous circuits is difficult because most logic functions require at least two levels of logic, which introduces delays and may cause hazards. The main problem for standard cell synthesis algorithms is how to accomplish *technology mapping*, the translation of Boolean functions to a simple library of gates, without introducing hazards. When tech mapping speed-independent circuits, it is required that the decomposition is also speed-independent, which may not always be possible. For a limited class of SI circuits, Varshavsky et al. [184] proved that a decomposition using 2-input NANDs exists, but this result does not hold for all SI circuits. Tech mapping will be avoided in this dissertation; the interested reader will find recent material in [4, 41, 93].

When synthesizing two-level hazard-free logic, Espresso-HF can be used [174]. This is a development of the popular tool Espresso-II [15], which targets PLA design but can also be used for two-level logic or complex gates. A different approach to logic synthesis was presented by Lin and Devadas [109], where a binary decision diagram representing a logic function is translated into a tree of multiplexers. This ensures freedom from most hazards if the multiplexers are static-hazard-free, as they are on CMOS. These circuits look to be large, even with the given optimizations.

FPGAs

Field programmable gate arrays have traditionally been targeted at synchronous circuits, with very little provision for asynchronous elements; arbiters present a particular problem, because the required analogue circuitry is not present. Hauck et al. [71] designed an FPGA with arbitration blocks that can be used for mixed synchronous/asynchronous work, and gave mappings from common DI circuit elements onto the FPGA structures. Moore [129] gave an arbiter design on standard synchronous FPGAs which appears to be remarkably resistant to metastability. These approaches mean that FPGAs can be used for prototyping asynchronous circuits, even though FPGAs are relatively slow.

Transistor-level optimization

Several things can be done to improve custom and standard-cell designs, but not FPGA-based or multiplexer circuits. Changing the ordering of transistor stacks can have speed and power benefits. Carlson and Lee [23] showed that swapping the inputs of 2-input NAND gates can improve the speed of circuits, but simulation of the whole circuit was the only way to determine which ordering was best. Power consumption was optimized by Panwar and Rennels [142], by reordering transistors to reduce the power dissipated inside a gate as a result of charging and discharging internal nodes. Transistor sizing using Logical Effort was described by Sproull and Sutherland [169, vol. II], which can be used as a good first-approximation to a detailed simulation-based approach. Transistor-level optimization will not be considered in this dissertation, although the algorithms that are used should allow these sort of optimizations to be added at a later date.

Initialisation

Initialization will not be explicitly considered in this dissertation. Two possible methods of initializing circuits are to build everything out of flip-flops with resets, such as the approach used by ASSASSIN [199], or to use the method suggested by Coates et al. [34], which is to turn an inverter after a complex gate into a NAND or NOR gate as appropriate. The method used for initialization depends on the synthesis style, and it is unlikely to either be a difficult problem or to perturb the results by very much.

2.7 Summary

Asynchronous circuit design is a very diverse field, with a variety of complementary methods. A number of different delay models have been used to design circuits, each with its own advantages and disadvantages, but there is no clear winner. Two-phase and four-phase signalling again each have their own strengths, but in the end it comes down to personal preference which one is used.

A major division in the subject is whether to use a graph-based specification, or to use a programming language to design circuits. Graph-based approaches are clear and readily understandable, but are typically only used to build subcircuits which will later be composed to create a system. Text-based approaches can specify an entire system, which can be decomposed into smaller and smaller pieces until these pieces can be implemented directly in hardware, but these methods have their own problems: Tangram and Ebergen's method give large and slow circuits, and Martin's method often requires expert assistance.

Finally, there is little consensus on whether complex gates or library synthesis is best, but this decision is affected more by outside factors, such as what the rest of the institution uses for their work.

I believe that asynchronous circuits are in a reconnaissance phase. Synthesis approaches are being scouted out and tested against other styles; given time, it will become clear which methods give the best results, and then the field will mature as effort is concentrated at the most promising techniques. This dissertation attempts to illuminate a little more of the research landscape, specifically in the direction of a modified Petri net specification, combined with concurrency reduction and a link back to the early work of Unger and Tracey.

Overview and Motivations 3

Abstract

This chapter, which is based on my thesis proposal, gives the reasons why the work in this dissertation was attempted. Furber and Day [59] designed a number of latch controllers essentially by hand, using ad-hoc concurrency reducing transformations. It was found that the use of a concurrency reduction operation on an intermediate form called a *blue diagram* could give the same results without manual intervention, which suggests that a synthesis tool based upon blue diagrams could replicate and generalize the earlier latch controller work. This chapter gives a brief and simplified overview of the proposed synthesis tool, from the point of view of replicating the results of Furber and Day.

3.1 Delay assumption

One of the strengths of asynchronous circuits is modularity. Complete circuits can be built up out of smaller modules; if these modules adhere to well-defined interface constraints, then their composition will function correctly. The steady increase of wire delays in current CMOS processes means that the only reliable assumption that can be made about delays between modules is that they are finite but unbounded, so the connections between modules must be delay insensitive. If a module presents a delay-insensitive interface, then it does not matter how it is implemented internally. Kuwako and Nanya [100] looked at which delay assumption should be used for various circuit parts, and concluded modules should be built using the Quasi-delay-insensitive assumption internally. This and the popular speed-independent assumption are both pessimistic when applied inside modules, often giving circuits that are larger and slower than the more realistic fundamental mode assumption would have given [45]. The delay assumption used in this dissertation is fundamental mode with additional conditions which attempt to cause correct behaviour for concurrent multiple-input changes.

3.2 STGs, Fragments and Snippets

STGs often specify an ordering between outputs of a circuit, even though this ordering cannot ever be seen if the connections between modules are delay insensitive.

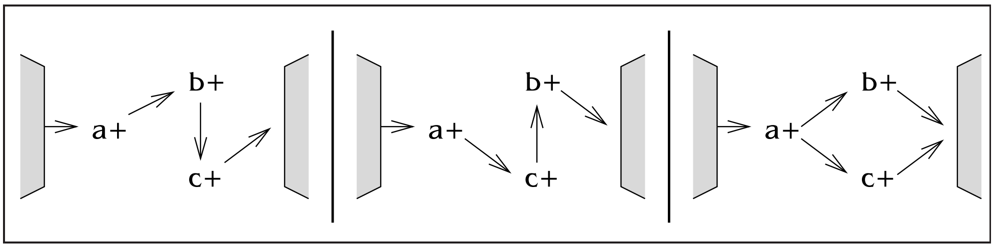


Figure 3.1: Three different STGs for essentially the same behaviour

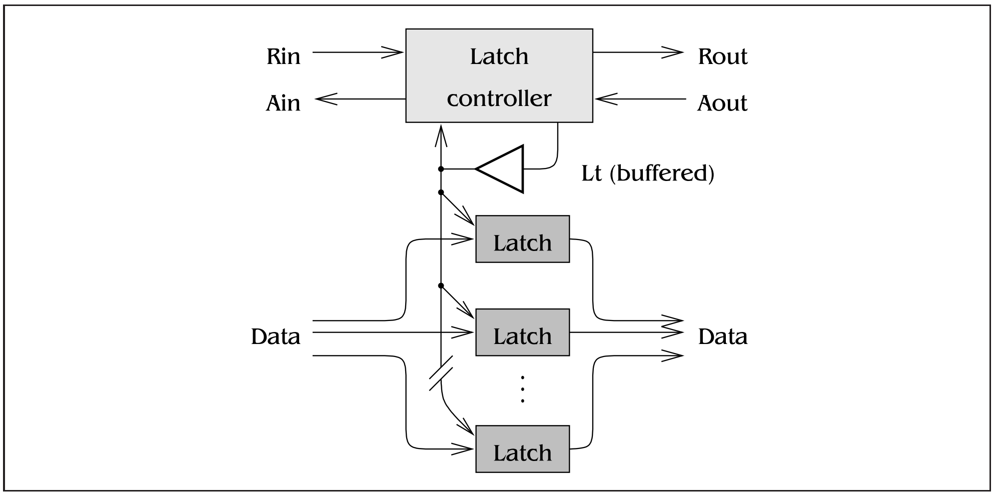


Figure 3.2: Four-phase latch controller

Figure 3.1 shows a small fragment of three larger STGs for a hypothetical circuit module, where a is an input to the circuit and b and c are outputs. When a goes high, both b and c should go high in some order. Under a DI model of module interconnections, the three behaviours in Figure 3.1 are indistinguishable even though their STGs are different. The STG has restricted the implementation of the module as well as giving its specification.

Another problem with STGs is that it is difficult to compare the concurrency of two specifications. Figure 3.2 shows a design problem for a latch controller. Furber and Day [59] specified a solution by the STG fragments given in Figure 3.3. These fragments show:

- An input handshake on `Rin` and `Ain`
- An output handshake on `Rout` and `Aout`
- Data flowing forwards through the latch `Rin+` ⟶ `Rout+`
- Latch set and reset conditions:
  - *Set* (`Lt+`) when data received (`Rin+`), and then acknowledge the previous stage.

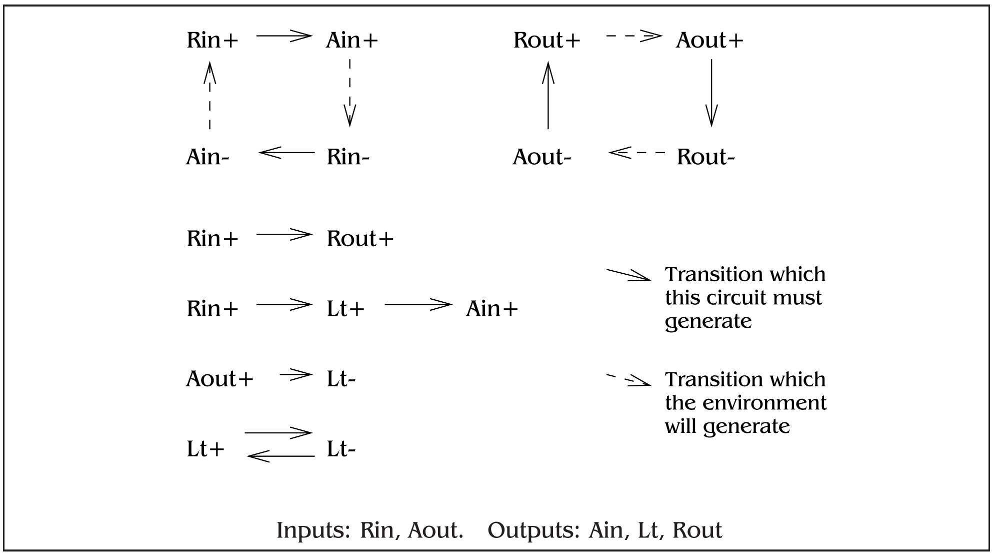

Rin+
Ain+
Ain-
Rin-
Rout+
Aout+
Aout-
Rout-
Rin+
Rout+
Rin+
Lt+
Ain+
Aout+
Lt-
Lt+
Lt-
Transition which this circuit must generate
Transition which the environment will generate
Inputs: Rin, Aout. Outputs: Ain, Lt, Rout

Figure 3.3: STG fragments given in Furber and Day's paper [59]

- *Reset* (Lt-) when the next stage acknowledges it has received the data (Aout+).
- *Alternate* (Lt+ ↔ Lt-) once every input/output handshake cycle.

A similar specification to STG fragments is the concept of *snippets*, which were used by Sutherland et al. [171] to design latch controllers using a four-phase latch and two-phase communication.

STG fragments are a natural specification of four-phase asynchronous circuits. Only important sequencing constraints need to be included in the set of fragments; the precise timing of the return-to-zero handshakes does not need to be specified. A full STG would have to specify orderings between all signals, and also satisfy a number of correctness criteria, such as persistence or complete state coding.

The first part of the synthesis problem from the STG fragments in Figure 3.3 is to find a complete STG that includes all of the fragments. Figure 3.4 (a) shows the simplest merged STG, which Furber and Day called their *simple* controller. The circuit corresponding to this STG consists of a single 2-input C-element. It has the problem that when two or more simple controllers are connected together in a pipeline, and that pipeline stalls, then every other stage will be empty. This can be seen by examining the STG for two simple controllers in series, shown in Figure 3.5: stage $(n + 1)$ must unlatch its data before stage n can latch new data, which is shown by the bold arrows.

Furber and Day's *semi-decoupled* controller, shown in Figure 3.4 (b), was proposed to cure this problem. A new signal A has been introduced to decouple the input and output handshakes, and allow every stage to fill in a stalled pipeline. The circuit corresponding to this STG is shown later in Figure 3.10 (b), and consists of a pair of asymmetric C-elements.

Yun, Beerel and Arceo [201] also designed a semi-decoupled controller, referred

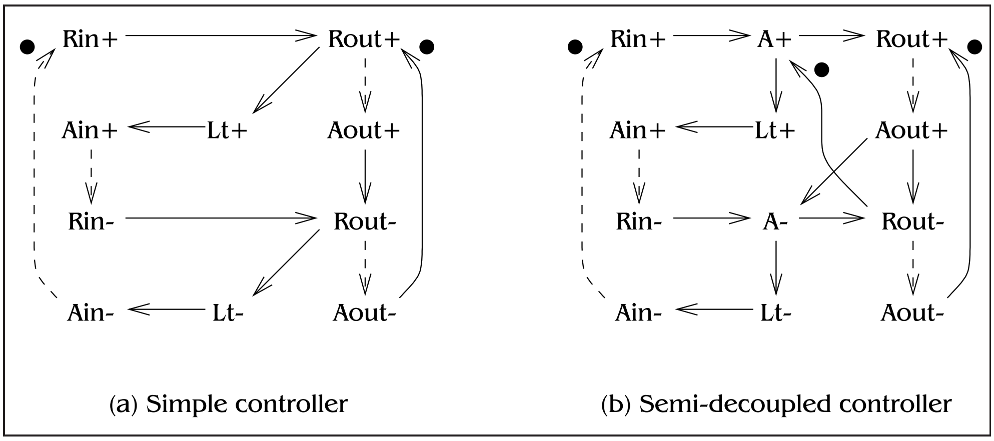


Figure 3.4: Two latch controller STGs from Furber and Day [59]

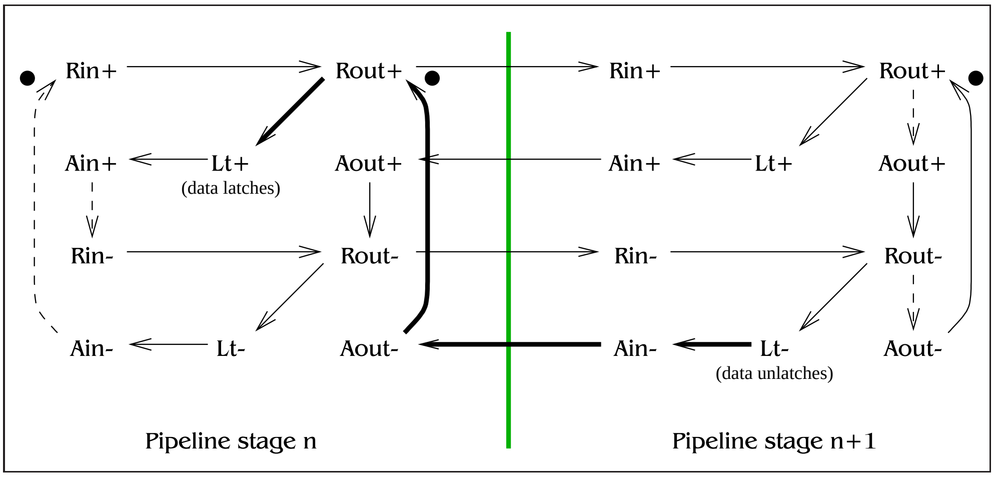


Figure 3.5: STG for two simple latch controllers in a pipeline

to here as the *improved*[1] controller, using a different specification and synthesis style to Furber and Day. An STG of their latch controller, derived from their XBM specification, is shown in Figure 3.6. Unlike the semi-decoupled controller, this has no state variable. State variables add circuitry, so it would be expected that the improved controller would be smaller and probably faster than the original semi-decoupled controller. Unfortunately, the circuits are not directly comparable because they assume different things about the operations of the latch driver buffer. In an apples-to-apples test between the semi-decoupled and improved versions, the improved controller will have a lower cycle time, but the semi-decoupled controller will pass data forward faster. As is often the case, the definition of "faster" is fuzzy, and depends on the circumstances in which a circuit will be used.

[1] "Improved" is their term; maybe "different" would be better.

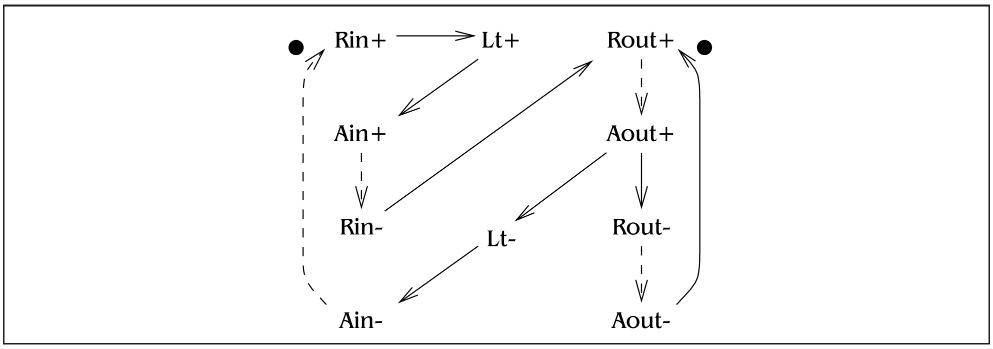


Figure 3.6: STG for an "improved" controller due to Yun, Beerel and Arceo

3.3 Concurrency

It has been pointed out by Ivan Sutherland that the way to achieve speed is by exploiting concurrency. A highly concurrent circuit allows many actions to take place at the same time, but the circuit tends to be larger and therefore slower than a less concurrent one. There is a fine balance between concurrency and complexity, if the aim is maximum speed.

It is difficult to see from the STGs in Figures 3.4 (a), 3.4 (b) and 3.6 what the differences are in concurrency between the specifications. It would be intuitively expected that there are behaviours that the semi-decoupled controller could exhibit that the simple controller could not, but it is not obvious whether the reverse is true, and it is also nontrivial to make comparisons between the improved controller and the other two STGs. A state graph for each specification could be constructed, but these tend to be rather large, and they again suffer from the problem that output transitions are ordered, so identical behaviours when viewed from outside the module may appear different. A question that could be asked of these STGs is "Does there exist an STG that has complexity between that of the simple and semi-decoupled STGs?" An STG fitting this description might give rise to a circuit which is simpler and faster than the semi-decoupled controller, without the problem of only filling every other stage on a stall.

3.4 Blue Diagrams

Blue diagrams, or BDs, are an intermediate representation of module behaviour that can be viewed as a cross between state graphs and burst-mode diagrams; two examples are shown in Figures 3.7 and 3.8. Inputs are ringed in a blue diagram, and outputs are written above the inputs. When moving from one state to another along an arrow, exactly one input must change, but any number of outputs can be different. BDs are state graphs in which the output changes are considered to be instantaneous. They are intended to represent the behaviour of modules that compose in a delay-insensitive way, so the output changes can only be seen by other modules after an unbounded but finite delay. It is assumed that the universe

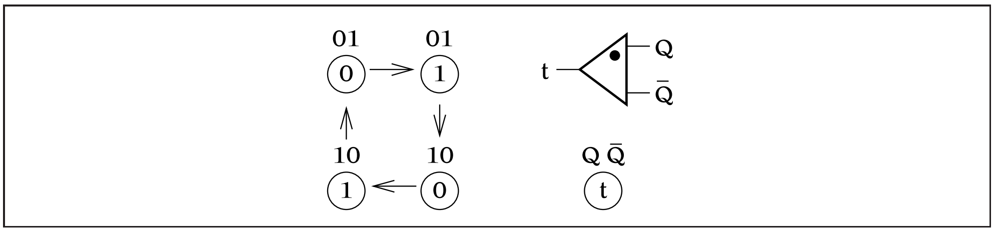


Figure 3.7: Blue diagram for toggle element

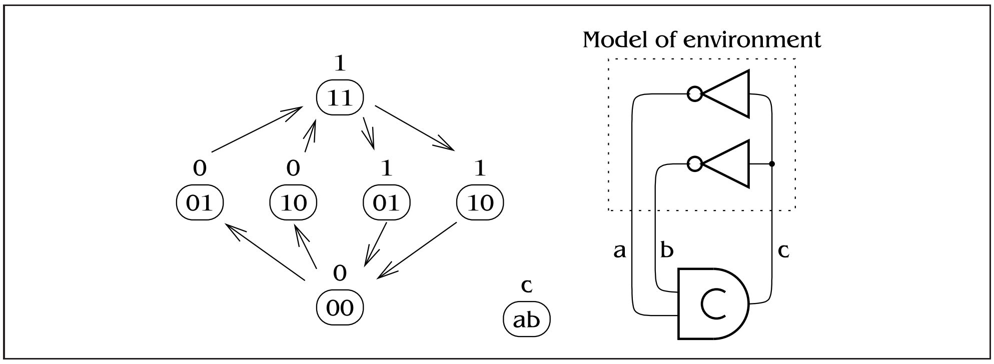


Figure 3.8: BD for C-element with usual environment

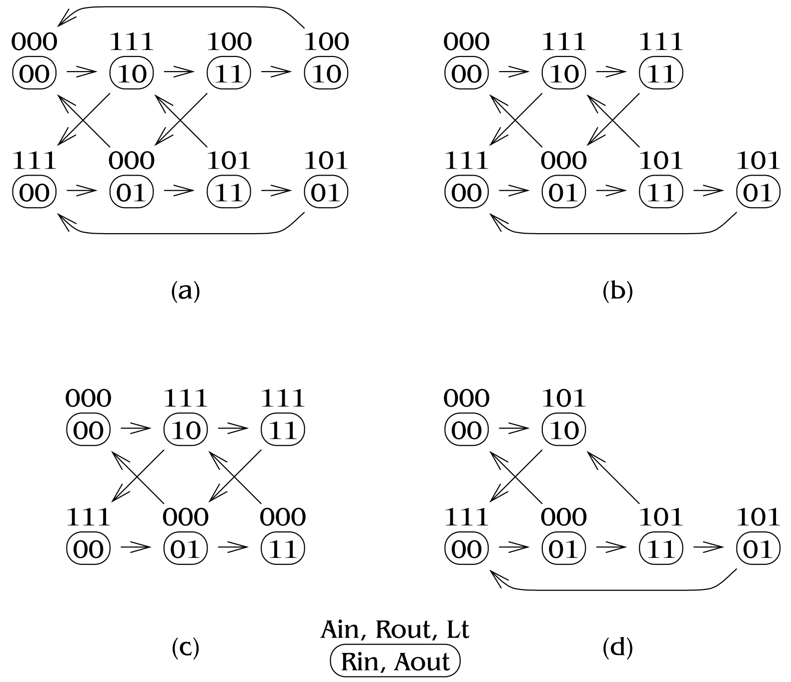
consists of modules specified by BDs, so it is not necessary to add delays on all input wires; these delays are already present, considered to be part of the output of another module. A circuit corresponding to a BD can be viewed as an instantaneous decision-making element, followed by a set of arbitrary delays.

Figure 3.9 shows the BDs of the three latch controllers discussed, and also a fourth BD derived directly from the STG fragments used to specify the latch controller[2]. It can be easily seen from the diagrams that the simple controller and the improved controller both have a restricted set of behaviours compared to the semi-decoupled controller, although they are restricted in different ways. It can also be seen that there is no specification that is between the simple and semi-decoupled controllers in terms of concurrency, answering the question in section 3.3. Comparing diagrams (a) and (b) shows that there is still a loss of concurrency in the semi-decoupled STG compared to the fragments, something that is hard to see just by considering the STGs.

It can be seen from Figure 3.9 that when a state is removed from a BD, some changes must occur in the outputs of previous states. This will be explored in Chapter 5.

Blue diagrams, being essentially little more than compacted state graphs, are not a very useful specification for circuits, but the observations above suggest that they may be useful as an intermediate representation to explore concurrency reduc-

[2] An additional restriction was added that transitions on `Ain` should directly follow `Lt`. This restriction simplifies the diagrams so that the point of this example is clearer, and is due to the original fragments not being quite precise enough. An alternative addition is given in Section 3.5, which leads to Furber and Day's fully decoupled controller.



(a) BD derived from STG fragments, (b) BD of Furber and Day's semi-decoupled latch controller and (c) their simple controller, and (d) the improved controller of Yun, Beerel and Arceo.

Figure 3.9: Blue diagrams of some latch controllers

tion transformations. BDs are close enough to finite state machines that established methods can be used for synthesis. When the BD of Figure 3.9 (b) is converted into a flow table, and then state minimization, race removal and implementation performed using C-elements, the circuit in Figure 3.10 (a) results. This is identical to an earlier controller proposed by Day and Woods [47]. Yakovlev found that this circuit was not speed independent using FORCAGE, which prompted the semi-decoupled controller of Furber and Day [59], shown in Figure 3.10 (b). However, Furber and Day [59] states that Figure 3.10 (a) operates correctly given any reasonable distribution of gate delays regardless of its speed-dependent nature.

If the most concurrent BD, Figure 3.9 (a), is synthesized, the circuit is larger and slower than any of the other circuits. The additional concurrency just increases the size of the circuit and hence its delay. Concurrency reduction on BDs can, when used correctly, make circuits faster, but when a specification becomes overly sequential, any speed benefits will be lost.

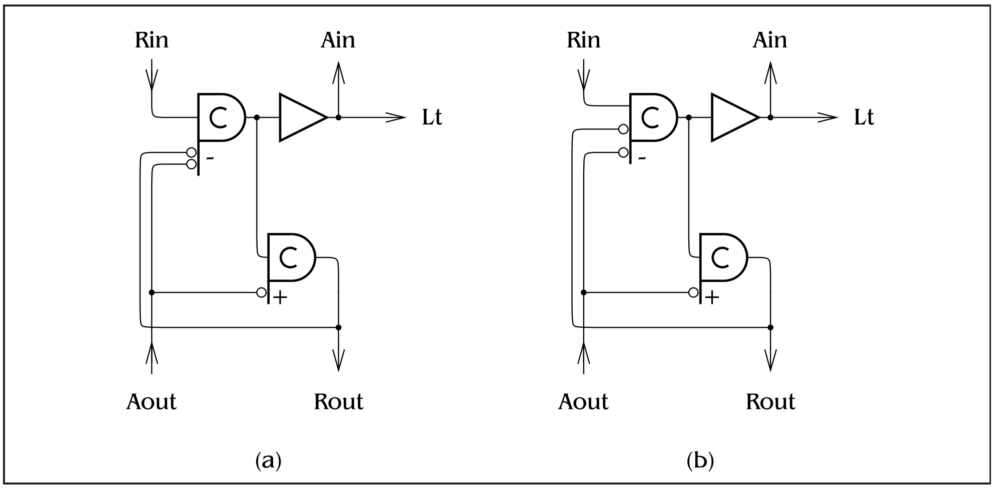


Figure 3.10: (a) Circuit derived by use of blue diagrams, (b) Furber and Day's circuit.

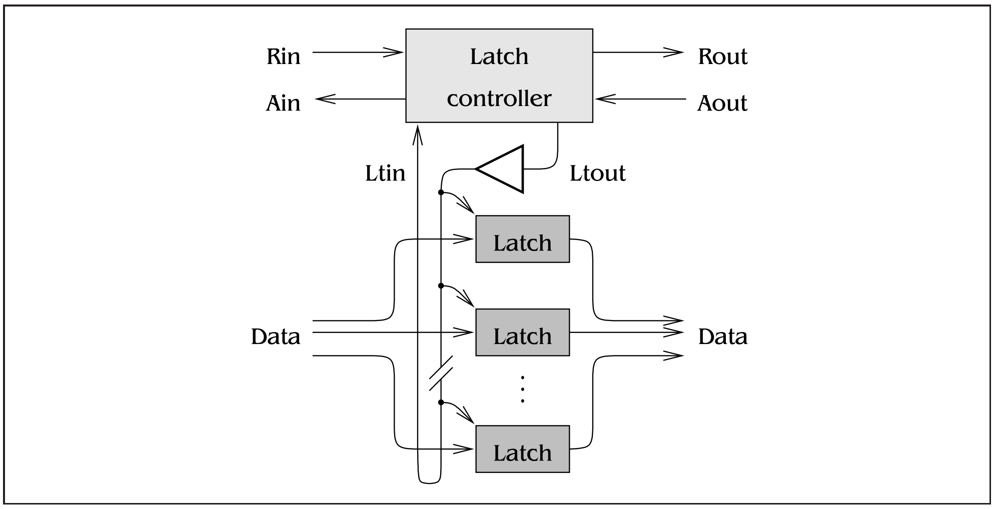
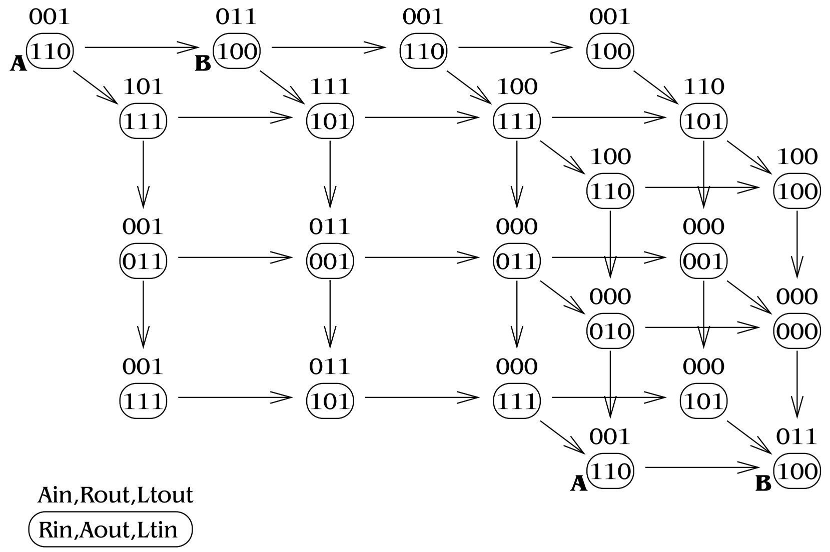


Figure 3.11: Four-phase latch controller, modified to have `Ltin` and `Ltout`

3.5 Fully decoupled controller

The semi-decoupled controller has the property that a processing delay of size t caused by logic between pipeline stages adds a delay $2t$ to the cycle time of the pipeline. Furber and Day also presented a *fully decoupled* latch controller, which has a cycle time that only increases by t in this case. To derive a blue diagram for the fully decoupled controller, it is necessary to consider the latch driver buffer to be outside the control circuit, as it is drawn in Figure 3.11. The latch drive buffer, taking `Ltout` and producing `Ltin`, is considered to be part of the environment.

The STG fragments were modified trivially for the addition of `Ltin` and `Ltout`. The fragment `Ltin-` → `Rout+` was also added, which is required because the latch



The two states labelled **A** are the same, as are the two labelled **B**.

Figure 3.12: Blue diagram derived from modified fragments

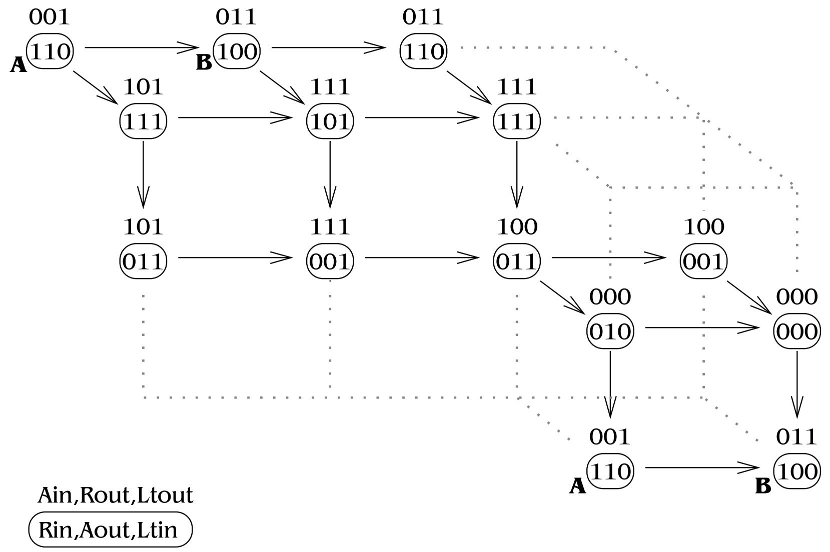


Figure 3.13: Blue diagram for semi-decoupled controller from modified fragments

must be transparent before the receiver can be told that the data is available[3]. The BD derived from the modified fragments is shown in Figure 3.12. The BD for the fully decoupled controller corresponds to deleting the rightmost state on the top

[3] This fragment was not included in Furber and Day's paper, although the ordering it represented was present in all derived STGs.

row of Figure 3.12, so at the outset of this work it seemed likely that a circuit very like the fully decoupled controller could also be produced using BDs. It turns out that this is not the case, although similar circuits can be constructed. Figure 3.13 shows the blue diagram for the semi-decoupled controller using the modified latch driver, showing that it too can be derived from Figure 3.12 by removing states.

3.6 Summary

- STG fragments are an intuitive way to specify the behaviour of circuit modules, even though STGs may specify too much information.
- A Blue diagram derived from the specification displays concurrency in an straightforward way. Concurrency-reducing transformations take the form of state deletions in the blue diagram, with associated patching of output values.
- Highly concurrent specifications can be slow for certain applications; highly sequential specifications will certainly be slow. The fastest circuit for a particular task will usually be somewhere in the middle.
- Blue diagrams can be synthesized using the well-trodden path of FSM synthesis, originally described by Unger.

These points suggest a plausible design methodology for asynchronous circuits. Starting from a specification in terms of STG fragments, a blue diagram could be produced. Deleting various combinations of states from this blue diagram would produce a large number of different diagrams with varying amounts of concurrency. These diagrams could all then be synthesized and scored according to criteria given by the designer, for example any combination of small size, low power and high speed given a particular environment. A synthesis tool based on these ideas would be easy to use and could provide significantly better circuits than current synthesis tools.

Specification 4

Abstract

Blue diagrams cannot reasonably be used as a specification by a designer, so another starting point for the design process must be found. This chapter covers the development of a specification language, and the translation from it into a blue diagram. The translator has been implemented, and some sample results are presented.

Structure of this chapter

Section 4.1 gives some definitions that will be useful. Section 4.2 contains some examples that the specification language must be able to cope with, and some conclusions that can be drawn from these examples. Section 4.3 looks at the creation of the specification language. Section 4.4 covers the translation from the input file to an intermediate Petri net, which is then converted to a pair of blue diagrams with using the material in Section 4.5. An overview of the translation process is given in Figure 4.1. The b2ps program for producing PostScript figures of blue diagrams is briefly described in Section 4.6. Finally, Section 4.7 gives the results of the L2b program for all the examples used.

4.1 Preliminary definitions

Definition 1 *A Blue Diagram or BD is* $(S, p, q, f_{in}, f_{out}, R)$ *with*

$S = \{0, 1, 2, \dots |S| - 1\}$, *the set of states, with 0 as the initial state,*
$p =$ *number of inputs,*
$q =$ *number of outputs,*
$f_{in} : S \to I$ *with* $I = \{0, 1\}^p$, *assigning inputs to states,*
$f_{out} : S \to O$ *with* $O = \{0, 1\}^q$, *assigning outputs to states,*
$R \subseteq S \times S$ *with the property that*
$(a, b) \in R \Rightarrow d(f_{in}(a), f_{in}(b)) = 1$
where d is the usual Hamming metric on I

As a notational aid, if $(a, b) \in R$, I will write $a \to b$. I will also write $a_{in} \equiv f_{in}(a)$ and $a_{out} \equiv f_{out}(a)$. An example blue diagram with its usual graphical representation is shown in Figure 4.2. State numbers are typically unimportant, and they are

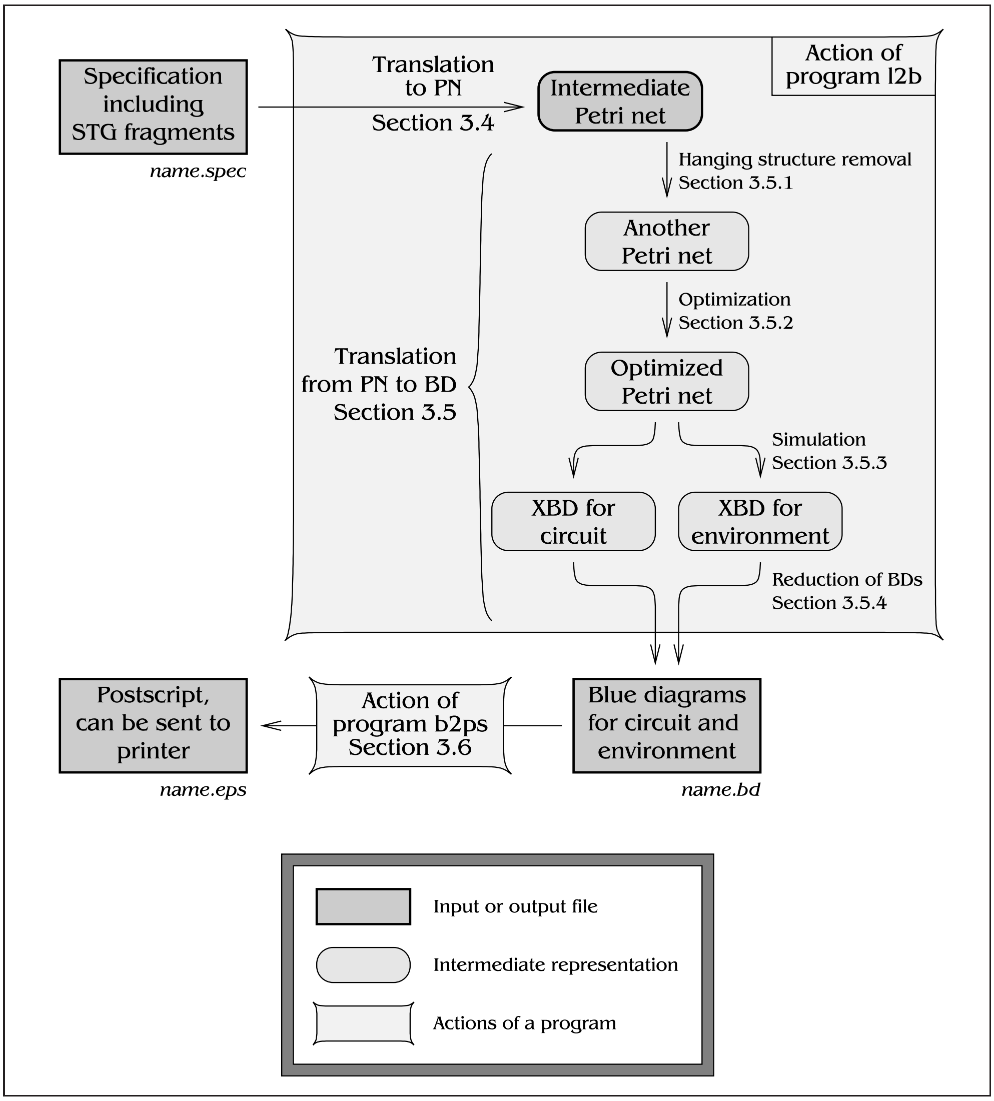
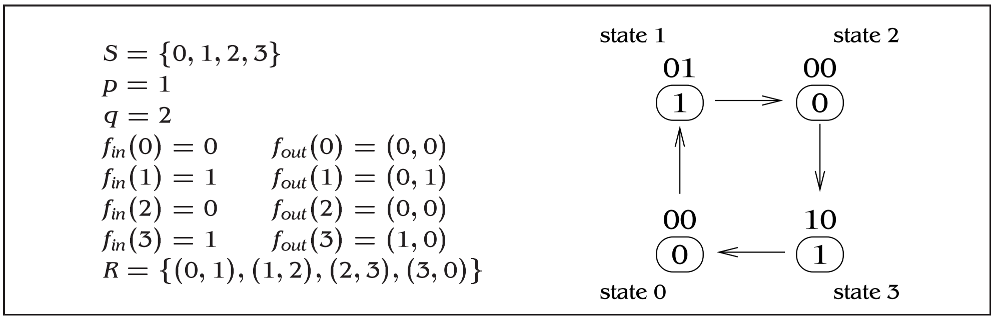


Figure 4.1: Overview of translation from fragments to blue diagram



$$
\begin{aligned}
&S = \{0, 1, 2, 3\} \\
&p = 1 \\
&q = 2 \\
&f_{in}(0) = 0 \qquad f_{out}(0) = (0, 0) \\
&f_{in}(1) = 1 \qquad f_{out}(1) = (0, 1) \\
&f_{in}(2) = 0 \qquad f_{out}(2) = (0, 0) \\
&f_{in}(3) = 1 \qquad f_{out}(3) = (1, 0) \\
&R = \{(0, 1), (1, 2), (2, 3), (3, 0)\}
\end{aligned}
$$

Figure 4.2: An example BD with its graphical representation

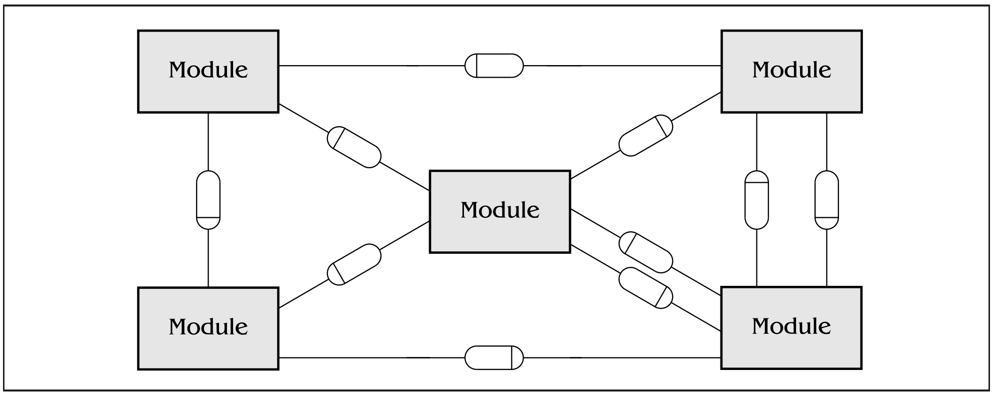


Figure 4.3: Network of modules connected in a DI way

usually left off and the initial state distinguished in some other way; b2ps sets the intial state in a bold face.

Blue diagrams can be viewed as state graphs where output changes are assumed to be instantaneous, or as burst-mode diagrams where the states are labelled rather than the arcs between states. When circuits are derived from blue diagrams, precautions must be taken to ensure that the non-zero response time of real circuit elements will not cause erroneous behaviour.

Blue diagrams are used as a specification of the interface behaviour of a module, assuming delay-insensitive interconnections to other modules. Consider two networks of modules connected by unbounded finite delays, both looking something like Figure 4.3. The first network, $\mathcal{N}$, has modules with nonzero internal delays that are bounded above by m; the second, Z, has the same modules but with zero internal delay, and the same connection between modules as $\mathcal{N}$. Wire delays are in the range $(0, \infty)$ in both networks, and are assumed to be variable on a per-signal basis. Are these two networks capable of the same set of behaviours?

- It is clear that any behaviour exhibited by $\mathcal{N}$ can also occur in Z, because the non-zero module delays in $\mathcal{N}$ can be absorbed into the wire delays in Z, giving a set of actions that could be observed in Z.

- Assume we have a finite sequence s of signal events in Z. Let w be the smallest delay that occured on any wire in this sequence. We may scale up the wire delays by any constant factor, which keeps the ordering of all signals the same, so scale up all wire delays in the sequence s by m/w to produce a sequence s'. Now all wire delays in s' are m or more, so when the intrinsic delay of an element in $\mathcal{N}$ is subtracted from the delay between two events in s', there will be a non-negative delay left. This non-negative delay can be ascribed to the inter-module delays in network $\mathcal{N}$, giving a valid sequence of events in $\mathcal{N}$.

This informal argument shows that a network of bounded-response circuit modules under a DI wire assumption has identical behaviour to a network of instantaneous decision-making elements under the same wire assumption, so it is valid to use a blue diagram to describe a circuit which has non-zero delays between inputs and outputs. It is always assumed that there are finite unbounded delays on the *outputs* of an element specified by a blue diagram, but not the inputs; delays on the inputs are taken to be the output delays of some other element. The type of the delay—pure, inertial or some combination of the two [22]—is unimportant, because there will never be two transitions propagating through a delay at the same time.

Definition 2 *If R instead has the property that $(a, b) \in R \Rightarrow d(a_{in}, b_{in}) \leq 1$, then $(S, p, q, f_{in}, f_{out}, R)$ is an Extended Blue Diagram or XBD.*

XBDs can exhibit hazards on output wires, so they do not always correspond to a useful specification. A state graph is an XBD, because at most one input changes between a state and its successor. Similarly, the projection of a state graph or Blue Diagram onto a subset of its inputs and outputs is also an XBD. By repeatedly collapsing pairs of states a and b such that $a \longrightarrow b$ and $a_{in} = b_{in}$ into a single state c with $c_{in} = b_{in}$ and $c_{out} = b_{out}$, an XBD can either be transformed into a BD, or seen to have output hazards.

Definition 3 *Given a blue diagram S, if $\exists s, x, y \in S$ such that $s \longrightarrow x, s \longrightarrow y, x \neq y$ and $x_{in} = y_{in}$, then S is* non-deterministic*; if there are no such s, x, y, then it is* deterministic.

Definition 4 *Given a blue diagram S, if $\exists s, u, v, x, y \in S$ such that $s \longrightarrow u, u \longrightarrow v, s \longrightarrow x, x \longrightarrow y, u_{in} \neq x_{in}, v_{in} = y_{in}$ but $v \neq y$, then S is* non-semi-modular*. If there are no such s, u, v, x, y, then S is* semi-modular.

A semi-modular BD is one where the ordering of concurrent input transitions does not matter; it could alternatively be called an *order independent* diagram. An obvious non-semi-modular diagram is the one for a Seitz arbiter, where the precise timing between requests affects which grant is asserted. Non-deterministic diagrams are only useful for specifying the environment for a circuit, where they represent input choice.

Figure 4.4 shows a first attempt at defining the interconnections between a circuit and its environment. Technically, the environment behaviour does not need to be specified at all, because it can be inferred from the blue diagram for the circuit. However, concurrency reduction will change the circuit behaviour while leaving the environment alone, so it is useful to have the environment specified in some way. It also makes sense to use the same specification for the environment as for the circuit, although state graphs or trace expressions could be used instead.

4.2 Example circuits

Any useful specification style must be able to describe a wide variety of possible circuits, so the specification was created with a number of representative design problems in mind. Five of these were abstract descriptions of the required behaviour, each of which required recasting as a set of STG fragments, possibly with

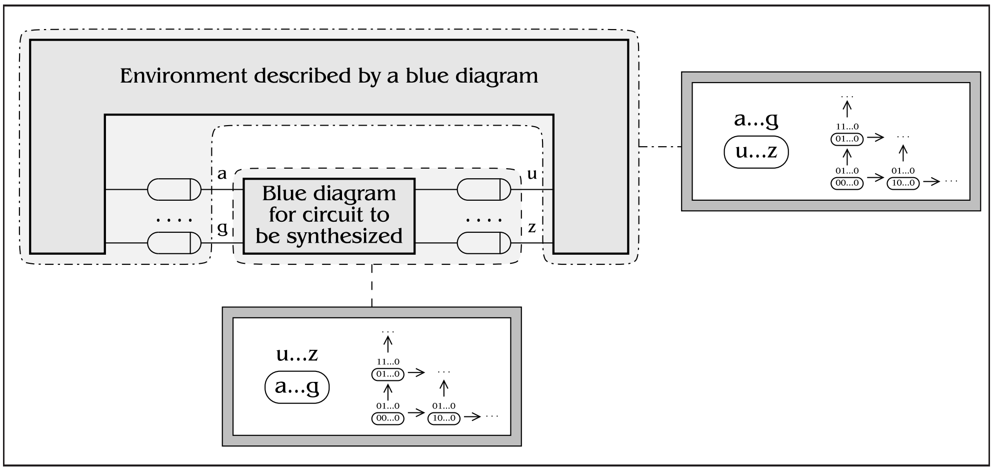
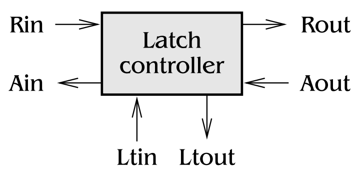


Figure 4.4: First model of connections between a circuit and its environment



Rin+ → Ain+ → Rin− → Ain− → Rin+
Rout+ → Aout+ → Rout− → Aout− → Rout+
Ltout+ → Ltin+ → Ltout− → Ltin− → Ltout+
Rin+ → Rout+
Rin+ → Ltout+ → Ltin+ → Ain+
Aout+ → Ltout− → Ltin− → Rout+

Figure 4.5: Latch controller specified by STG fragments

some additional features. The other examples were taken from the set of standard SIS STG benchmarks [101], which do not really count as design examples because they have already been written as STGs; they were included to make sure that the specification style was powerful enough for most examples that will be encountered.

4.2.1 The Furber/Day latch controller

The STG fragments for this circuit have been discussed already, but are shown again in Figure 4.5 for completeness. A blue diagram can be constructed by interpreting these fragments as a Petri net, adding arcs from x+ to x− and back for all signals x, which gives the net shown in Figure 4.6, and then simulating this net. If, during this simulation, it is assumed that any excited output transition will occur without delay and simultaneously with the input that caused it, then a blue diagram is formed rather than a state graph.

Unnecessary arcs and places, of which there are many, have been shaded in Figure 4.6. Most of these are arcs that were added from x+ to x− and back for all x, but these are useful for reasons that will be covered in Section 4.4.1.

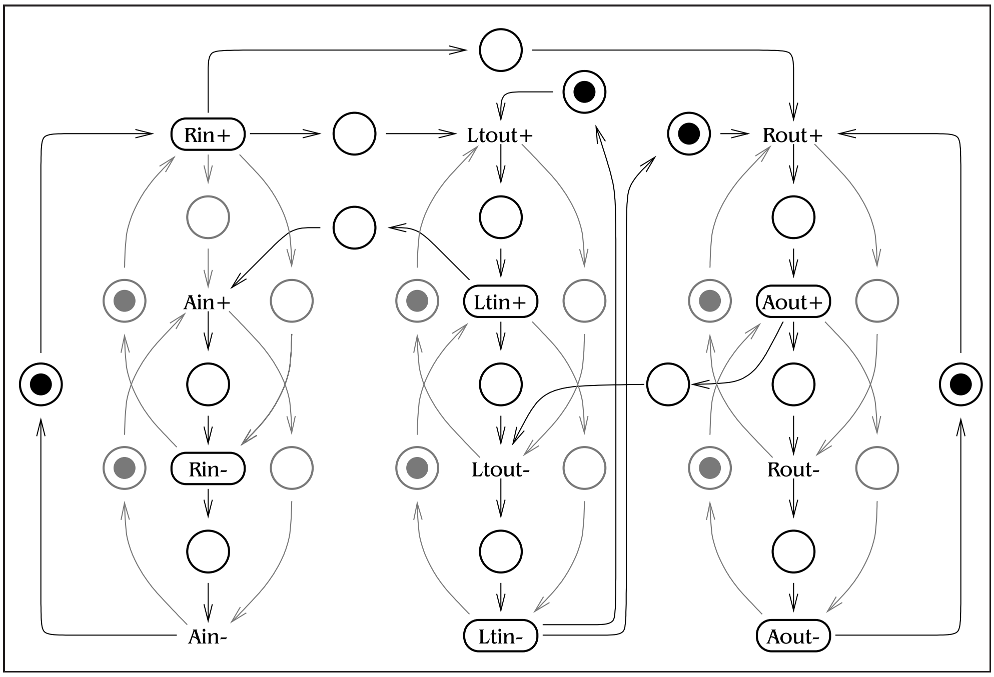
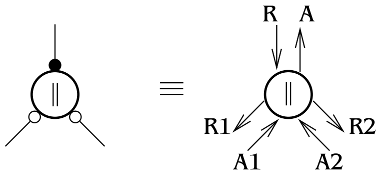


Figure 4.6: Intermediate Petri net for latch controller example



R+ → R1+ R1+ → A1+ A1+ → A+
R+ → R2+ R2+ → A2+ A2+ → A+
R+ → A+ → R– → A– → R+
R1+ → A1+ → R1– → A1– → R1+
R2+ → A2+ → R2– → A2– → R2+

Figure 4.7: Parallel component specified by STG fragments

4.2.2 Abstract definitions of more example circuits

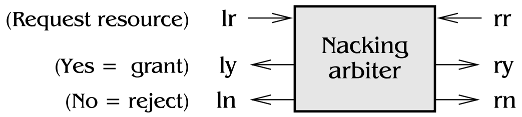
This section gives a high-level description of the other four circuits that were used as representative examples of asynchronous circuit behaviour.

Tangram-like parallel component [10]

This is shown in Figure 4.7, and is similar to the latch controller. It too can be specified using only STG fragments.

Nacking arbiter

The nacking arbiter is also known by the names *non-blocking arbiter* and *arbiter with reject*. It is an arbiter that is capable of refusing access to a shared resource, allowing the requester to get on with something else until the resource becomes free. The design considered here is the four-phase version shown in Figure 4.8;



Resource is free:

lr+ → ly+ → lr– → ly–
Request Grant
rr+ → ry+ → rr– → ry–

Resource is busy:

lr+ → ln+ → lr– → ln–
Request Reject
rr+ → rn+ → rr– → rn–

Figure 4.8: Nacking arbiter specification

other variants are possible.

Rising edges on lr and rr request the resource from the arbiter. If the resource is free, the arbiter signals with the grant wire ly+ or ry+, and the requester is allowed to access the resource. The four-phase handshake is completed as fast as possible, with no meaning attached to the falling edges of signals. To free the resource, another request is made (lr+/rr+), but this time the reject wire is asserted (ln+/rn+) to signify that the resource is now free, and the four-phase handshake again completed. If the resource was busy when the request was received, a handshake occurs on the reject wire instead of the grant wire.

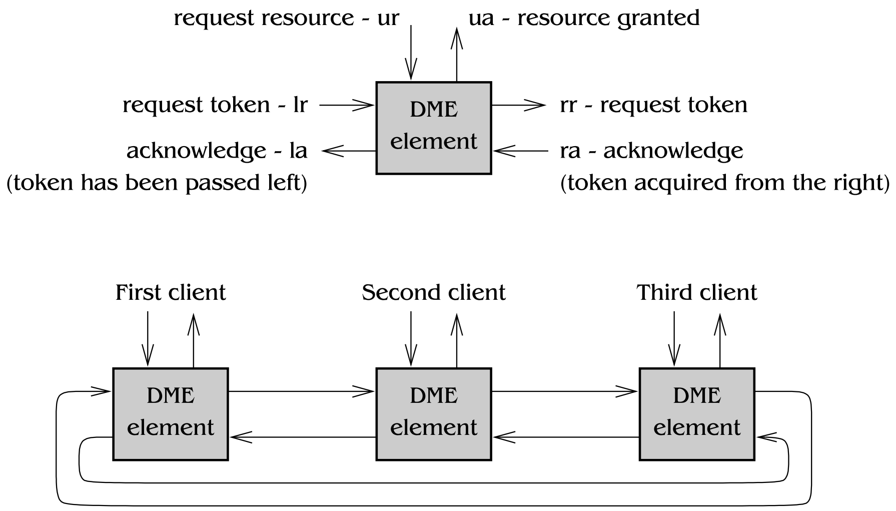
This circuit requires some method to cope with the metastability that can result when lr+ and rr+ happen almost simultaneously.

Martin's Distributed Mutual Exclusion element (DME)

Martin's DME element [116] is an ingenious solution to the multi-way arbiter problem, using n identical circuit elements to arbitrate between n parties, as shown in Figure 4.9. One of the DME elements initially has the *token*, and is the only element that can grant the resource. If a DME element, say at position i from the left, receives a request for the resource and it does not have the token, it can request the token from the element $(i+1)$ on its right by initiating a handshake on rr. When element i receives ra+ from its right, the token is deemed to have moved from element $(i+1)$ to element i. If element $(i+1)$ does not have the token, the request is passed on to element $(i+2)$ and further to the right until the token is found, then the token passed all the way back and the resource finally granted. Unlike some similar circuits, the token only moves when a request has been received; it is not constantly moving round the ring.

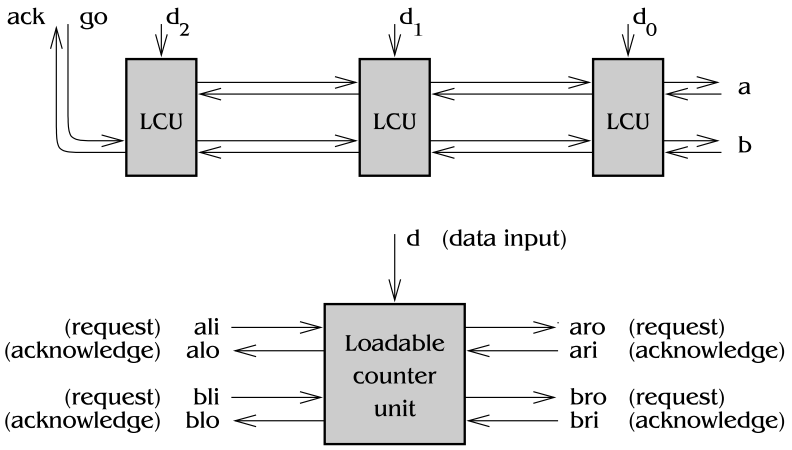
The loadable counter: a design problem from ACiD-WG 1996 [111]

The problem is to find a circuit which, when presented with a binary number k on a bus and given a 'go' signal, will give k handshakes on output port a, followed by a single handshake on output port b and then acknowledge the 'go' signal. A possible decomposition is given in Figure 4.10. It is assumed that handshakes on ali/alo and bli/blo cannot occur at the same time.



```
When a rising edge on lr or ur is received
    If you have the token
        Acknowledge the request
        If the request was on lr
            We do not now have the token
        Complete four-phase handshake on lr/la or ur/ua
    Else
        Get the token: assert rr+ and wait for an ra+
        Complete the four-phase handshake on rr/ra sometime
        Acknowledge the request on lr/ur
        If the request was on ur
            We now have the token
        Complete four-phase handshake on lr/la or ur/ua
    Endif
```

Figure 4.9: Martin's DME element



```
If ali+ received
    Complete two handshakes on aro/ari
    Acknowledge alo+ and complete ali/alo handshake
Else if bli+ received
    If (d == 1)
        Complete one handshake on aro/ari
    Endif
    Complete one handshake on bro/bri
    Acknowledge blo+ and complete bli/blo handshake
Endif
```

Figure 4.10: The loadable counter example

4.2.3 Examples from the SIS benchmarks

Some examples were chosen from the standard set of SIS STG benchmarks [101]. Rather than use the whole set, examples were chosen that appear to have been used often in published papers. Most of the examples used look like they have been originally derived from burst mode machines, so they are mostly sequential and will not be affected by the concurrency-reducing transformations that are a central topic in this dissertation. However, these examples show that the specification and synthesis tools are powerful enough to cope with most real examples, even if the concurrency-reduction tool cannot be used for them. Versions with a *.nousc* suffix were preferred to versions without if both were present; the *.nousc* ending denotes the STGs that have not had the Unique State Coding property enforced, which gives a little more implementation freedom. The examples, shown in Figures 4.11–4.23, are *alloc-outbound*, *atod*, *isend*, *master-read*, *mp-forward-pkt*, *nak-pa*, *nowick*, *pe-send-ifc*, *ram-read-sbuf*, *rcv-setup*, *rlm*, *sbuf-ram-write*, and *sbuf-read-ctl*.

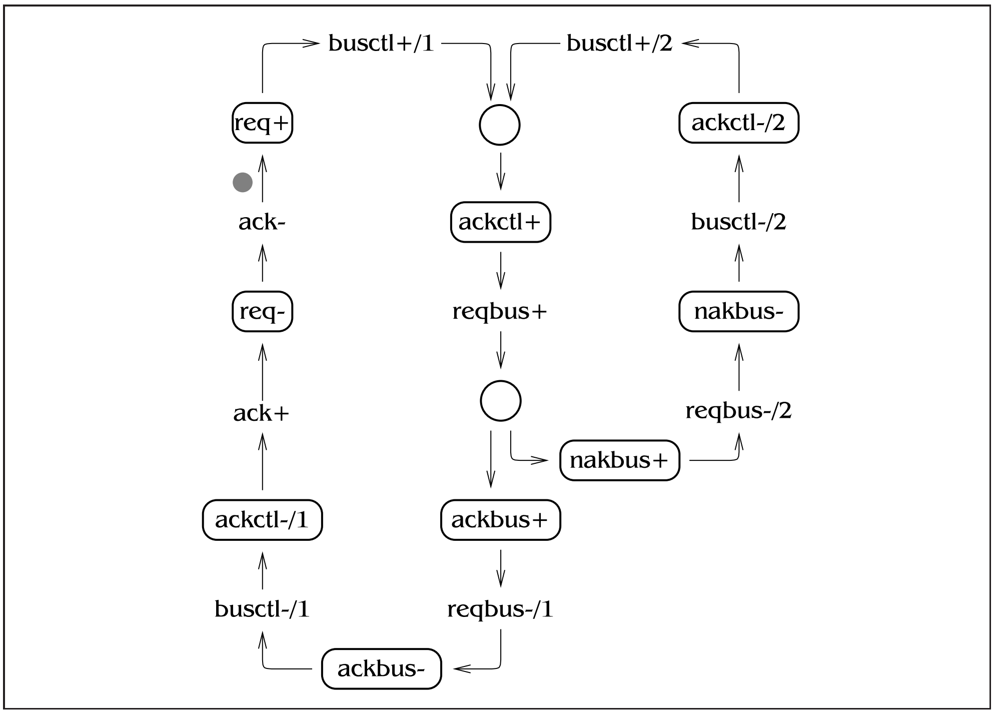


Figure 4.11: Example *alloc-outbound*

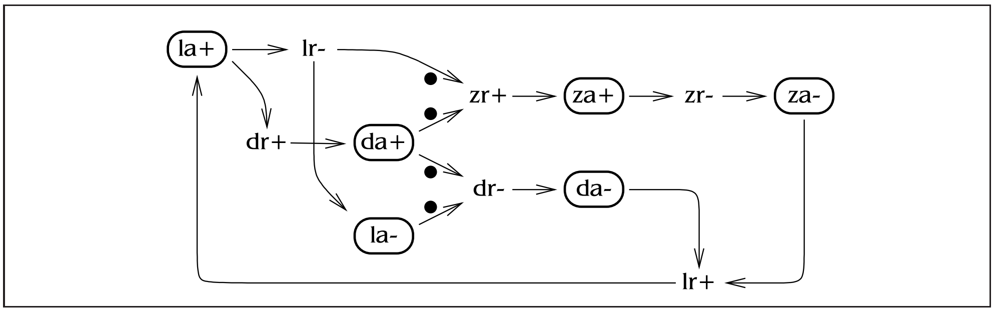


Figure 4.12: Example *atod*, from T.A.Chu's thesis [28], page 133

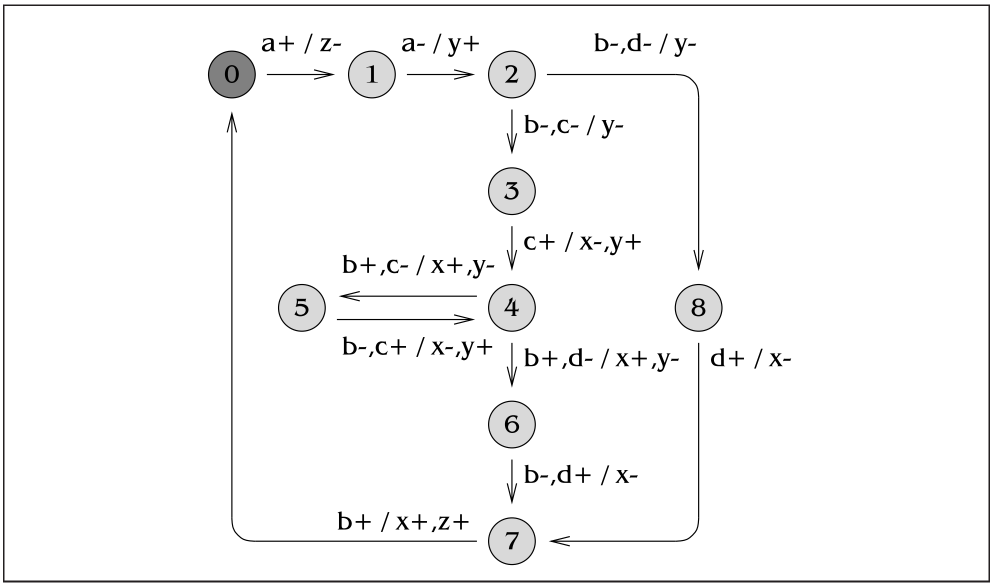


Figure 4.13: Example *isend*

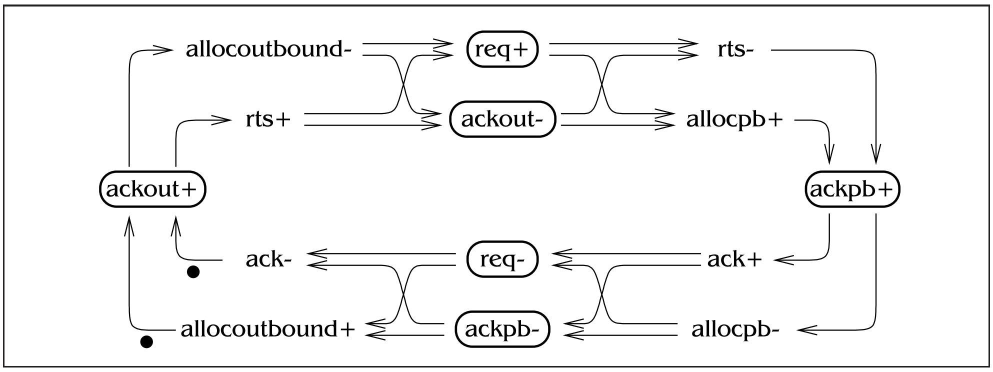


Figure 4.14: Example *mp-forward-pkt*

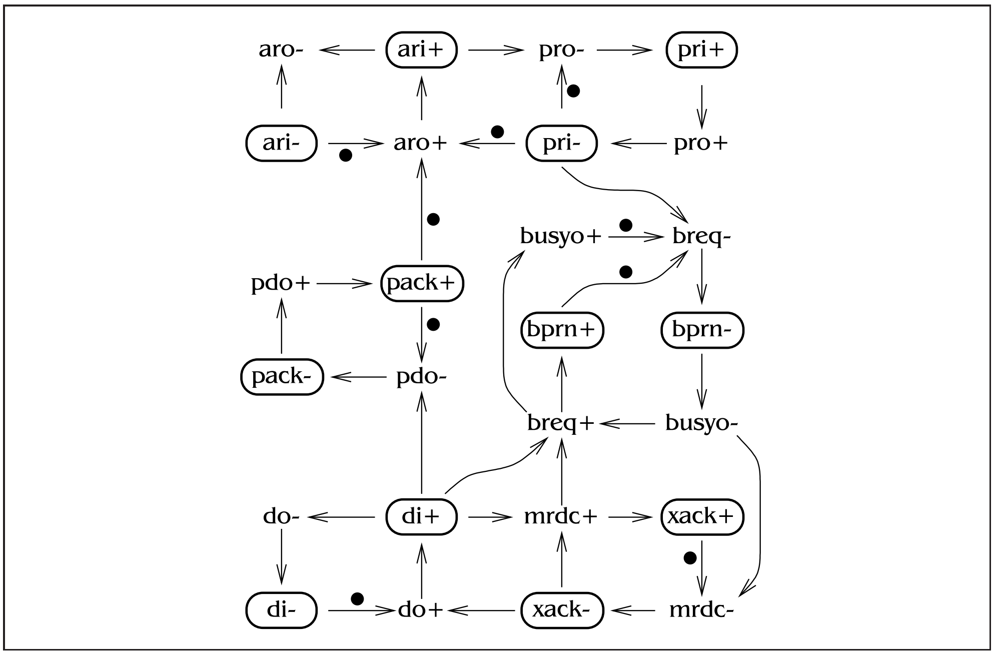


Figure 4.15: Example *master-read*

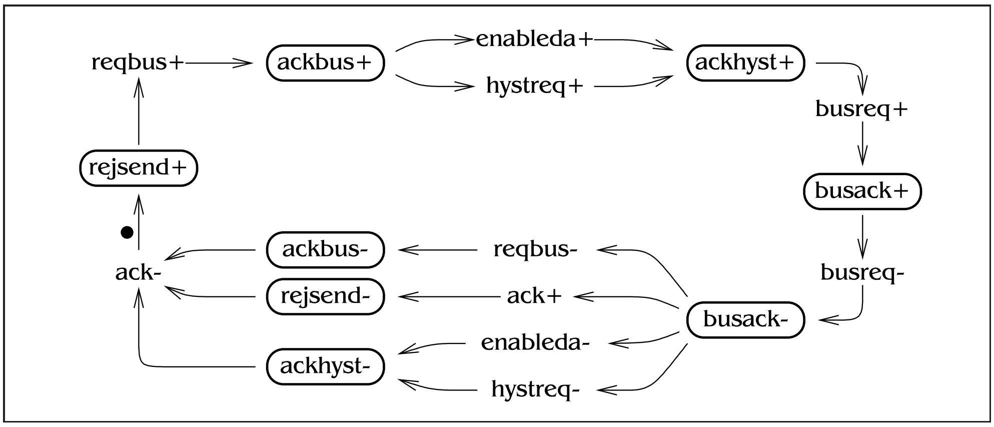


Figure 4.16: Example *nak-pa*

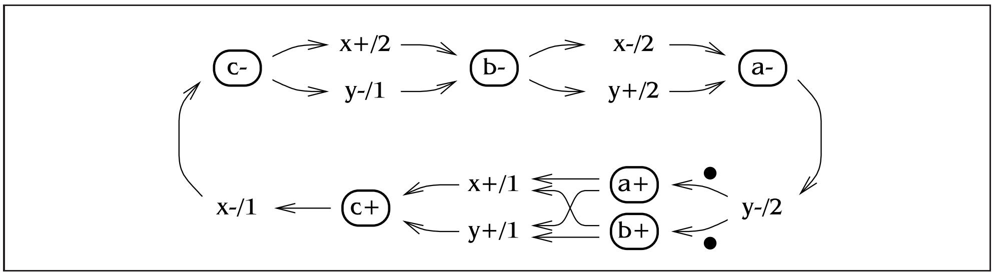


Figure 4.17: Example *nowick*

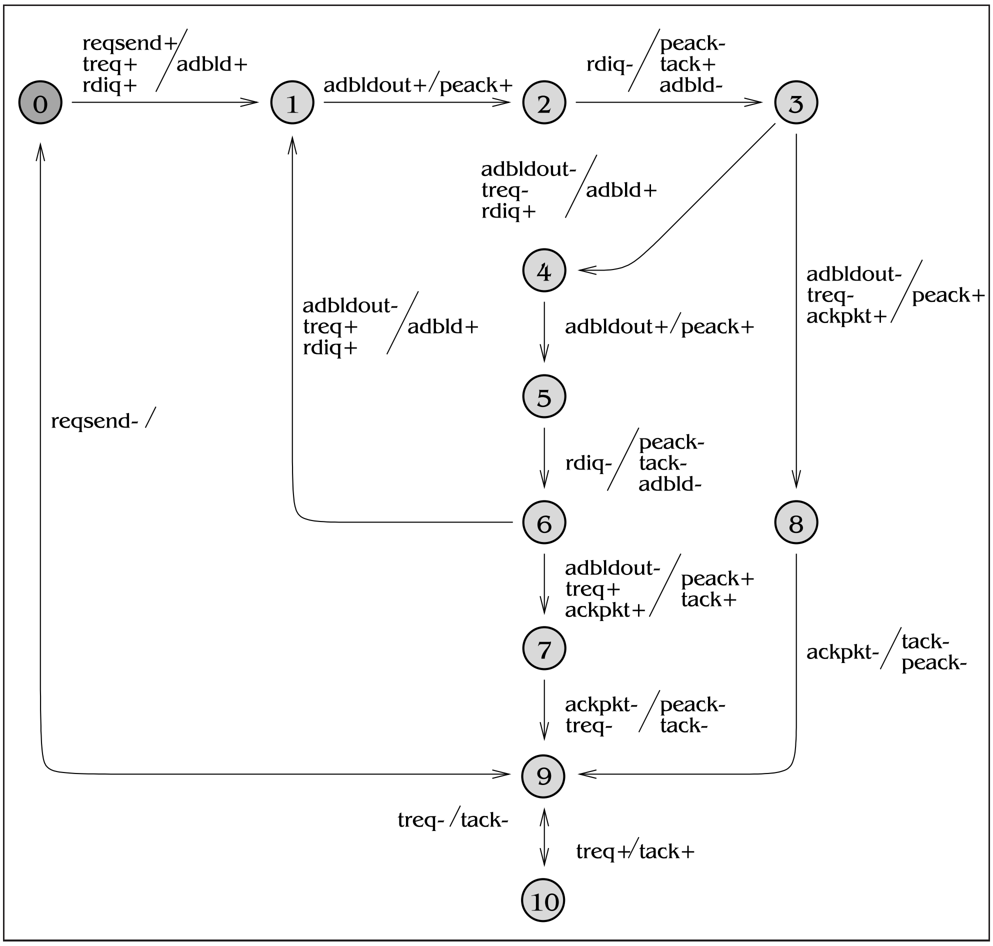


Figure 4.18: Example *pe-send-ifc*

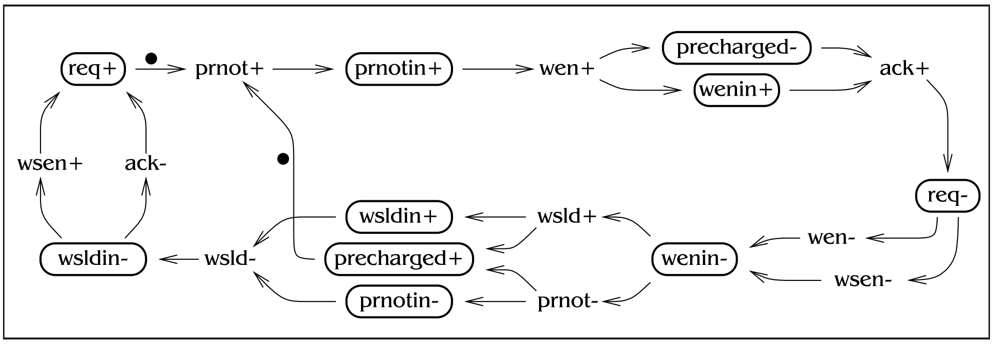


Figure 4.19: Example *ram-read-sbuf*

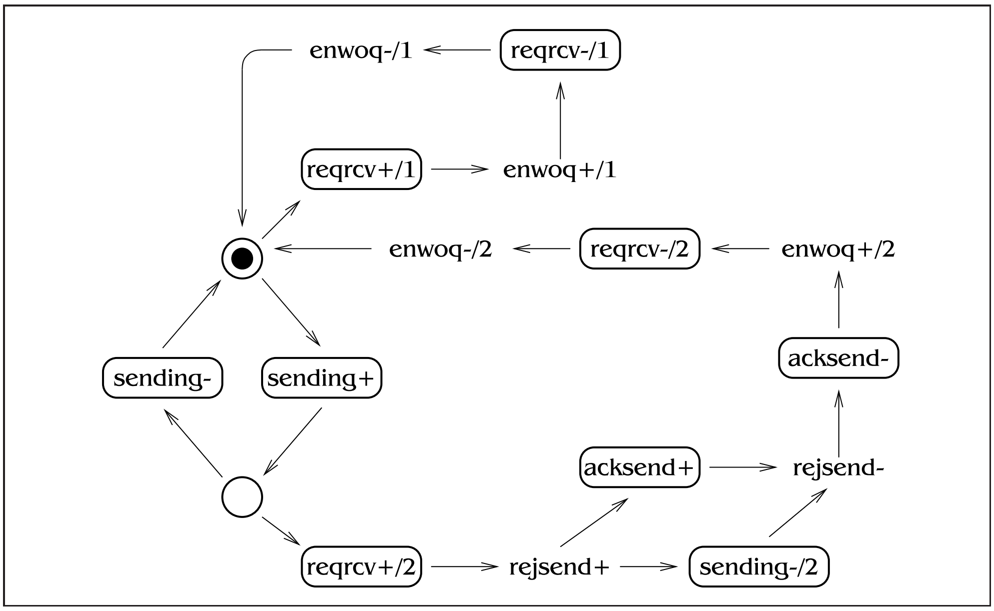


Figure 4.20: Example *rcv-setup*

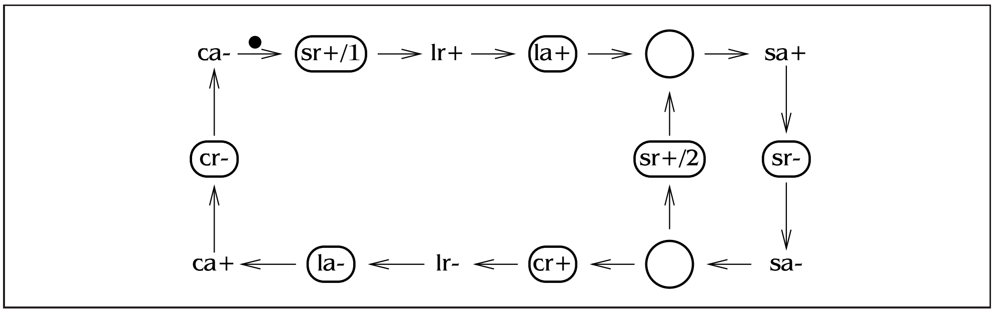


Figure 4.21: Example *rlm*, from Chu's thesis [28], page 172

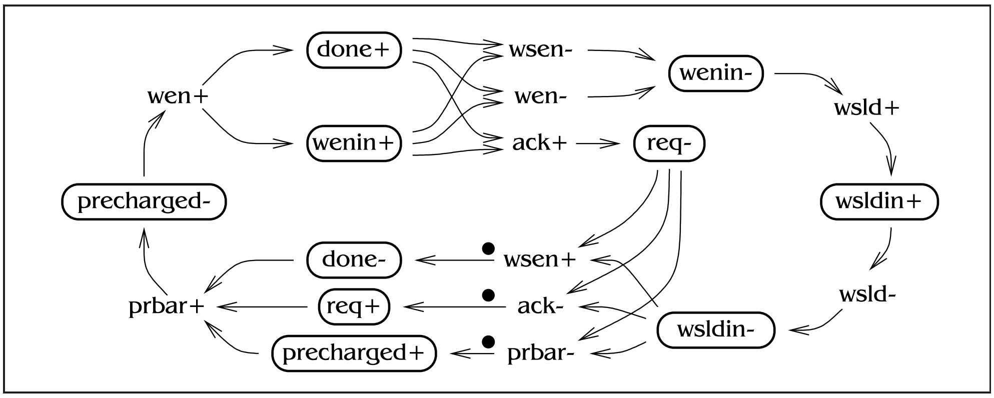


Figure 4.22: Example *sbuf-ram-write*

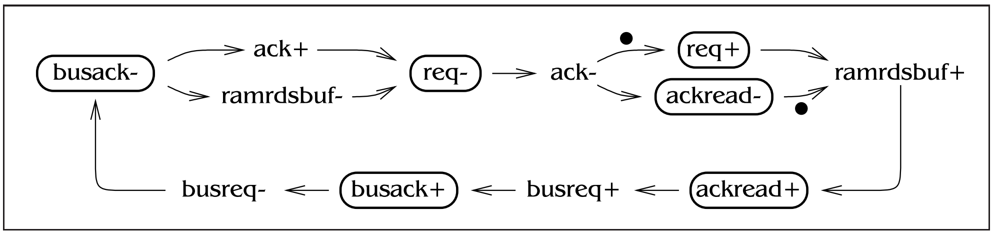


Figure 4.23: Example *sbuf-read-ctl*

4.2.4 Inadequacies of the simple interconnection model

Figure 4.4 showed a first attempt at naming the wires that are needed between a circuit and its environment, but from the examples given above, it can be seen that this model will not be sufficient for most useful circuits. Three problems can be seen to arise:

- *Outputs causing outputs requires new wire labels.* If an output causes another output, such as `lr-` causing `zr+` in *atod* (Figure 4.12), then the two transitions may occur in either order when received by the environment. What is meant by `lr-`⟶`zr+` is that on the outputs of the circuit, `zr+` will follow `lr-` by a small delay; it does not mean that the environment will see `lr-` before `zr+`, which is what an ordering between the labelled transitions in Figure 4.4 would give. The circuit outputs need to be labelled in front of the wire delays to allow this behaviour to be specified. This also happens at several points in *master-read* (Figure 4.15).

  This problem occurs because STGs, and hence STG fragments, use the speed-independent delay model, but the delay model used with blue diagrams is delay-insensitivity on interconnecting wires; observing that a transition has been sent is not the same as knowing it has been received when using blue diagrams, but it is in the STG fragments.

- *Inputs causing inputs requires a wire back.* If an input causes some other input, for example `sending+` causing either `sending-` or `reqrcv+/2` in *rcv-setup* (Figure 4.20), and the circuit must see the transitions occurring in order, then some mechanism must be included to force this ordering. It is not sufficient for the environment to send a `sending+` transition, then wait for any bounded time before giving `reqrcv+/2`, because the unbounded wire delays can swap the ordering of the two signals from the circuit's point of view. A wire must be included from the circuit end of the `sending` wire back to the environment, so the environment can tell when `sending+` has been received before giving another transition. This also happens in *pe-send-ifc* (Figure 4.18) when `reqsend-` causes `reqsend+`, and the loadable counter (Figure 4.10) when the `d` input must change and be stable before either an `ali+` or `bli+`.

  Note that this extra wire is only a *model* of the behaviour of the environment; in reality, there will be some upper bound on the delays in the circuit, which

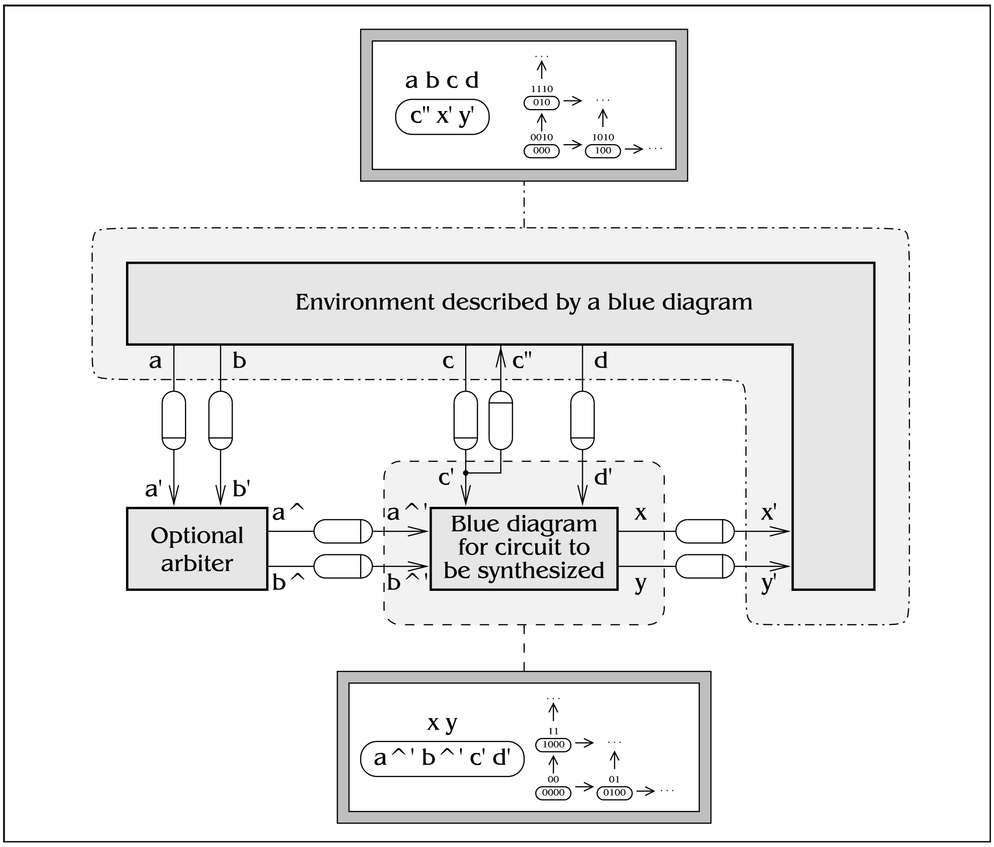


Figure 4.24: Improved model of connections between a circuit and its environment

the designer can use at a global level to make this extra wire unnecessary. Precisely how this is done is outside the scope of this dissertation.

- *Arbiters are required.* In the nacking arbiter and DME circuits (Figures 4.8 and 4.9), an arbiter is required on the input side of the circuit to remove metastability problems. This is similar to the approach used by Cortadella et al. [43, 44]. Ivan Sutherland has pointed out that the use of *observing arbiters* would lead to increased speed, an approach that was used for a fast tree arbiter by Josephs and Yantchev in [80] and Yakovlev et al. [194], but this will be left as a possible extension to the work.

The changes listed above are shown in Figure 4.24, which will be referred to as the improved model of module interconnections. Primes (x', x'') are used to indicate that a signal has been through an unbounded wire delay, and a postfixed circumflex ($x\hat{}$, $x\hat{}'$) to show the output of an arbiter. A given transition x in the STG fragments may need to be interpreted as x', x'', $x\hat{}$ or $x\hat{}'$, depending on its context. The arbiter used, Figure 4.25, is a CMOS version of the design proposed by Seitz [160, 161]. The arbiter is optional, and does not occur in most of the examples.

The connections to the arbiter are overly pessimistic—it is assumed that there

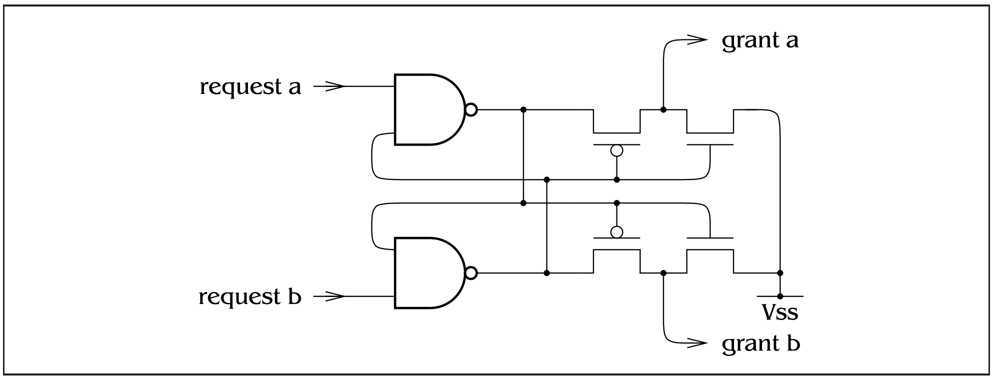


Figure 4.25: A standard arbiter unit: the Seitz arbiter

are unbounded finite delays between the arbiter and the circuit. In reality, the arbiter and the circuit will be very close together, so there will be almost no delay between the arbiter and the circuit. However, problems occur if these delays are left out. When the arbiter changes state between granting one request and granting the other, its outputs briefly go through a state in which neither is granted; ie. the outputs of the arbiter go 01 → 00 → 10. If it is assumed that the arbiter outputs are connected directly to the inputs to the circuit, then the synthesis tools often produce a circuit which *relies* upon the existence of the neither-grant state. In practice, the neither-grant state persists for such a short time that the implementation does not manage to see it, so the implementation fails. Adding extra delays between the arbiter and the circuit allows the inputs to the circuit to go through either 01 → 00 → 10 or 01 → 11 → 10, so the circuit cannot rely on the neither-grant state for correct operation, and hence the circuits produced are more robust.

4.3 The specification language

This section describes the creation of a specification style that is based upon STG fragments. An obvious choice that must be made is whether to make the user interface graph-based or text-based; for a first attempt, it would be easier to use a text-based approach and create a graphical front-end later on, if necessary.

It would be sensible to make the form of the specification similar to an existing text-based style. It would also be good to be able to integrate the synthesis tools with an existing synthesis environment. Verilog is used in the Computer Laboratory for the design of synchronous and asynchronous circuits, so it was decided to make the specification look similar to Verilog. The idea is that, ultimately, asynchronous specifications based on STG fragments can be included in Verilog files. Synthesis of the Verilog file could then invoke a program to extract asynchronous modules, synthesize them, and substitute a circuit layout back in the original file. The integration of the tools into Verilog would not constitute original research, and will not be attempted in this dissertation. Figure 4.26 gives an example Verilog file [112, page 6] showing the file format.

```
module dff (q,qb,clk,d,rst);
   input clk,d,rst;
   output q,qb;
   wire dl,dbl;
   // Verilog-specific statements; STG fragments would go here
endmodule
```

Figure 4.26: An example Verilog definition, showing the file format

4.3.1 Extending STG fragments

When converting the Furber/Day latch controller to a blue diagram, a Petri net was used as an intermediate representation. Petri nets are very general, so there will not be a problem representing all the examples as Petri nets. A specification file consisting of just STG fragments will not be sufficient to describe several of the examples, so some way must be found to make this description more powerful. Two different ways are possible:

- Allow places and dummy transitions to be specified in the file, so that the STG fragments become simply Petri nets or STGs.

  Advantages:

  - Petri nets are a well-known specification.

  Disadvantages:

  - It may be difficult to determine what $p \longrightarrow \ldots \longrightarrow q$ means in the specification if there are arbitrarily many dummy transitions and places between p and q in the specification. Does it mean $p \rightarrow q$, $p' \rightarrow q$ or $p'' \rightarrow q$?

- Add language constructs to allow any additional features that are required.

  Advantages:

  - It will be easier to provide helpful error messages if basic assumptions are violated, e.g. if an input signal is arbitrated against an output signal, or if the circuit presents a non-DI interface.
  - Typical Computer Scientists or Electrical Engineers may find language constructs such as AND and OR to be easier to use than place/transition relationships.
  - A similar but different specification to Petri nets will serve to remind users that the underlying delay model is not the same as that usually used for Petri nets.

  Disadvantages:

  - Some Petri net knowledge will always be required, because tokens cannot be automatically placed in the generated intermediate Petri net.

On balance, it seems that creating a new specification language that includes STG fragments and keywords such as `arbitrate` would be more useful than using Petri nets. A question that could be asked at this point is "Why does the world

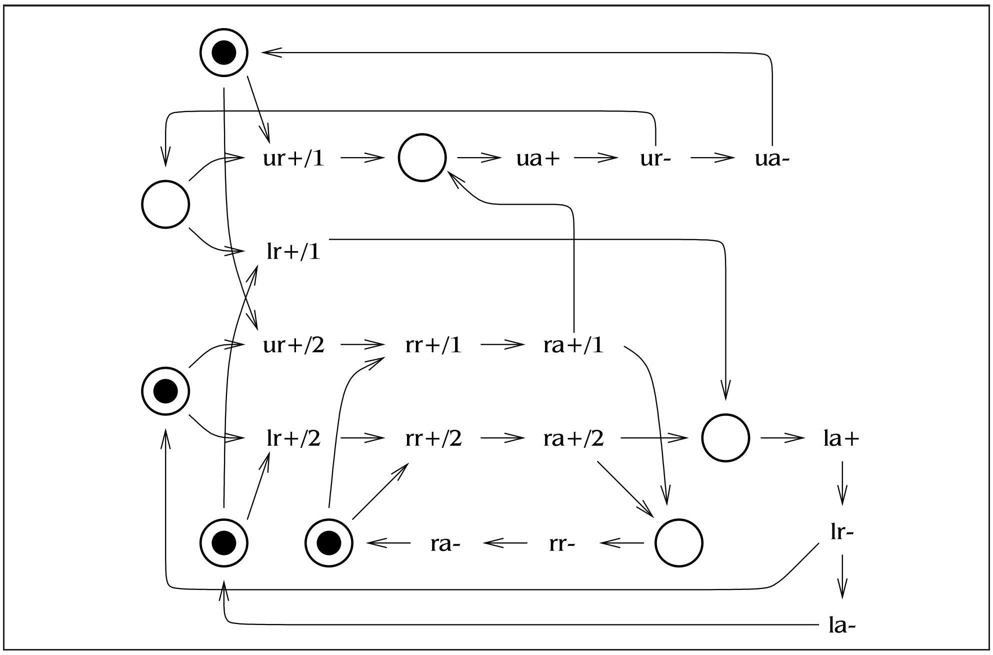


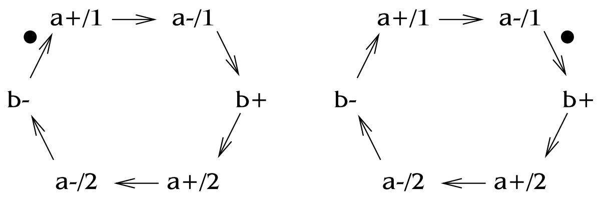
Figure 4.27: STG for Martin's DME element

need yet another form of specification for asynchronous circuits? What is wrong with simply translating an existing form, such as an STG, to a Blue Diagram?" The answer is concurrency: in an STG, all transitions must be specified, with temporal constraints placed between many of them. Not all of these constraints will be necessary for the circuit to function correctly; the others may have been put in to satisfy persistency, unique state coding, or maybe just so the designer has a clearer picture of the operation of the circuit. An example of this is shown in Figure 4.27, showing an STG for Martin's DME element; while creating this STG, it was tempting to place an ordering between `rr-`⟶`ra-` and `la+` or `ua+`, rather than allowing the separate handshakes to operate concurrently. This would have made the STG easier to design and read, but made the final circuit slower. A new specification language will allow the designer to specify the important relationships between transitions, leaving the translation tool to take care of the falling edges of handshakes and any correctness constraints. This gives the concurrency reduction tool more freedom to perform transformations on the Blue Diagram produced, and allows a larger set of possible solutions to be explored. This conclusion has also independently been reached by Cortadella et al. [39] while this dissertation was being written.

Figure 4.27 also shows, when compared to the specification for the DME element in Figure 4.33, that the proposed specification style can be significantly more readable than an STG.

Features will now be described that are needed in the specification language.

Transitions: Transitions in SIS and other tools have a means to distinguish two different transitions of one variable in the same direction, e.g. `a+/1` and `a+/2`. This mechanism must exist in the language.



Initial state: a = b = 0

This STG has six places but only four states of the two signals a and b, so some values of a and b correspond to more than one marking. Either of the two markings given could be the intial marking for a = b = 0. Any net without the USC property will have this problem, so the designer must place tokens manually in the net.

Figure 4.28: A problem with automatic placement of tokens

Arrow operator: Causality between transitions will be denoted by the arrow operator ⟶, written `->` in a text file. On some arcs, tokens will need to be explicitly included, because in general it is not possible to tell from the structure of the net where tokens should be placed—see Figure 4.28. A token on an arc from `a+` to `b+` will be written `a+` ⟶ `token` ⟶ `b+`.

Data inputs: The `d` input to the loadable counter does not have rising and falling transitions; rather, it is a level input that is guaranteed to be stable before an `ali+` or `bli+`. SIS does not have this functionality, but ASSASSIN does; on page 20 of the ASSASSIN manual [199], `SDA*` was used to denote a data signal `SDA` going unstable, and `SDA&` denoted when it became stable at some value. The generic form of an ASSASSIN-style data input would be $A \longrightarrow$ `d*` ⟶ `d&` ⟶ B, so for brevity this will simply be written $A \longrightarrow$ `d*` ⟶ B. This latter form is a bit of a fudge—the place between `d*` and `d&` must still exist, but is not visible in the shortened version—yet the meaning of the shortened construct is still clear.[1]

And/Or: Several of the examples—the loadable counter, *alloc-outbound*, *rlm*, *rcv-setup*, *isend* and *pe-send-ifc*—demonstrate input choice: the ability of the environment to nondeterministically choose one out of a number of alternative transitions to fire. Without explicit places, an `or`[2] keyword must be provided. If used inside the circuit, this would create a nondeterministic circuit, which should be disallowed. An `and` keyword is strictly unnecessary, because the Petri net firing rule will create an AND function anyway, but the language will be more elegant if it has both `and` and `or` operators.

The `and` and `or` operators should take a pair of transitions, and create an object that behaves like another transition. This will allow constructs such as `a+` ⟶ (`b+ and`

[1] In fact, the statement `a+` ⟶ `d*` ⟶ `b+` can be written using other language constructs as `(a+ or d+ or d-)` ⟶ `(d+ or d- or b+)`, so data inputs are not required, but the form using `d*` is much more natural.

[2] Purists may require this to be `xor`, but I am assuming the English meaning of *or*, rather than the Boolean meaning. The synonym `xor` is also provided.

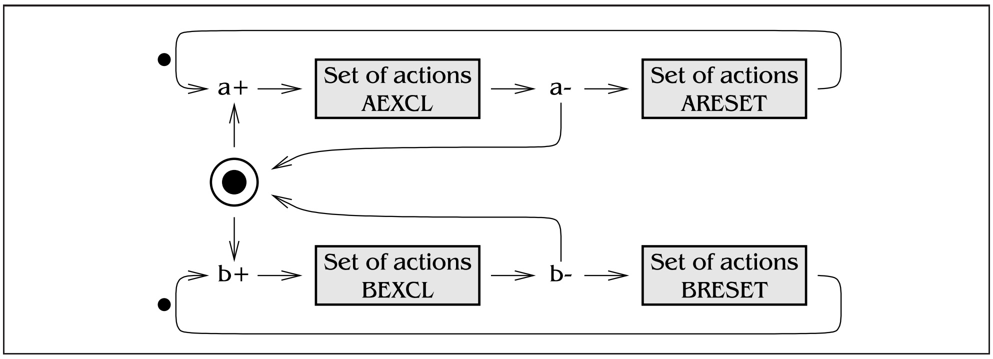


Figure 4.29: How arbitration appears to the designer

c+) → d+, and give it the meaning that a+ causes both b+ and c+, and then after both b+ and c+ have fired, d+ can happen. If the → operator is treated in the same way, this will allow structures like a+ → ((b+ → c+) or d+) → e+, meaning that a+ causes either of b+ or d+, after a b+ comes a c+, and either of c+ and d+ causes e+.

Arbitration: Arbitration needs a special language construct because an arbiter must be created to resolve metastability. Arbitration can be thought of as being described by the Petri net given in Figure 4.29. Sets of actions AEXCL and BEXCL are mutually exclusive due to the action of the arbiter. The single transition a+ in Figure 4.29 actually corresponds to four transitions in the intermediate Petri net; in Figure 4.24, these would be a+, the environment raising a; a′+ the arbiter receiving the request; a^+, the arbiter granting the request; and a^′+, the circuit receiving the grant. Each of the other three transitions in Figure 4.29 similarly corresponds to four separate transitions. Sets of actions ARESET and BRESET are circuit actions that cause a+ or b+ to be re-asserted respectively. These do not have to be part of the arbitrate language construct, but including them would make the specification more modular and easy to read—someone reading the specification would not have to hunt down the rest of the file to find the conditions that cause another a+ to be asserted, for example.

The arbitration construct will be written as:

```
arbitrate
  a+ => (block AEXCL)  => a-
     => (block ARESET) => a+
| b+ => (block BEXCL)  => b-
     => (block BRESET) => b+
end
```

In this construct, the doubled arrows (=>) signify the same relationship between transitions as ordinary arrows (->), but the two must be distinguished so that the LALR(1) front end can parse the statement correctly.

If...then: The loadable counter has an input d that controls whether the counter should add one to the present count or not. It is not transitions on d that are important, but levels, and all the statements above deal only with transitions. Something is needed that will behave like the STG notion of *input choice*, or more recognizably, like an `if...then` statement in a programming language. This function could also be written as in Dijkstra's guarded command language [51], but of the three, an `if` statement is probably the easiest to understand for most people. The form of the `if` statement should be:

(Transition A) `->` `if` (condition) `then` (actions) `else` (actions) `endif`

The A transition at the start of the statement is important; it must be specified *when* the condition is to be tested, and here it should happen immediately after A.

Other features: The aim of the specification language is to make it easier to specify circuits, so common structures should be abbreviated as much as possible. A familiar structure in asynchronous circuits is the handshake `a+` ⟶ `b+` ⟶ `a-` ⟶ `b-` ⟶ `a+`, for which it would be useful to have a shorthand notation: `h/s (a,b)`. Note that it is possible to automatically insert tokens at the right place in the handshake, if the initial state of the signals is known. A synonym for `h/s` is `delay`, which models the effect of a delay external to the circuit, such as the latch driver delay in the Furber/Day latch controller.

Another useful shorthand is the `cycle` keyword, where `cycle (A` ⟶ `...` ⟶ `B)` is equivalent to `A` ⟶ `...` ⟶ `B` ⟶ `A`. This is useful in cases where `A` is long or complicated, or in cases such as `cycle (A or B)` to specify a choice between two distinct behaviours. The `cycle` keyword can be seen to be very useful in the loadable counter example, Figure 4.34.

4.3.2 BNF description of language

This section gives a definition of the specification language in a BNF-like format. The "+" symbol represents one or more copies of what it follows, "*" is zero or more copies, square brackets denote optional clauses, and a vertical bar within braces denotes a selection from different options. All whitespace is equivalent and serves to delimit strings and operators. The input file is of the form

`module` ⟨*name*⟩ `(`⟨*parm*⟩{`,`⟨*parm*⟩}*`)`
 ⟨*i/o declaration*⟩+
 ⟨*top level statement*⟩+
`end`

where

⟨*i/o declaration*⟩ = `input[s]` ⟨*i/o dec list*⟩
| `output[s]` ⟨*i/o dec list*⟩
| `internal[s]` ⟨*i/o dec list*⟩
| `external[s]` ⟨*i/o dec list*⟩
(`wire` is a synonym for `internal`)

⟨*i/o dec list*⟩ = ⟨*i/o dec stmt*⟩ {, ⟨*i/o dec stmt*⟩}*

⟨*i/o dec stmt*⟩ = {⟨*name*⟩ `=`}+ {`0`|`1`}

```
⟨top level statement⟩ = ⟨statement⟩
                      | h/s (⟨name⟩,⟨name⟩)
                        (delay is a synonym for h/s)
                      | cycle (⟨statement⟩)
                        ⎧ arbitrate
                        ⎪   ⟨transition⟩ => ⟨statement⟩ => ⟨transition⟩
                        ⎪                => ⟨statement⟩ => ⟨transition⟩
                      | ⎨ | ⟨transition⟩ => ⟨statement⟩ => ⟨transition⟩
                        ⎪                => ⟨statement⟩ => ⟨transition⟩
                        ⎩ end
⟨statement⟩           = ⟨statement⟩ or ⟨statement⟩
                      | ⟨statement⟩ and ⟨statement⟩
                      | ⟨statement⟩ -> ⟨statement⟩
                      | if (⟨if cond⟩) then ⟨statement⟩
                              [else ⟨statement⟩] endif
                      | (⟨statement⟩)
                      | ⟨transition⟩
⟨transition⟩          = ⟨name⟩ {+|-|*} [/⟨number⟩]
                      | token
⟨if cond⟩             = ⟨if cond⟩ or ⟨if cond⟩
                      | ⟨if cond⟩ and ⟨if cond⟩
                      | (⟨if cond⟩)
                      | ⟨name⟩
⟨name⟩                = Alphanumeric string, possibly including underscores
```

4.3.3 Specifications for the examples given

This section gives the specifications used for each of the example circuits, to illustrate the use of the language. The first five examples in Figures 4.30–4.34 are derived from abstract specifications, and show that quite complicated behaviour is easily expressible in the language. The other figures are translated from the STG or burst-mode specifications. The burst-mode specifications are particularly clumsy in the new language, but not as bad as when they were translated to STGs in the SIS benchmarks. In *isend* and *pe-send-ifc*, a transition x from state s_{from} to s_{to} is written as $x/s_{from}s_{to}$ to avoid name clashes.

It can be seen that the examples given show the structure of the corresponding STGs much better than the input format to SIS. This is especially true in Figure 4.45, the improved specification for *rcv-setup*, where the use of `sending*` has made the specification shorter and easier to understand.

```
module latchc (rin, ain, rout, aout,
               ltout, ltin);
inputs rin = aout = ltin = 0;
outputs ain = rout = ltout = 0;
h/s (rin, ain)
h/s (rout, aout)
delay (ltout, ltin)
rin+ -> (ltout+ -> ltin+ -> ain+)
        and rout+
aout+ -> ltout-
      -> ltin-
      -> token
      -> rout+
endmodule
```

Figure 4.30: Furber/Day latch controller

```
module parallel (Rin, Ain, R1, A1,
                 R2, A2);
inputs Rin = A1 = A2 = 0;
outputs Ain = R1 = R2 = 0;
h/s (Rin, Ain)
h/s (R1, A1)
h/s (R2, A2)
Rin+ -> (R1+ -> A1+)
          and
        (R2+ -> A2+)
     -> Ain+
endmodule
```

Figure 4.31: Parallel component

```
module narb (lr, ly, ln, rr, ry, rn);
inputs   lr=0, rr=0
outputs  ly = ln = ry = rn = 0
internal lhas = rhas = 0
arbitrate
  lr+ => if (lhas) then
           lhas- -> ln+
         else if (rhas) then
                ln+
              else
                lhas+ -> ly+
              endif
         endif
      => lr-
      => if (ly) then
           ly-
         else
           ln-
         endif
      => lr+
| rr+ => if (rhas) then
           rhas- -> rn+
         else if (lhas) then
                rn+
              else
                rhas+ -> ry+
              endif
         endif
      => rr-
      => if (ry) then
           ry-
         else
           rn-
         endif
      => rr+
end
endmodule
```

Figure 4.32: Nacking arbiter

```
module dme (lr, la, ur, ua, rr, ra);
inputs   lr = ur = ra = 0;
outputs  la = ua = rr = 0;
internal have_token = 0;
h/s (rr, ra)
arbitrate
  ur+ => if (!have_token) then
           rr+ -> ra+ -> have_token+
         endif
      -> ua+ => ur-
      => ua- => ur+
| lr+ => if (have_token) then
           have_token-
         else
           rr+ -> ra+
         endif
      -> la+ => lr-
      => la- => lr+
end
endmodule
```

Figure 4.33: Martin's DME element

```
// Loadable counter from
// ACiD-WG, Groningen.

module loadcnt (ali, alo, bli, blo,
                ari, aro, bri, bro, d)

inputs ali = bli = ari = bri = d = 0;
outputs alo = blo = aro = bro = 0;

h/s (ali, alo)
h/s (aro, ari)
h/s (bli, blo)
h/s (bro, bri)

cycle (
 token
  -> d*
  -> (ali+ -> aro+/1 -> ari+/1
           -> aro-/1 -> ari-/1
           -> aro+/2 -> ari+/2
           -> aro-/2 -> ari-/2
           -> alo+ -> ali- -> alo-)
     or
     (bli+ -> if (d) then
                (aro+ -> ari+ ->
                 aro- -> ari-)
              endif
           -> bro+ -> bri+ ->
              ((blo+ -> bli-)
              and (bro- -> bri-))
           -> blo-)
)

endmodule
```

Figure 4.34: Loadable counter

```
module alloc_outbound
  (req, ackctl, ackbus, nakbus,
   ack, busctl, reqbus);

inputs req = ackctl = ackbus
       = nakbus = 0;
outputs ack = busctl = reqbus = 0;

busctl+/1 or busctl+/2
  -> ackctl+
  -> reqbus+
  -> ackbus+ or nakbus+

ackbus+ -> reqbus-/1
  -> ackbus-
  -> busctl-/1
  -> ackctl-/1
  -> ack+
  -> req-
  -> ack-
  -> token -> req+ -> busctl+/1

nakbus+ -> reqbus-/2
  -> nakbus-
  -> busctl-/2
  -> ackctl-/2
  -> busctl+/2

endmodule
```

Figure 4.35: Example *alloc-outbound*

```
module atod (da, la, za, dr, lr, zr);

inputs la = za = 0, da = 1;
outputs lr = zr = 0, dr = 1;

lr- and da+ -> token -> zr+
la+ and da+ -> token -> dr-

(zr+ -> za+ -> zr- -> za-)
    and (dr- -> da-)
  -> lr+
  -> la+
  -> (lr- -> la-) and (dr+ -> da+)

endmodule
```

Figure 4.36: Example *atod*

```
module mp_forward_pkt
  (ackout, req, ackpb,
   allocoutbound, rts, allocpb, ack);

inputs ackout = req = ackpb = 0;
outputs allocoutbound = 1,
        rts = allocpb = ack = 0;

ackout+
  -> allocoutbound- and rts+
  -> req+ and ackout-
  -> allocpb+ and rts-
  -> ackpb+
  -> ack+ and allocpb-
  -> req- and ackpb-
  -> ack- and allocoutbound+
  -> token -> ackout+

endmodule
```

Figure 4.37: Example *mp-forward-pkt*

```
module isend (a,b,c,d,x,y,z);

inputs a=0, b=c=d=1;
outputs y=0, x=z=1;

x+/70 and z+/70 -> token
  -> a+/01 -> z-/01
  -> a-/12 -> y+/12
  -> (b-/23 and c-/23 -> y-/23)
     or (b-/28 and d-/28 -> y-/28)

// State 3 branch
y-/23 -> c+/34 -> (x-/34 and y+/34)

// At state 4
((x-/34 and y+/34)
    or (x-/54 and y+/54))
  -> ((b+/45 and c-/45)
      -> (x+/45 and y-/45)
      -> (b-/54 and c+/54)
      -> (x-/54 and y+/54))
     or
     ((b+/46 and d-/46)
      -> (x+/46 and y-/46)
      -> (b-/67 and d+/67)
      -> x-/67)

// State 8 branch
y-/28 -> d+/87 -> x-/87

// State 7, recombine
(x-/67 or x-/87)
   -> b+/70
   -> (x+/70 and z+/70)

endmodule
```

Figure 4.38: Example *isend*

```
module master_read
  (ari, pri, bprn,       xack, di, pack,
   aro, pro, busyo, breq, mrdc, do, pdo);

inputs  ari=pri=di=0,
        xack=bprn=pack=1;
outputs aro=do=0,
        pro=breq=busyo=mrdc=pdo=1;

aro+ -> ari+
ari+ -> aro- and pro-
aro- -> ari-
ari- -> token -> aro+

pri- -> (token -> pro-)
        and (token -> aro+)
        and breq-
pro- -> pri+
pri+ -> pro+
pro+ -> pri-

breq- -> bprn-
bprn- -> busyo-
busyo- -> breq+ and mrdc-
breq+ -> bprn+ and busyo+
bprn+ -> token -> breq-
busyo+ -> token -> breq-

mrdc- -> xack-
xack- -> mrdc+ and do+
mrdc+ -> xack+ and breq+
xack+ -> token -> mrdc-

do+ -> di+
di+ -> do- and mrdc+ and pdo-
do- -> di-
di- -> token -> do+

pdo- -> pack-
pack- -> pdo+
pdo+ -> pack+
pack+ -> token -> pdo- and aro+

endmodule
```

Figure 4.39: Example *master-read*

```
module nak_pa
  (rejsend,ackbus,ackhyst,busack,
   ack,reqbus,hystreq,busreq,enableda);

inputs rejsend = ackbus = ackhyst =
       busack = 0;
outputs ack = reqbus = hystreq =
        busreq = enableda = 0;

rejsend+ -> reqbus+ -> ackbus+
  -> hystreq+ and enableda+
  -> ackhyst+ -> busreq+ -> busack+
  -> busreq- -> busack-
  -> (reqbus- -> ackbus-)
       and
     (ack+ -> rejsend-)
       and
     ((enableda- and hystreq-)
       -> ackhyst-)
  -> ack- -> token -> rejsend+

endmodule
```

Figure 4.40: Example *nak-pa*

```
module nowick (a,b,c,x,y);

inputs a = b = c = 0;
outputs x = y = 0;

cycle (
  a+ and b+ -> x+/1 and y+/1
    -> c+
    -> x-/1
    -> c-
    -> x+/2 and y-/1
    -> b-
    -> x-/2 and y+/2
    -> a-
    -> y-/2 -> token
)

endmodule
```

Figure 4.41: Example *nowick*

```
module pe_send_ifc (reqsend, adbldout, rdiq, treq, ackpkt, adbld, peack, tack);

inputs reqsend = adbldout = rdiq = treq = ackpkt = 0;
outputs adbld = peack = tack = 0;

reqsend-/90 -> token
  -> (reqsend+/01 and treq+/01 and rdiq+/01)
  -> adbld+/01

adbld+/01 or adbld+/61
  -> adbldout+/12
  -> peack+/12
  -> rdiq-/23
  -> (peack-/23 and tack+/23 and adbld-/23)
  -> (adbldout-/34 and treq-/34 and rdiq+/34 -> adbld+/34)
       or
     (adbldout-/38 and treq-/38 and ackpkt+/38 -> peack+/38)

adbld+/34
  -> adbldout+/45
  -> peack+/45
  -> rdiq-/56
  -> (peack-/56 and tack-/56 and adbld-/56)
  -> (adbldout-/61 and treq+/61 and rdiq+/61 -> adbld+/61)
       or
     (adbldout-/67 and treq+/67 and ackpkt+/67 -> peack+/67 and tack+/67)

(peack+/67 and tack+/67)
  -> (ackpkt-/79 and treq-/79)
  -> (peack-/79 and tack-/79)

peack+/38 -> ackpkt-/89
  -> (tack-/89 and peack-/89)

tack-/109 or (peack-/79 and tack-/79) or (tack-/89 and peack-/89)
  -> (treq+/910 -> tack+/910 -> treq-/109 -> tack-/109) or reqsend-/90

endmodule
```

Figure 4.42: Example *pe-send-ifc*

```
module ram_read_sbuf
  (req,precharged,prnotin,wenin,
   wsldin,ack,wsen,prnot,wen,wsld);

inputs  req = precharged = 1,
        prnotin = wenin = wsldin = 0;
outputs ack = prnot = wen = wsld = 0,
        wsen = 1;

req+ and precharged+
  -> token -> prnot+
  -> prnotin+
  -> wen+
  -> precharged- and wenin+
  -> ack+
  -> req-
  -> wen- and wsen-
  -> wenin-
  -> (wsld+ -> wsldin+)
       and
     (prnot- -> prnotin-)
  -> wsld-
  -> wsldin-
  -> ack- and wsen+
  -> req+

wsld+ and prnot- -> precharged+

endmodule
```

Figure 4.43: Example *ram-read-sbuf*

```
module rcv_setup
  (reqrcv, sending, acksend,
   enwoq, rejsend);

inputs reqrcv = sending = acksend = 0;
outputs enwoq = rejsend = 0;

reqrcv+/1 -> enwoq+/1
  -> reqrcv-/1
  -> enwoq-/1

// Two choice places
sending+ -> (sending-/1 or reqrcv+/2)

(enwoq-/1 or sending-/1 or enwoq-/2)
  -> token -> (reqrcv+/1 or sending+)

reqrcv+/2
  -> rejsend+
  -> sending- and acksend+
  -> rejsend-
  -> acksend-
  -> enwoq+/2
  -> reqrcv-/2
  -> enwoq-/2

endmodule
```

Figure 4.44: Example *rcv-setup*

```
module rcv_setup_better
  (reqrcv, sending, acksend,
   enwoq, rejsend);

inputs reqrcv = sending = acksend = 0;
outputs enwoq = rejsend = 0;

cycle (token
  -> sending*
  -> reqrcv+
  -> if (sending) then
       rejsend+
         -> acksend+ and sending-/2
         -> rejsend-
         -> acksend-
     endif
  -> enwoq+
  -> reqrcv-
  -> enwoq-
)

endmodule
```

Figure 4.45: Improved version of *rcv-setup*, using `sending*` construct

```
// Tam-Anh Chu Thesis Page 172

module rlm (Cr, Sr, La, Ca, Sa, Lr);

inputs Cr = Sr = La = 0;
outputs Ca = Sa = Lr = 0;

La+ or Sr+/2
  -> Sa+ -> Sr- -> Sa-
  -> Sr+/2 or Cr+

Cr+ -> Lr- -> La- -> Ca+
  -> Cr- -> Ca- -> token
  -> Sr+/1 -> Lr+ -> La+

endmodule
```

Figure 4.46: Example *rlm*

```
module sbuf_ram_write
  (req, precharged, done, wenin, wsldin,
   ack, prbar, wsen, wen, wsld);

inputs req=precharged=wenin=wsldin=0,
       done = 1;
outputs ack = wen = wsld = prbar = 0,
        wsen = 1;

prbar+
  -> precharged-
  -> wen+
  -> done+ and wenin+
  -> (wen- and wsen- -> wenin-
                     -> wsld+
                     -> wsldin+
                     -> wsld-
                     -> wsldin-)
     and (ack+ -> req-)
  -> (wsen+ -> token -> done-)
     and
     (prbar- -> token -> precharged+)
     and (ack- -> token -> req+)
  -> prbar+

endmodule
```

Figure 4.47: Example *sbuf-ram-write*

```
module sbuf_read_ctl
  (req, ackread, busack,
   ack, ramrdsbuf, busreq);

inputs req = ackread = busack = 0;
outputs ack = ramrdsbuf = busreq = 0;

req+ and (ackread- -> token)
  -> ramrdsbuf+
  -> ackread+
  -> busreq+
  -> busack+
  -> busreq-
  -> busack-
  -> ramrdsbuf- and ack+
  -> req-
  -> ack-
  -> ackread- and (token -> req+)

endmodule
```

Figure 4.48: Example *sbuf-read-ctl*

4.4 Translation to a Petri net

This section describes the translation from the specification language to the intermediate Petri net form. To make the simulation of the net easier, no Boolean labels are allowed on arcs, such as in input choice STGs, and the only transitions allowed are rising and falling transitions of named signals and dummy transitions. This means that `d*` must be translated into a structure containing `d+` and `d-`. The simulation is not made easier if the net is pure, so self-loops are allowed. It will be assumed that the size of the resulting Petri net is not an issue, so the translation can be very inefficient. The net will be optimized to reduce the number of transitions and places before simulation.

4.4.1 True/false places

When creating the intermediate Petri net, a pair of places is created for each signal that guarantees that rising and falling transitions alternate. These places were shown to be unnecessary for the latch controller example back in Figure 4.6, but there are three reasons why these places should be included for all signals in all specifications:

- True/false places may be needed to translate `if ... then` statements—see Section 4.4.4.
- These places induce a mapping from Petri net markings onto the state graph of the Petri net, which makes the simulation stage easier. The state of all signals does not have to be stored during simulation, because it can be simply read off from the x-true places for all signals x.
- These places stop certain kinds of misbehaviour. If the specification contains a nondeterministic choice between firing `a+` and `b+`, and `a` is already at logic 1, the choice must be to fire `b+`; without places to record the state of `a`, the net could not make this decision.

When the specification is first read in, true and false places are created for all signals x mentioned, as well as x', x'', $x\hat{}$ and $x\hat{}'$ if required, and arcs to and from these places included in the Petri net. If several rising transitions of a single signal are used in the input file, then arcs must be included to each of these transitions, and the same for multiple falling transitions. For example, in the loadable counter example, there would be arcs from the `aro`-false place to each of `aro+/1`, `aro+/2` and `aro+/3`, and from each of those transitions to the `aro`-true place.

4.4.2 Transitions

The intermediate Petri net describes the behaviour of the circuit when used in the improved model of interconnections shown back in Figure 4.24, so when $p \longrightarrow q$ is written, p may need to be changed to be p' or p'' depending on the types of p and q. Possible signal types are I, O, N and X, short for Input, Output, iNternal and eXternal respectively[3]. The sixteen combinations of the types of p and q are shown in Table 4.1.

If the only operator used in a specification was the arrow operator, then the translation into a Petri net could be carried out by taking each pair of transitions p and q with an arrow between them in the specification file, looking up p and q in Table 4.1, and creating an arc in the intermediate Petri net from one of p, p' or p'' to

[3] N for internal follows the convention in [19]

Transition p		Transition q is of type			
is of type		Input	Output	Internal	External
Input	I	$p'' \longrightarrow q$	$p' \longrightarrow q$	$p' \longrightarrow q$	$p'' \longrightarrow q$
Output	O	$p' \longrightarrow q$	$p \longrightarrow q$	$p \longrightarrow q$	$p' \longrightarrow q$
Internal	N	Error	$p \longrightarrow q$	$p \longrightarrow q$	Error
External	X	$p \longrightarrow q$	Error	Error	$p \longrightarrow q$

Table 4.1: Meaning of $p \longrightarrow q$ for different types of p and q

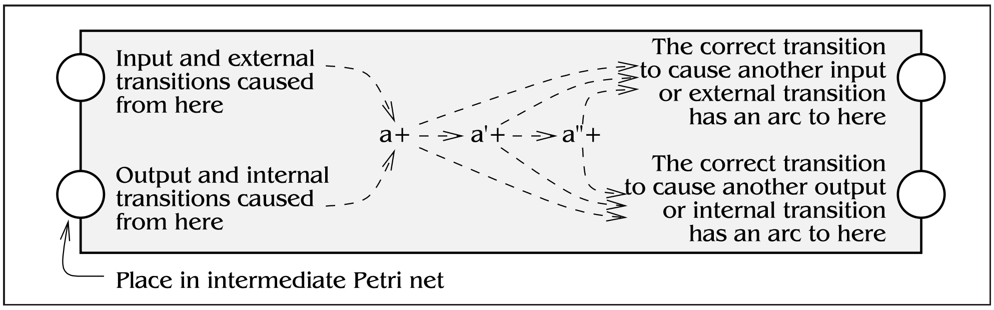

Input and external transitions caused from here
Output and internal transitions caused from here
a+
a'+
a''+
The correct transition to cause another input or external transition has an arc to here
The correct transition to cause another output or internal transition has an arc to here
Place in intermediate Petri net

Figure 4.49: Representation in the Petri net of a transition in the specification file

q as appropriate. However, it is not clear whether this approach can be generalized to a language that also includes `and`, `or` and `if` statements. Consider:

`a+` ⟶ `if (b) then (c+ and d+) endif` ⟶ `(e+ or f+)`

In order to work out whether `a+` should actually be `a+`, `a'+` or `a''+`, the translation program needs to know the type of the transition caused by `a+`. Here, `a+` can cause either `c+` and `d+` if `b+` is true, or one of `e+` or `f+` if `b` is false. It could be declared that in cases like this, all transitions caused by `a+` must be the same type, but that would disallow `a+` ⟶ `(b+ and c+)` if `b+` and `c+` are different types, which is clearly allowable, because this could be written as `a+` ⟶ `b+` and `a+` ⟶ `c+`.

To find a solution to this problem, it is necessary to notice two facts:

- In Table 4.1, the columns for types I and X are identical, as are O and N. Therefore in $p \longrightarrow q$, it is only necessary to know whether q is of type I or X, or of type O or N.

- A single transition in the specification file does not have to be translated into just one transition in the intermediate Petri net; any reasonably small combination of transitions and places would be acceptable.

Given this information, a possible solution is shown in Figures 4.49 and 4.50. A transition in the specification file is translated into a small net fragment, a *transition block*, which has two input places and two output places. Places between two transitions are not shown in Figure 4.50 for clarity. To translate a statement `p+`⟶`q+`, the transition blocks for `p+` and `q+` are abutted, overlapping the output places of `p+` with the input places of `q+`, as shown in Figure 4.51. Any hanging structures—

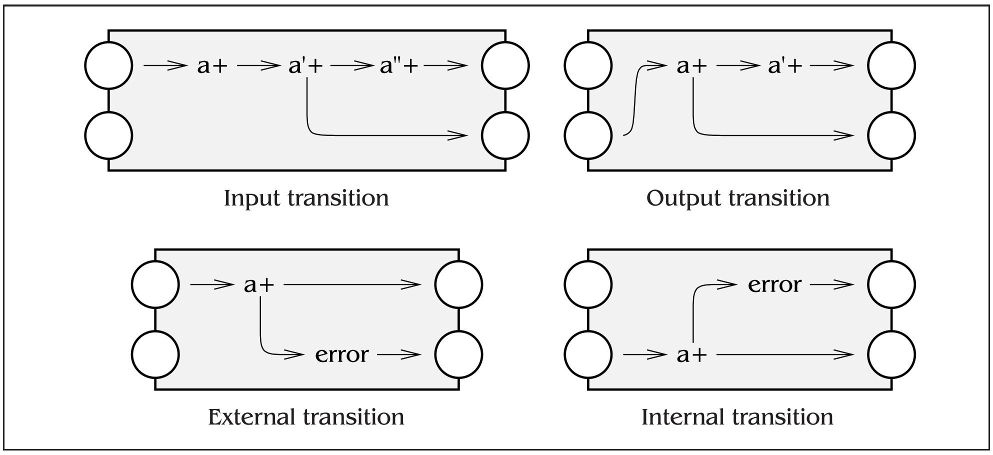


Figure 4.50: Representation of input, output, external and internal transitions

places or transitions with either no predecessors or no successors—created by this procedure should be removed before the net is simulated to create a blue diagram.

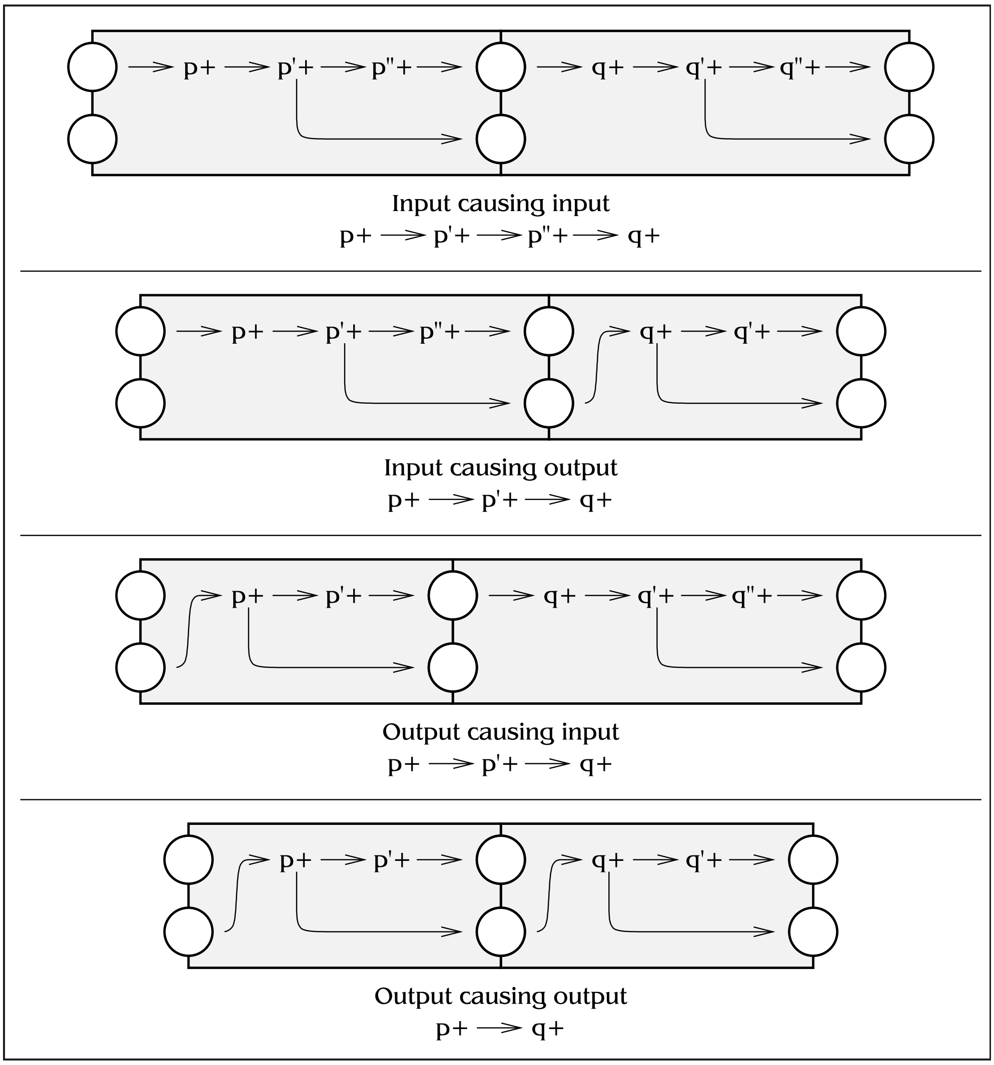
The word "error" in Figure 4.50 denotes a special error transition. If an internal transition *p* causes *q*, where *q* is an output or internal transition, then the error transition will have no successors, and so will be removed just before simulation. If *q* was an input or external signal, then the error transition will have both successors and predecessors, so will not be removed. L2b searches for error transitions that are still present just before the net is simulated, and prints an appropriate message if one is found. In this case, a message "Internal transition *p* was used to cause an input or external signal" will be printed by L2b and the translation stopped.

The structures in Figure 4.50 take care of the DI model of interconnections at a low level. It is now possible to define new operators to compose transitions, such as `and`, `or` and `if`, without having to pay too much attention to details of the underlying delay model.

4.4.3 And and Or operators

It is easy find a Petri net structure which can be used to create the `and` operation on two transitions, shown in Figure 4.52. Figure 4.53 shows a similar structure for the `or` keyword; this must include an error transition, because there is a danger that a nondeterministic specification will result from writing `a+` → (`b+ or c+`) when either of `b+` or `c+` are output or internal transitions.

A problem with the `or` statement can be seen by looking at the specification in Figure 4.54. The intended behaviour is clear: the circuit should wait for one of `a+` or `b+`, raise `x`, wait for the other input to rise, raise `y`, then wait for both inputs to fall before dropping `x` and `y` together. This behaviour is given by the Petri net on the left of Figure 4.55. What actually happens is immediate deadlock, because the intermediate Petri net created by using this naive version of the `or` statement is actually the one on the right of Figure 4.55. The two occurrences of each transition are assumed to be the same, an action which is essential when considering STG



Input causing input
p+ → p'+ → p''+ → q+

Input causing output
p+ → p'+ → q+

Output causing input
p+ → p'+ → q+

Output causing output
p+ → q+

Figure 4.51: Composition of transitions in the intermediate Petri net

fragments as in the Furber/Day latch controller. Here, they are *not* the same; the token that enters the `or` statement will only take one of the two branches, so a transition in one branch should be distinct from the same transition in the other branch. The same problem will also occur with the two alternative paths through `if`...`then` and `arbitrate` statements; collectively these will all be referred to as *branching* statements.

An easy solution to this problem would be to label each transition in a branching statement with a suffix indicating which fork it was in, for example splitting `x+` into `x+/left` and `x+/right`. If `x+` was then mentioned outside of the branching statement, say by adding an input `c` and a fragment `c+` → `x+` to the example in Figure 4.54, then it is clear that this means that `c+` must occur before *either* of the `x+` transitions `x+/left` and `x+/right`. This can be implemented in the Petri net by

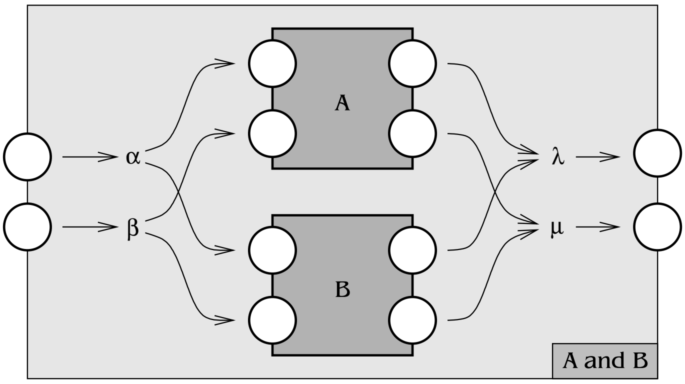


Figure 4.52: Composition of transitions using the `and` keyword

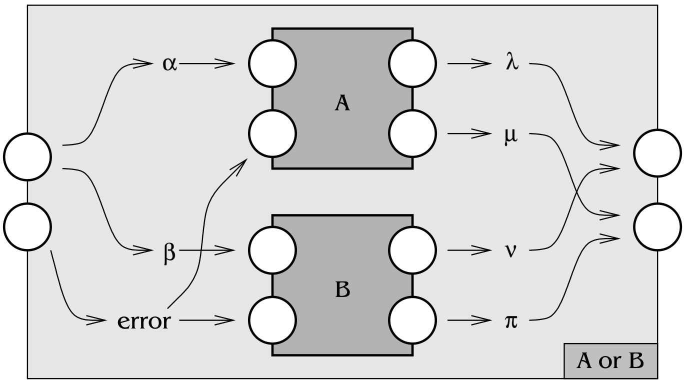
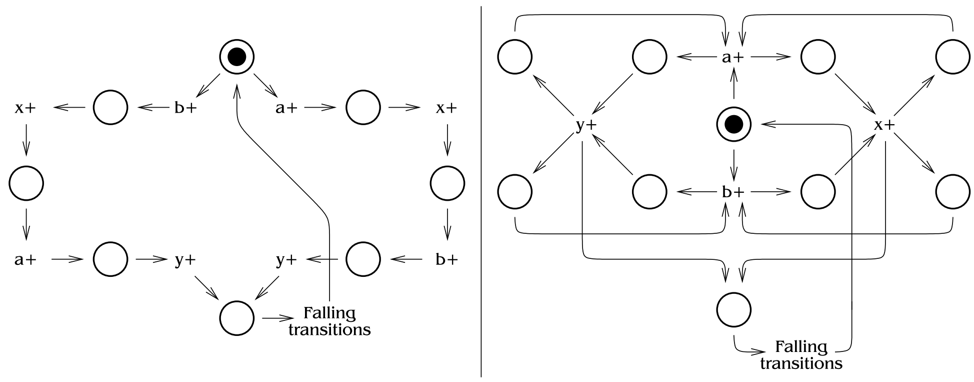


Figure 4.53: Composition of transitions using the `or` keyword

```
inputs a = b = 0; outputs x = y = 0;
cycle (token -> (a+ ⟶ x+ ⟶ b+ ⟶ y+)
                   or
                (b+ ⟶ x+ ⟶ a+ ⟶ y+)
    ⟶ (a- and b-)
    ⟶ (x- and y-)
)
```

Figure 4.54: A specification showing a problem with direct translation of the `or` keyword



Left, translation as intended, resulting in correct behaviour. Right, the translation that actually occurs, because two occurrences of a transition with the same name are taken to be the same transition

Figure 4.55: Possible translations of Figure 4.54

```
a+ → if (b) then  ⋯ → t1+ → x+ → ⋯
     else  ⋯ → t2+ → x+ → ⋯
     endif
c+ → if (d) then  if (e) then  ⋯ → t3+ → x+ → ⋯
                  else  ⋯ → t4+ → x+ → ⋯
                  endif
       ⋯ → t5+ → x+ → ⋯
     else  ⋯ → t6+ → x+ → ⋯
     endif
```

Figure 4.56: An example specification with nested `if ... then` statements

an arrow from `c+` to a single place, which has arrows to both the `x+` transitions.

However, in arbitrarily nested branching statements, it may be difficult to create a Petri net structure that causes the correct behaviour. Consider which of the transitions `t`*n*`+` cause `x+` in Figure 4.56. This example is quite contrived, but is similar enough to the specification for Martin's DME element that it would not be surprising if it actually occurred. Because `x+` is mentioned in both parts of both `if` statements, `x+` can only occur after an `a+` and a `c+`, and not just one. Looking at the first `if` statement, `x+` will be caused by either a `t1+` or a `t2+`, but not both. If `d` is true, `x+` must be caused by one of `t3+` or `t4+`, and by `t5+`; if `d` is false, it will be caused by `t6+`. To summarize, `x+` can be caused by one of:

`t1+`, `t3+` and `t5+`
`t1+`, `t4+` and `t5+`
`t1+` and `t6+`
`t2+`, `t3+` and `t5+`
`t2+`, `t4+` and `t5+`
`t2+` and `t6+`

A solution is needed for a single branching statement that can be recursively applied to nested sets of statements. A Petri net structure that accomplishes this goal is shown in Figure 4.57, which will be called a *gateway*. It has the property

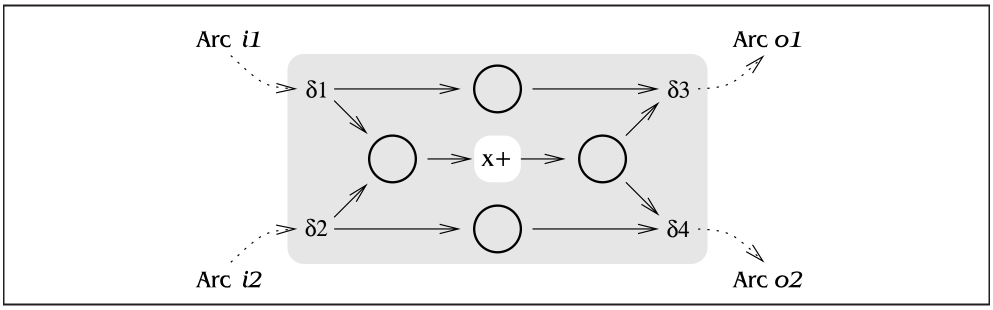
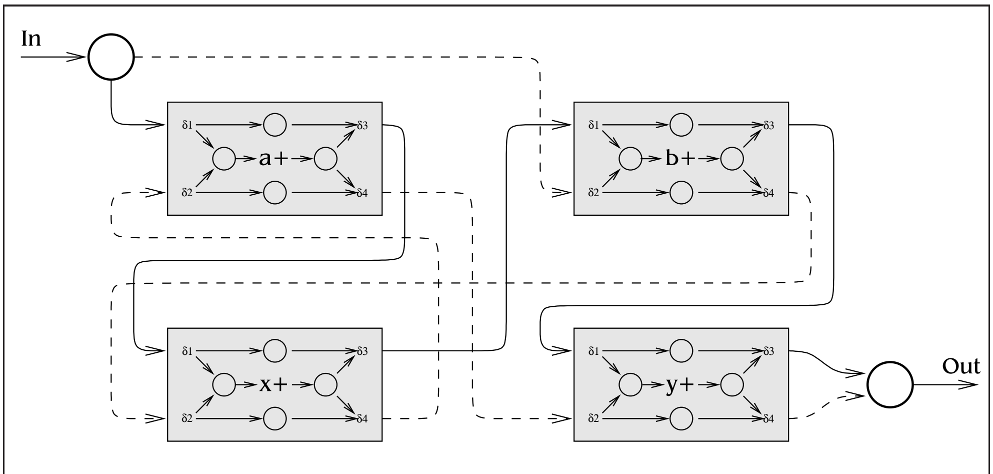


Figure 4.57: A *gateway* structure



Translation for `(a+ ⟶ x+ ⟶ b+ ⟶ y+) or (b+ ⟶ x+ ⟶ a+ ⟶ y+)`
Solid arcs are used for the path `a+ ⟶ x+ ⟶ b+ ⟶ y+`
Dashed arcs are used for the path `b+ ⟶ x+ ⟶ a+ ⟶ y+`

Figure 4.58: Translation of the `or` statement in Figure 4.54 using gateways

that `x+` can be caused by a token arriving on either arc *i1* or *i2*, but if the token arrives on arc *i1* it must leave on *o1*, and similarly for *i2* and *o2*. This can be used to keep track of which branch of an `or` statement a token is in, at the same time as allowing either branch to cause a particular transition. A translation of the `or` statement in Figure 4.54 using gateways is shown in Figure 4.58; it can be seen that this net captures the intuitive meaning of the `or` statement in this example.

Gateways are notionally used by the following procedure:

- When a branching statement is encountered, then for each transition x that is mentioned anywhere in the specification, create four places and four transitions as in Figure 4.57.

- Whenever transition x is used in the first fork of the branching statement, direct all arcs that should be to x to $\delta 1$ instead, and all arcs from x should now come from $\delta 2$. This includes the arcs formed by the creation of further nested gateways.

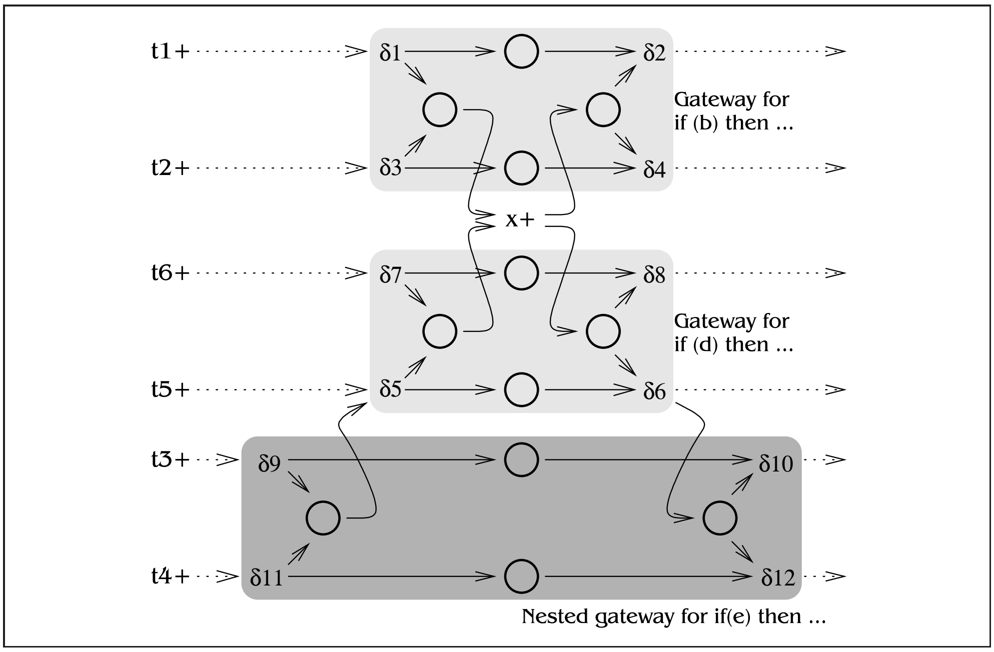


Figure 4.59: Multiple nested gateways for the example in Figure 4.56

- Arcs in the second half should be directed to $\delta 3$ and $\delta 4$ respectively.
- When leaving the branching statement, forget about the created places and transitions, but leave them in the net.
- When the translation is finished, remove transitions and places with either no successors or no predecessors.

Applying this procedure to the nested `if` example of Figure 4.56 gives the net shown in Figure 4.59. It can be seen that this gives the required behaviour. For efficiency reasons, this algorithm was implemented in a lazy way, by only creating gateways when they were actually referenced.

Gateways and transition blocks are orthogonal methods for solving their respective problems; by this I mean that either method can be built on top of the other, without either of then needed to know that the other is there. When translating the specification file, the sequence of actions of the L2b program on each transition x encountered in the file can be either of:

1. Create a gateway structure for each of x, x', possibly x''.
   Bind these gateways into a transition block.
2. Combine x, x', possibly x'' into a transition block.
   Place a gateway structure around this transition block.

There are no real reasons for preferring one to the other; the former method was used in L2b.

One insurmountable problem with `or` statements, but not with the other kinds of branching statements, is what happens when the same choice is made twice. If a specification includes fragments `x+` ⟶ (`a+ or b+`) and `y+` ⟶ (`a+ or b+`), then this creates two separate choice places, as shown in Figure 4.60. If one place chooses `a+` and the other `b+`, then the net will deadlock because each transition is waiting for both tokens to arrive. In this example, the net optimization that will be described in Section 4.5 will reduce the net in Figure 4.60 (c) to that in part (d), which works correctly, but the problem may occur in more complicated examples when optimization fails. The only solution is to reword the fragments given to be (`x+ and y+`) ⟶ (`a+ or b+`)[4]

4.4.4 The if...then statement

The `if`...`then` statement is similar to the `or` statement, but the choice between alternatives depends on the state of a signal in the circuit. It will be assumed that the environment behaviour is always fairly simple, so the `if` statement will not be used to determine whether a transition fires in the environment. Typically, the environment specification will use `or` statements where choice is required. If an `if` statement is used to conditionally fire an environment transition, L2b rejects the specification. This restriction will possibly rule out some useful behaviours, but will catch a number of incorrectly specified circuits at an early stage.

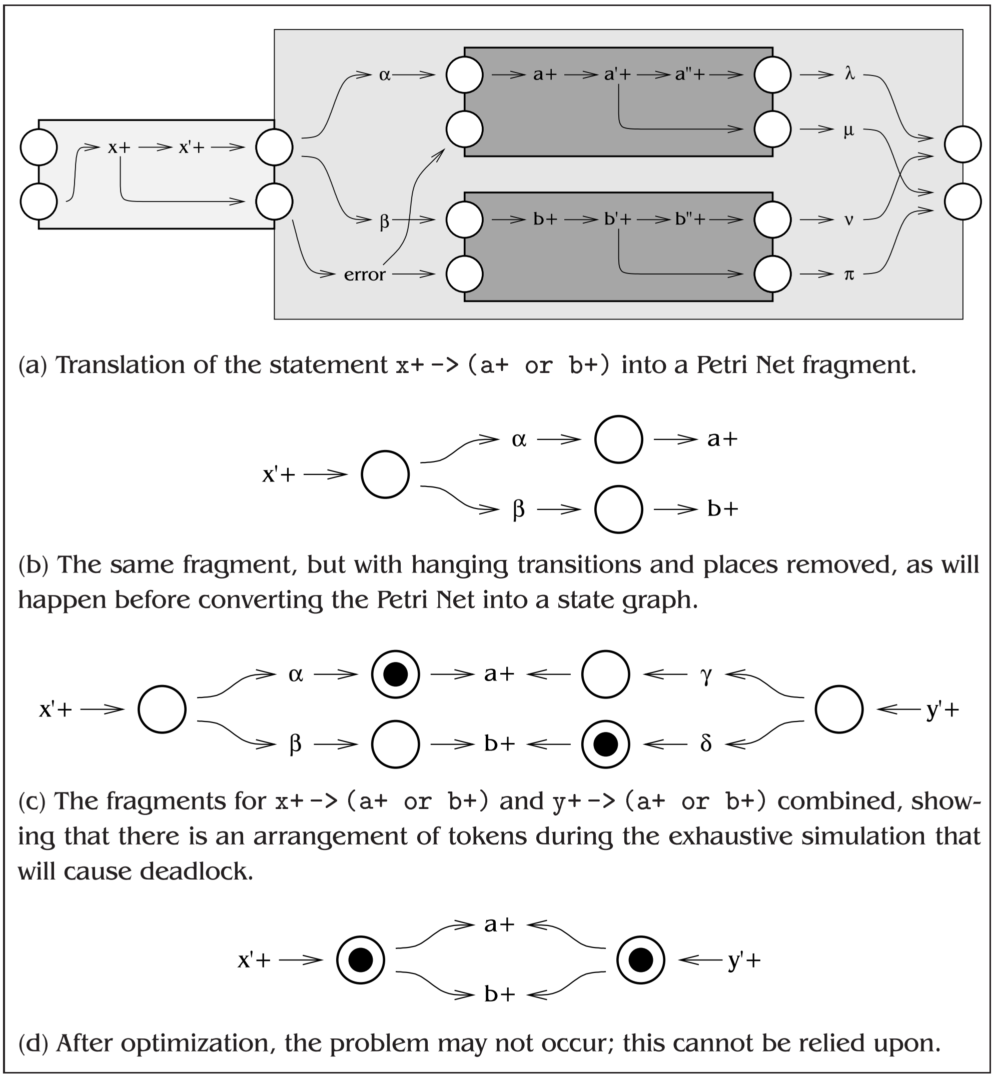
The Petri net structure for an `if`...`then` statement is shown in Figure 4.61. Places for *x true* and *x false* already exist in the net, so are not part of the `if`...`then` structure. The double-headed arc from α to the *x true* place in Figure 4.61 means that α can only fire when `x` is true, and when α does fire, it takes a token from the *x true* place and instantly replaces it. The resulting net is not pure, but that does not affect any algorithms that are being using. The *error* transition in Figure 4.61 has the same meaning as in Figure 4.53.

Figure 4.62 shows how `if (x and y)` is translated to a Petri net. This is taken to be a nested pair of `if` statements, `if (x) then if (y) then`, but with the `else` clauses merged. The use of the `or` conjunction is similar, and nested conjunctions such as `((x and y) or z)` follow by recursive application of Figure 4.62.

4.4.5 Data inputs

Data inputs such as `d*` can be translated using the structure in Figure 4.63, which is derived from the translation of a input given in Figure 4.50. Only the environment can decide when `d` is stable, so the transition following `d*` in the specification must be an input or internal transition. The *error* transition will stop the translation if this rule is violated.

[4] An early solution to this problem was to make sure that the choice places were always connected directly to the alternative transitions as in Figure 4.60 (d), rather than being connected through dummy transitions as in part (c). This was carried out by rewriting expressions that use nested `and` and `or` statements to have the `and`s at the top level. However, this method still fails when gateways exist in the intermediate net, which is pretty often, so it was abandoned.



(a) Translation of the statement `x+ -> (a+ or b+)` into a Petri Net fragment.

(b) The same fragment, but with hanging transitions and places removed, as will happen before converting the Petri Net into a state graph.

(c) The fragments for `x+ -> (a+ or b+)` and `y+ -> (a+ or b+)` combined, showing that there is an arrangement of tokens during the exhaustive simulation that will cause deadlock.

(d) After optimization, the problem may not occur; this cannot be relied upon.

Figure 4.60: Problems with multiple choice points

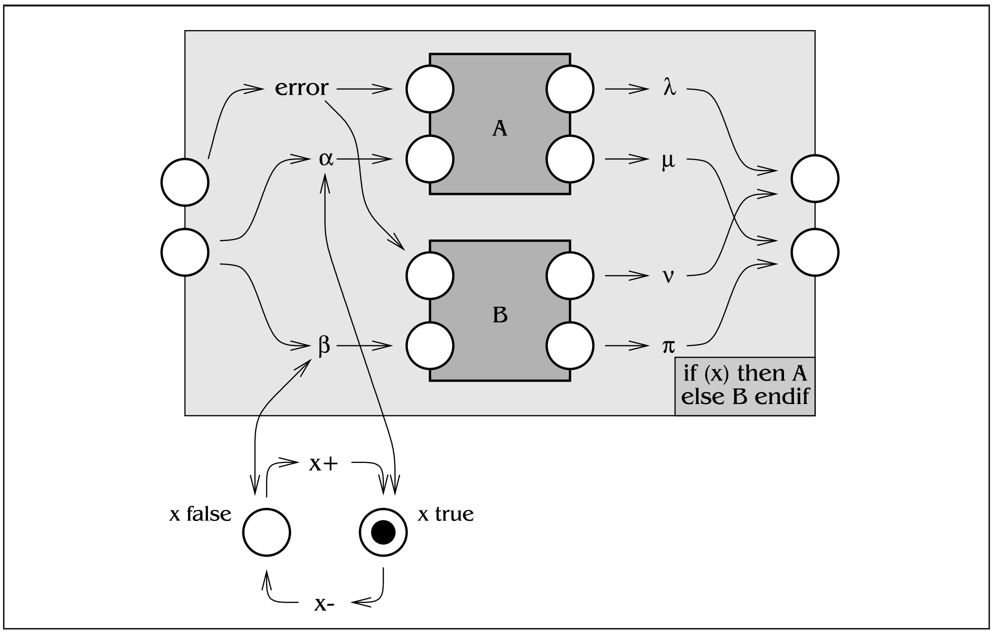


Figure 4.61: Petri net structure for the `if ... then` statement

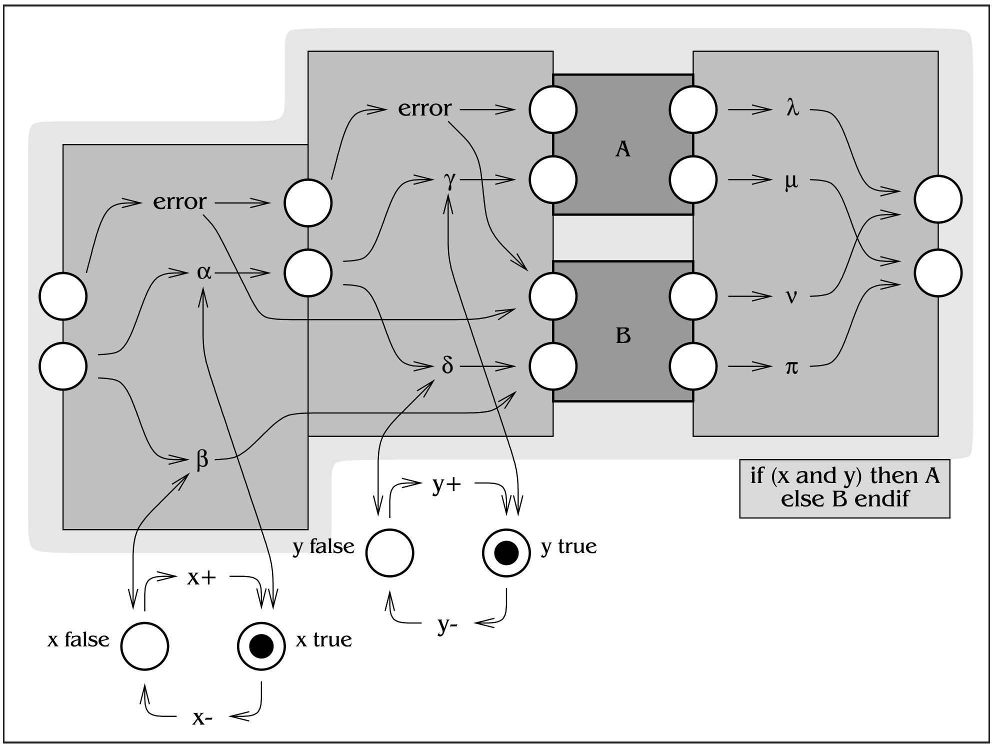


Figure 4.62: Petri net structure for an `if ... then` statement using an `and` conjunction

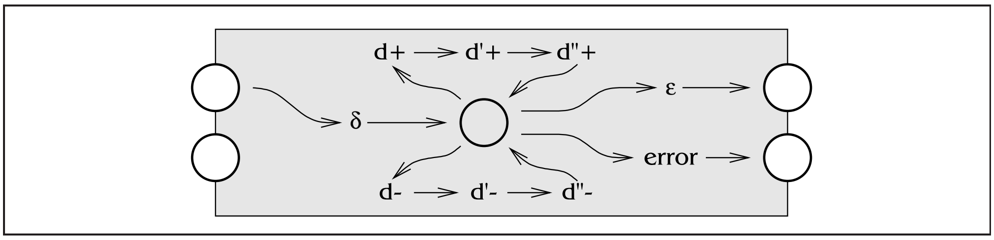


Figure 4.63: How to translate a data input into the intermediate Petri net

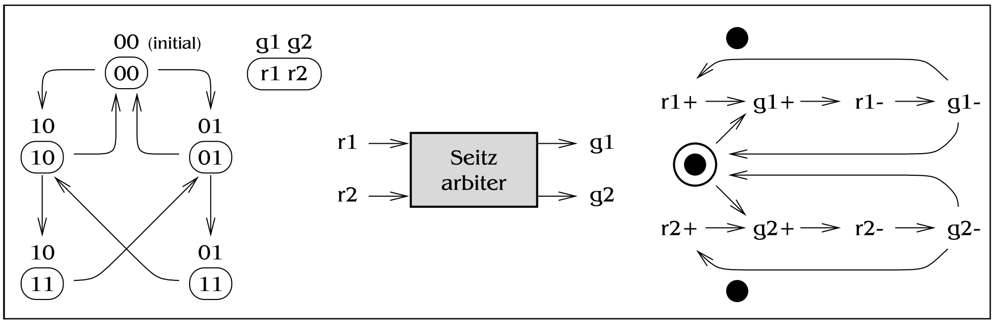


Figure 4.64: Representation of Seitz arbiter as a Blue Diagram and as a Petri Net

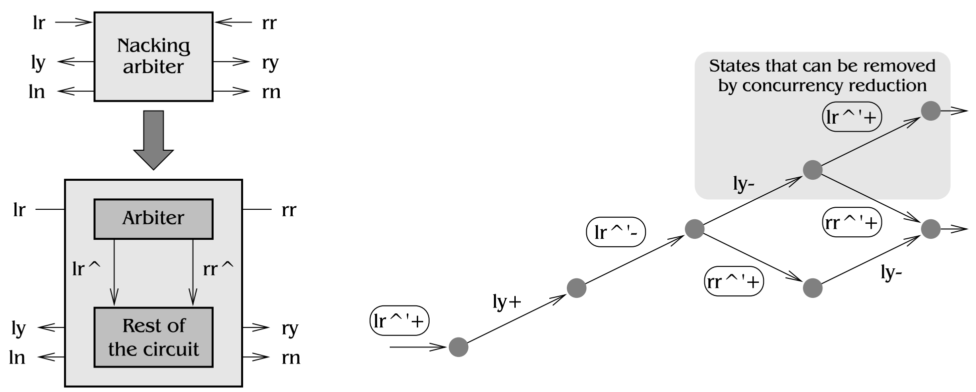
4.4.6 Arbitration

The `arbitrate` statement has three effects:

- It writes a line to the output file that causes the later synthesis tools to create a Seitz arbiter in the right place.
- It introduces new signal names, corresponding to the wires leading from the physical arbiter to the circuit to be synthesized.
- It creates structures in the intermediate Petri net that model the action of an arbiter and cause arbiter-like behaviour while the blue diagram is being created.

The first two actions are trivial, so only the third will be discussed. It is possible use the Petri net for a Seitz arbiter, shown on the right of Figure 4.64, as a model of arbiter behaviour. The concurrency reduction program prune will also need a model of the arbiter; it takes its models as blue diagrams, so a blue diagram corresponding to this Petri net is shown on the left of Figure 4.64. Although Figure 4.64 gives the correct result when used in L2b, problems will be encountered later if prune also uses this model of an arbiter. The problem can be seen by looking at a partial state graph[5] for the nacking arbiter example as shown in Figure 4.65. A valid concurrency-reducing transformation is to insist that `ly-` only happens after

[5] The prune program does not actually use state graphs, but they are used here to illustrate what goes wrong. This example is similar to the concurrency reduction approach used by Ykman-Couvreur et al.[197]



This is part of the state graph for the nacking arbiter, and shows that arbitration can be turned into alternation if the wrong model is used for the arbiter.

Figure 4.65: A problem that can occur during concurrency reduction

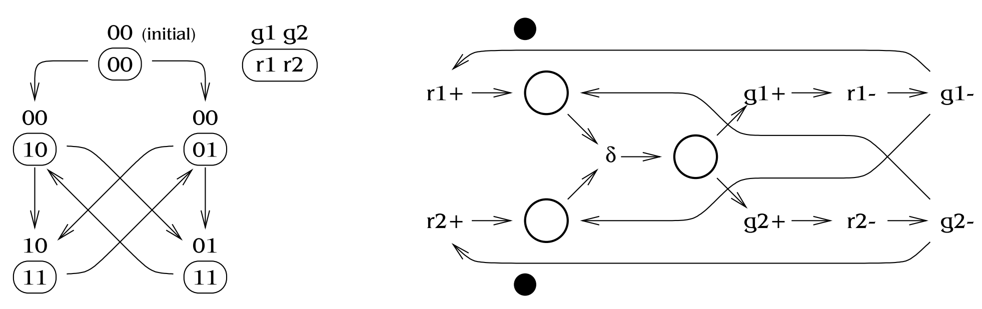


Figure 4.66: Modified arbiter behaviour, which cures a problem in prune but breaks L2b.

rr$^{\wedge\prime}$+ in this part of the state graph, which removes the shaded states and implies that, after a `lr+` and `lr-`, the nacking arbiter will only respond to a `rr+` and not to another `lr+`. This has gone half-way towards replacing the arbitration by mere alternation; after one request, only the other may happen. This is undesirable behaviour, but technically correct because the circuit will eventually respond to every input and deadlock will never occur. This is often called a violation of *fairness*.

A possible solution is to replace the arbiter module in Figure 4.65 by one that will wait until both requests are asserted before nondeterministically raising one of the grant wires, which can be represented by the BD and PN in Figure 4.66. This changes the partial state graph in Figure 4.65 to that in Figure 4.67, where there are no concurrency-reducing transformations that can favour one of `lr+` and `rr+` over the other. The nondeterministic choice in the arbiter cannot be affected any concurrency-reducing transformations, which forces prune to avoid tampering with the arbitration. This does not actually affect the circuit or how it can be used; the model is solely used inside prune to determine candidate transformations for concurrency reduction.

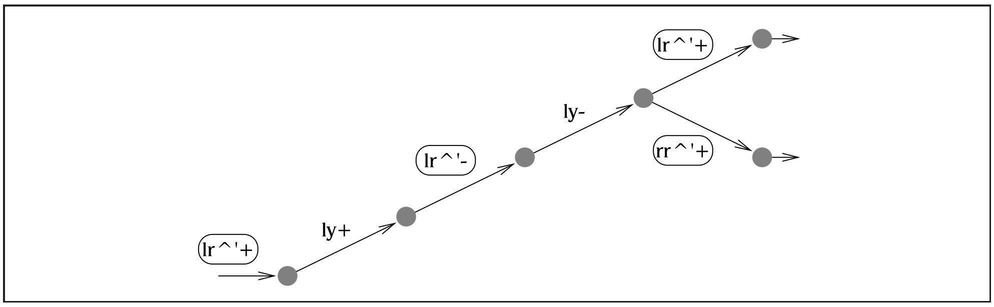


Figure 4.67: Part of the state graph for the nacking arbiter with modified arbiter behaviour

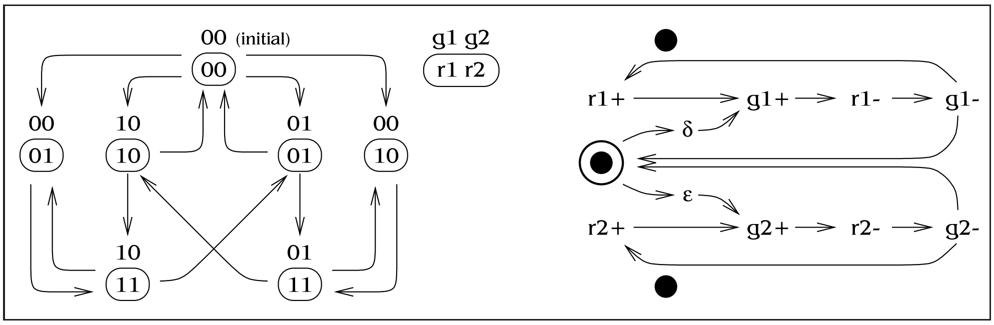


Figure 4.68: Correctly modified arbiter behaviour, which can be used in prune and L2b.

Unfortunately, the model in Figure 4.66 cannot be used in L2b as a model for the arbiter in the intermediate Petri net, because it does not exhibit certain behaviours that are required. The Petri net of the real Seitz arbiter in Figure 4.64 shows that after the sequence of transitions r1+, g1+, r2+, r1-, the arbiter gives the two transitions g1- and g2+. Because unbounded delays are assumed on wires between the arbiter and the rest of the circuit, the circuit may see both grant wires g1 and g2 being high for an unbounded time. In contrast, after that sequence of transitions in the modified arbiter of Figure 4.66, only g1- occurs, so both grant wires can never seen to be high at the same time. It was found that circuits produced using this modified arbiter model tend to rely upon both grant wires being low for several gate delays, and so fail when built using a real arbiter.

It is possible to use one model of arbiter behaviour in L2b and the other in prune, but it would seem prudent to use a single model for both situations if possible. An arbiter model is needed that will make a nondeterministic choice between grant wires, as in the modified arbiter, but display the full behaviour of the Seitz arbiter. One possible solution is shown in Figure 4.68; others also exist. The Petri net on the right of Figure 4.68 is very similar to the original Seitz arbiter, but the choice is made between two dummy transitions δ and ϵ rather than the grant wires. When this arbiter has unbounded finite delays on all inputs and outputs, it behaves equivalently to a normal Seitz arbiter, but because the choice of which side to grant cannot be affected from outside, it works properly in the prune program.

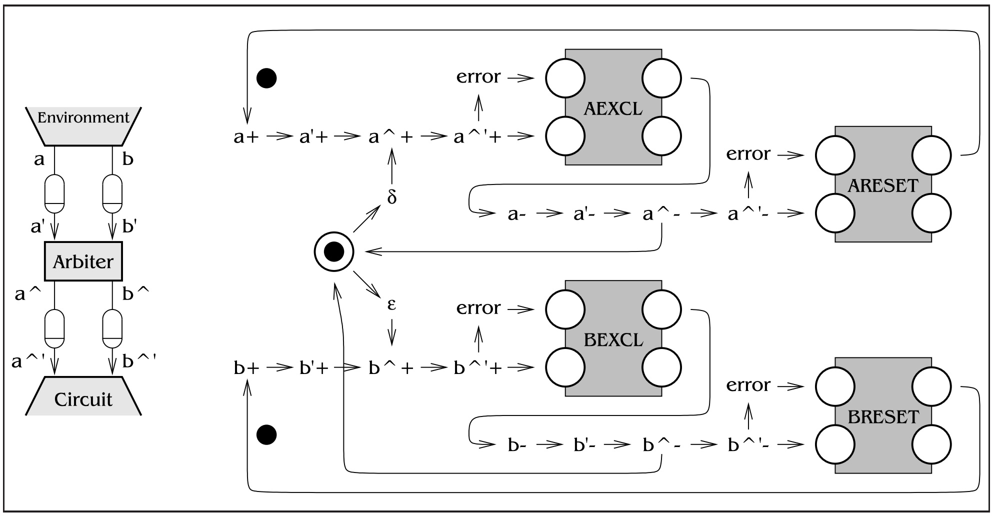


Figure 4.69: Translation of the `arbitrate` statement to a Petri net structure

The intermediate Petri net structure that should be used for the `arbitrate` statement can now be derived. To recap, the `arbitrate` statement is:

```
arbitrate
  a+ => (block AEXCL)  => a-
     => (block ARESET) => a+
| b+ => (block BEXCL)  => b-
     => (block BRESET) => b+
end
```

The transitions in Figure 4.68 must be replaced with transitions from this arbitration construct, by looking up their actual names in the improved model of module interconnections given in Figure 4.24. The request signals `r1` and `r2` will be `a′` and `b′` respectively. The grant signals `g1` and `g2` will be called `a^` and `b^`. The arc between `g1+` and `r1-` corresponds to the circuit seeing the grant (`a^′+`), the circuit's actions in response to this (AEXCL), and the environment's withdrawal of the request (`a-`). The other three grant to request arcs are similar. This gives the intermediate net structure is shown in Figure 4.69.

4.5 Converting the Petri net to a blue diagram

4.5.1 Hanging structure removal

After the intermediate Petri net is created, any places or transitions that either have no successors or no predecessors are recursively deleted. Only unnamed dummy transitions are removed, because all named transitions will always have arrows to and from their true and false places.

Two kinds of named transition should however be removed, if they only have their true and false places as successors:

- Rising and falling transitions of x'' for an input signal x. It is easier to assume that all input transitions need to be watched by the environment, and then remove the wire back if it does not get used, than it is to add the wire back when it is known whether it will be used.

- Rising and falling transitions of x' for an output signal x. This corresponds to the output wire for x not being sensed by the environment, and so means that x was actually an internal signal rather than an output. A warning message is printed in this case.

The second case happens for *master-read*, when the following is printed by L2b:

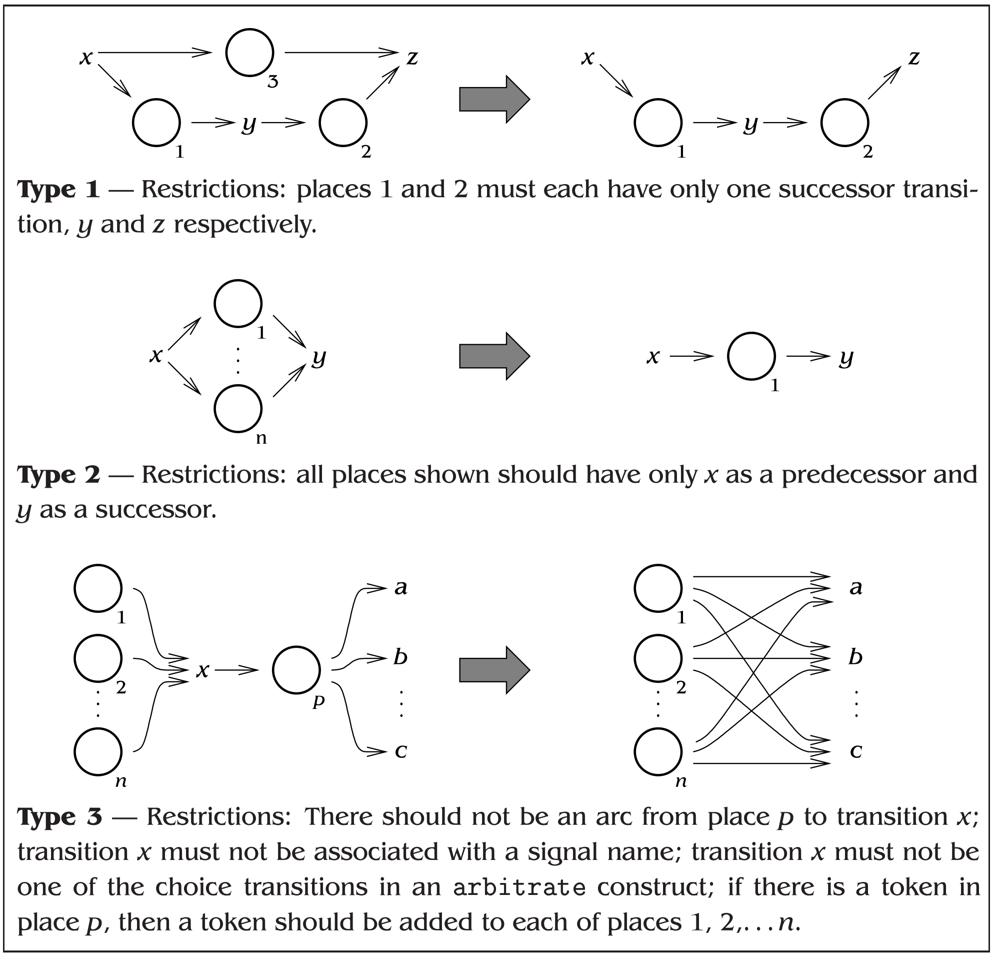
```
bash$ l2b master_read
Compiling:  module master_read
Net constructed:            172 transitions and 406 places

*** Message:  signal "busyo" does not affect the environment,
so its type has been changed from Output to Internal.

Hanging structures removed: 98 transitions and 158 places
Net optimized:              64 transitions and 124 places
Net optimized:              51 transitions and 111 places
Net optimized:              51 transitions and 110 places
Partial stategraphs created with 108 and 108 states.
Environment Blue Diagram has 108 states.
Circuit Blue Diagram has 108 states.

bash$
```

4.5.2 Net optimization

Three types of peephole optimization are used on the intermediate Petri net to reduce the number of places and transitions before simulation; they are shown in Figure 4.70. The first optimization is targeted at superfluous gateway structures, while the second removes multiple places between two transitions that are caused by, for example, writing both h/s (x,y) and x+ $\longrightarrow$ y+ in the specification file. The third rule shortens long chains of transitions and places by removing a redundant dummy transition and a following place. The third rule is the only one that is not left-right symmetrical, so a reflected version of rule 3 could also be included as an optimization. It was easier when writing L2b to reflect the entire net, apply the three rules again, and then reflect back.

4.5.3 Creating the blue diagrams

When the intermediate Petri net has been created, it must be converted into a pair of blue diagrams: one for the circuit and one from the environment. The blue diagram for the circuit can be derived by simulating the Petri net to form a state graph S, projecting this state graph onto the subset of the signals that the circuit can see, so forming a state graph S', and finally turning S' into a blue diagram by insisting that all enabled output signals fire instantaneously. This last step compacts several states in S' into a single state in the blue diagram. A similar procedure will create a blue diagram for the environment instead.



Type 1 — Restrictions: places 1 and 2 must each have only one successor transition, *y* and *z* respectively.

Type 2 — Restrictions: all places shown should have only *x* as a predecessor and *y* as a successor.

Type 3 — Restrictions: There should not be an arc from place *p* to transition *x*; transition *x* must not be associated with a signal name; transition *x* must not be one of the choice transitions in an `arbitrate` construct; if there is a token in place *p*, then a token should be added to each of places 1, 2,. . . *n*.

Figure 4.70: The three types of optimization performed on the intermediate Petri net

Producing the full state graph can be very time-consuming—the intermediate net for *master-read* has 25 610 states in its state graph—so a method that avoids generating the full state graph would be preferable. An obvious first attempt at a way to directly derive a blue diagram from the intermediate net is the following:

1. Fire any non-input transitions that are enabled in the default state.
2. Add the resultant marking to a list L of states that are still to be done.
3. Repeatedly, pop the next state *x* from list L, and for each enabled input transition *t*:
    (a) Fire *t*.
    (b) Fire all enabled non-input transitions in any order to form a state *y*.
    (c) Mark *y* as a successor of *x* in the blue diagram. Push *y* on to *L* if *y* has not been seen before.

The problem with this method is that in step 3b, the order that non-input transitions are fired in might make a difference to the resulting blue diagram if choice places are present. Instead, define *unsafe* transitions as those transitions which are either inputs to the circuit or share a predecessor place with another transition, and *safe* transitions as all the rest. Intuitively, safe transitions are those that can safely be fired as soon as they are enabled, and doing so will not change the blue diagram that results. If the above algorithm is modified by changing "non-input" to "safe" and "input" to "unsafe", then an algorithm is produced that will not be confused by choice places, but instead of giving a BD, it will give an XBD. Choice points in `if...then`, `or` and `arbitrate` statements will cause connected regions of states such as the shaded area in Figure 4.71, where several states have the same values for all inputs. These regions must be collapsed into a single state in the final blue diagram, as shown at the bottom of Figure 4.71. The shaded states do not necessarily have the same output values as each other, but in a deterministic circuit, the outputs for all states that have arcs leaving the shaded region will agree; in this case, the state in the final blue diagram has its outputs defined by this agreement. Steps have been taken to stop nondeterministic diagrams, such as limiting the use of the `or` operator to the environment only, so these outputs should not be found to disagree,[6] but if they do, `L2b` stops with an error.

By firing enabled safe transitions in one order rather than every possible order, a substantial time saving is made. The XBDs in the *master-read* example have only 108 states each, so a pair of XBDs was used instead of a 25 610-state state graph.

During the net simulation, one of three errors may be encountered:

- The net deadlocks.
- More then a preset number of tokens accumulates in a state, currently 10.
- Rising and falling transitions of a signal are given in quick succession, without either transition being detected by the intended recipient; this violates the assumption that all modules present a delay-insensitive interface, and also creates a possible static hazard.

Livelock is not checked for, but this will usually manifest itself as a violation of *n*-safety as tokens accumulate in the net.

While the net is being simulated, a record is kept of which transitions have been fired to get to a particular state. This record is used to provide an indication of where an error condition occurred. An example error, produced by removing the line "`wsld+ and prnot- -> precharged+`" from *ram-read-sbuf*, is shown below. Deleting this line removes any constraints on when `precharged+` fires, so it fires immediately after `precharged-`, giving a static hazard.

[6] although pathological specifications can still cause nondeterministic circuit behaviour, such as `...` → `(x+ and y+)` → `((if (x) then y- endif) and (if (y) then x- endif))`.

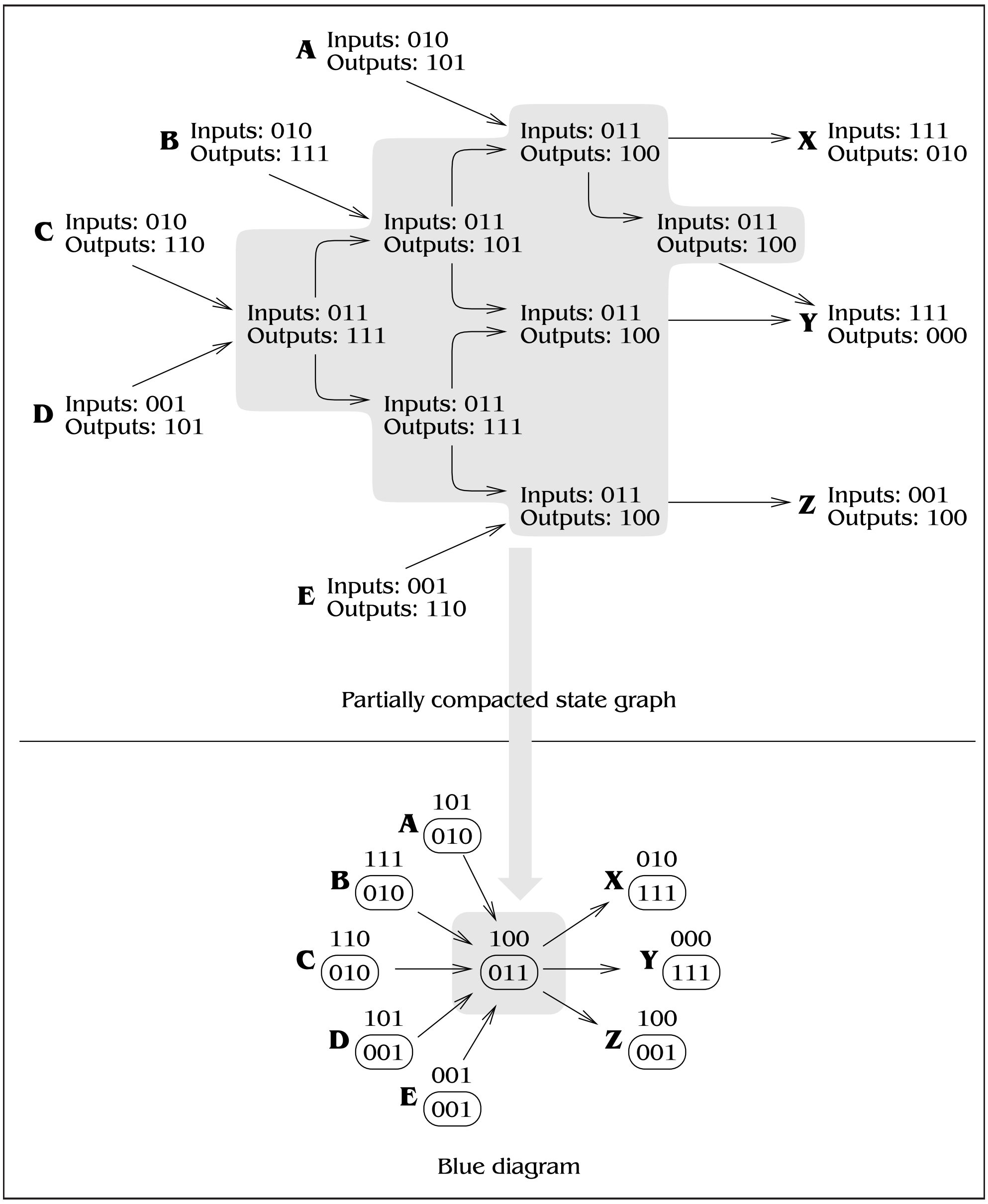


Figure 4.71: Removing redundant states from an XBD to form a blue diagram

```
bash$ l2b ram_read_sbuf
Compiling: module ram_read_sbuf
Net constructed:            102 transitions and 186 places
Hanging structures removed: 65 transitions and 109 places
Net optimised:              45 transitions and 89 places
Net optimised:              41 transitions and 85 places

Fatal error when simulating Petri Net:

Interface error in state reached by:
prnot+, prnotin+, wen+, wenin+, precharged-

After that, precharged+ can occur without waiting for any response
from the environment.  This breaks the assumption that the circuit has
a delay-insensitive interface to the environment.

h/s () declarations may help here.
It's also possible that there are too many tokens in the specification.

bash$
```

4.5.4 Reduction of the blue diagrams

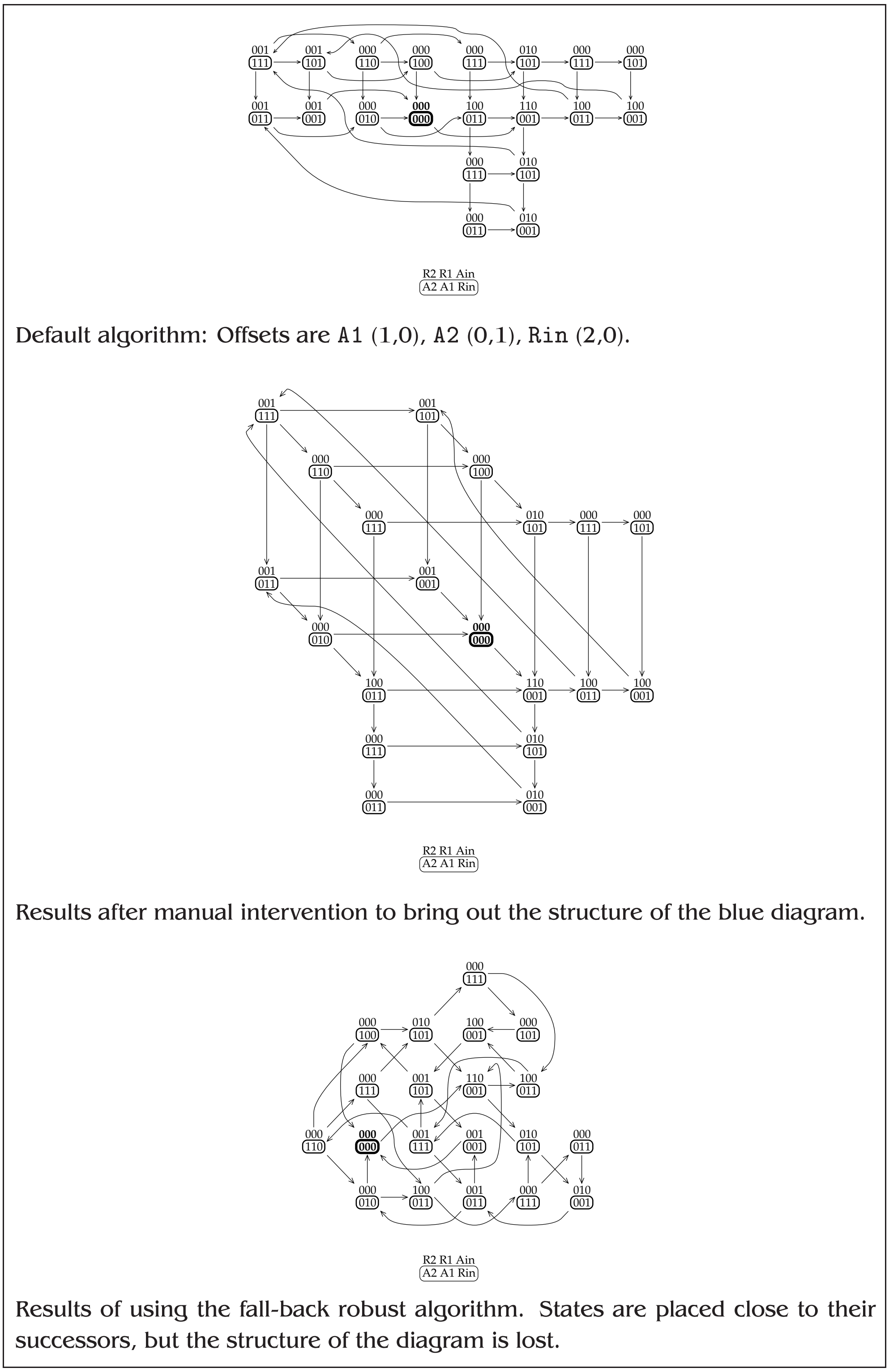
It is often the case that in the generated blue diagrams, there will be a state s with two identical successor states x and y. This occurs when the environment has made a decision between two alternatives, but the circuit has not yet seen the effects of this decision, or vice versa. In these cases, states x and y should be merged into a single state to simplify the blue diagram.

Occasionally, two identical states x and y are found that have the same successor states, and again these should be merged. In one example, *pe-send-ifc*, identical states can be found that have a successor in common, but one has successors that the other does not. Such states are only merged if the `-strong-reduce` command line option is given to l2b. The optionally merged states are tied together in Figure 4.81. This makes no difference to the generated circuit in this example.

4.6 Drawing blue diagrams

Drawing blue diagrams is tedious and error-prone, much more so than for burst-mode diagrams or STGs, so b2ps was written to convert blue diagrams into Encapsulated PostScript (`.eps`) files. The program is semi-automated; manual intervention is usually required to produce the best layout of states, although it can do a fairly good job on its own. An example of the output of b2ps is shown in Figure 4.72.

The default algorithm assigns a vector of integers (a_x, a_y) to each input signal name a. If state s is placed at position (x, y) in the diagram, and a successor state t of s which has not been seen yet differs from state s in the value of signal a, then state t is placed at position $(x + a_x, y + a_y)$. Vectors (a_x, a_y) are chosen so that states do not coincide, by choosing $(1, 0)$ and $(0, 1)$ for the most commonly changing signals, and picking progressively larger integers for other signals until no states overlap. Empty rows and columns are removed, then arrows placed between states using algorithms similar to circuit routing procedures—arrows cannot cross states, and can only cross other arrows at 45° or 90° to ensure that it is always possible to follow an arrow by eye to its destination. This algorithm created the top example in Figure 4.72, but the layout can be significantly improved by manual



Default algorithm: Offsets are A1 (1,0), A2 (0,1), Rin (2,0).

Results after manual intervention to bring out the structure of the blue diagram.

Results of using the fall-back robust algorithm. States are placed close to their successors, but the structure of the diagram is lost.

Figure 4.72: Results of b2ps on the blue diagram for the parallel component

intervention, shown in the middle example.

If the blue diagram is not semi-modular, the default algorithm will fail, and a fall-back procedure must be used. This treats all states as incompressible balls with elastic bands between them where arrows occur, then allows the system to stabilize in a high number of dimensions before the equivalent of dropping a heavy book on it to flatten it to two dimensions. States are encouraged towards grid points, and then arrows generated as above. This algorithm tends to destroy any structure in the diagram, as the example at the bottom of Figure 4.72 shows.

4.7 Results of translation

Table 4.2 shows the number of places and transitions in the intermediate Petri net for all of the examples, just after the creation of the net, after hanging structures have been removed, and after optimization of the net. The amount of time taken for the entire translation process is also listed, both with and without optimization. Net optimization takes almost no time and can significantly speed up program execution. It can be seen that the translation process was very inefficient in terms of the number of places and transitions used; even after optimization, the intermediate Petri nets were three or four times the size of the STGs that they were derived from. However, the execution times were all so small that this does not matter.

The execution times depend predominantly on three features of the specification: the number of places in the intermediate net, the number of states in the final blue diagrams, and whether there is choice behaviour in the intermediate net. Specifications that produce intermediate nets with a large number of places, such as *pe-send-ifc* and *isend*, take a long time to translate to a blue diagram, because each possible marking takes more memory to store. The *master-read* example produces a large blue diagram, which takes a long time to generate. Examples with arbitration, such as the nacking arbiter, or explicit choice, such as *pe-send-ifc*, produce several states internally that are collapsed down into a single blue diagram state, and this again increases the time taken.

This chapter concludes with the blue diagrams produced for all the examples except *master-read*, which is too big to fit comfortably on one page.

Specification	# trans *t* # places **p**	Hanging removed	Optimized	No opt. time (ms)	Opt. time (ms)
latchc	*48* **100**	*32* **63**	*24* **55**	202.6	23.7
parallel	*42* **85**	*30* **60**	*24* **54**	24.0	23.7
nacking arbiter	*298* **351**	*176* **257**	*58* **95**	1 213.11	112.8
DME	*178* **235**	*107* **168**	*47* **88**	212.1	60.8
loadable counter	*414* **522**	*201* **332**	*57* **132**	431.1	64.3
alloc-outbound	*135* **207**	*70* **105**	*36* **67**	36.6	29.5
atod	*72* **140**	*41* **68**	*26* **53**	24.9	21.8
mp-forward-pkt	*76* **140**	*49* **83**	*33* **67**	26.0	22.5
isend	*473* **604**	*214* **302**	*78* **123**	583.5	96.5
master-read	*172* **406**	*98* **158**	*51* **110**	612.6	148.7
nak-pa	*90* **164**	*41* **99**	*36* **76**	37.6	28.6
nowick	*72* **120**	*47* **71**	*30* **54**	27.4	23.3
pe-send-ifc	*607* **779**	*278* **388**	*102* **166**	3 312.8	310.0
ram-read-sbuf	*108* **202**	*67* **113**	*41* **87**	40.4	30.6
rcv-setup	*193* **262**	*91* **133**	*33* **62**	71.9	34.3
rlm	*121* **178**	*55* **85**	*26* **52**	26.7	23.5
sbuf-ram-write	*106* **190**	*68* **113**	*42* **87**	48.7	33.1
sbuf-read-ctl	*66* **120**	*40* **66**	*24* **50**	21.1	20.2

Table 4.2: Results of reduction and optimization

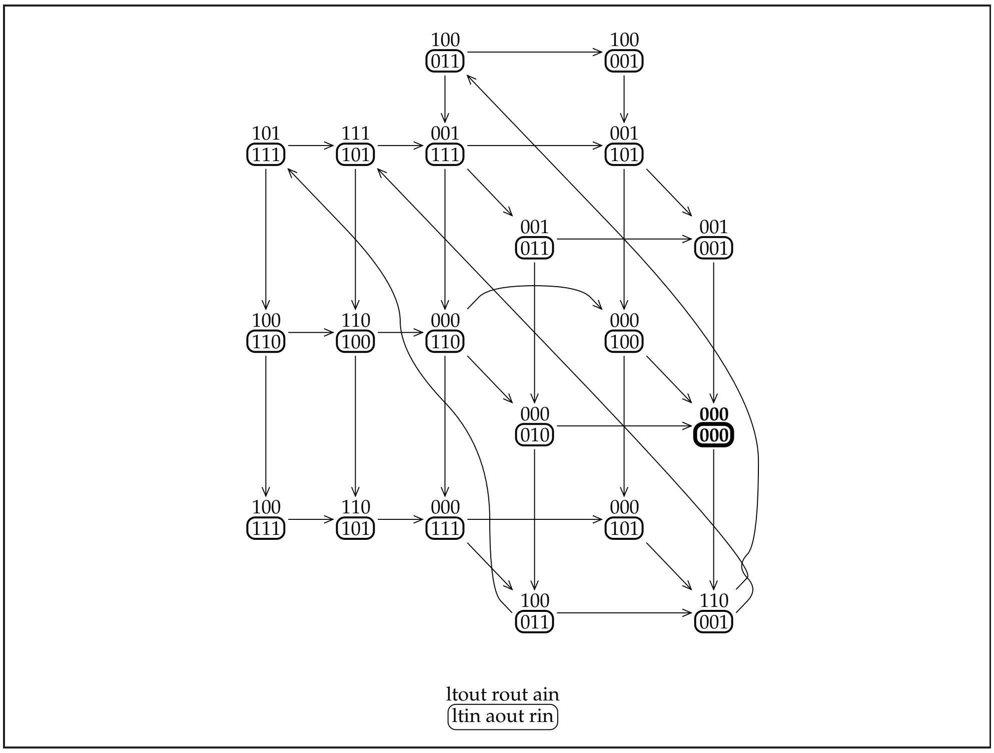


Figure 4.73: Blue diagram for latch controller example

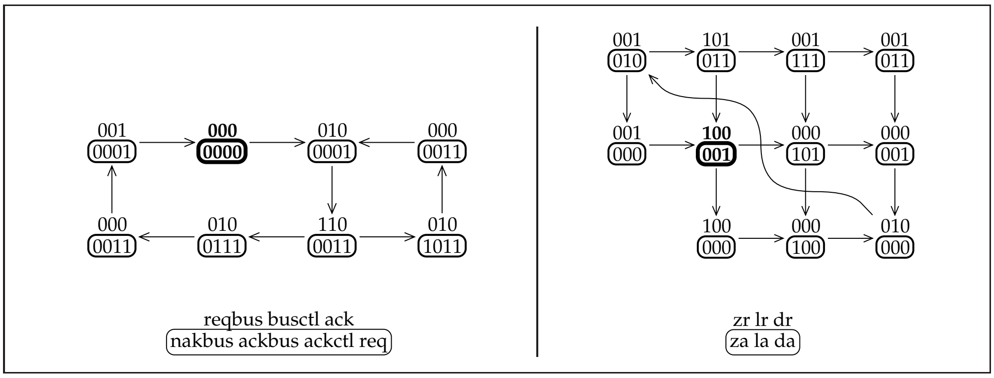


Figure 4.74: Blue diagrams for *alloc-outbound* and *atod*

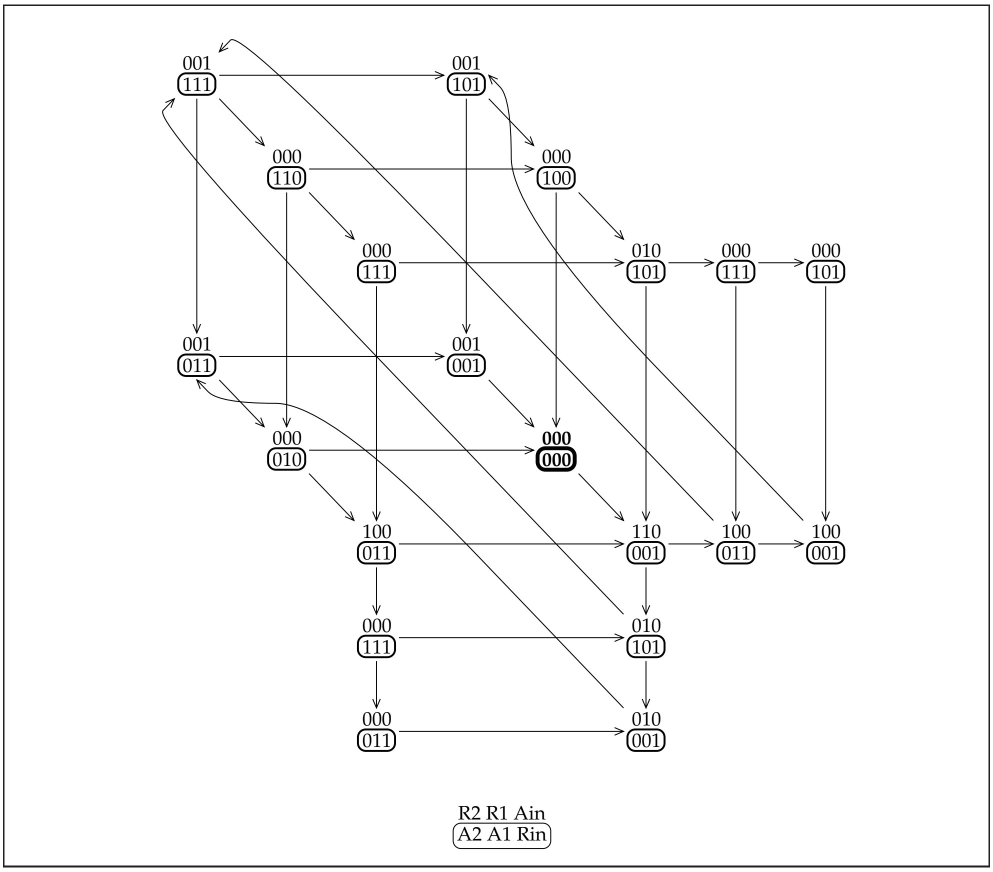


Figure 4.75: Blue diagram for parallel component

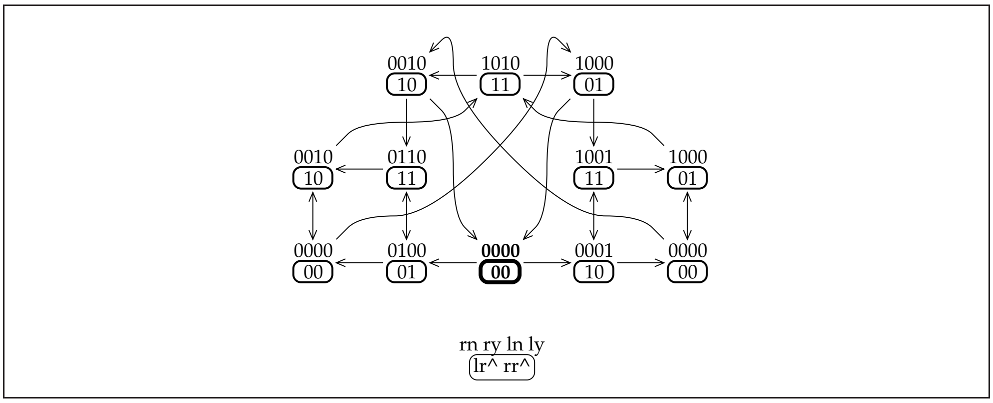


Figure 4.76: Blue diagram for nacking arbiter

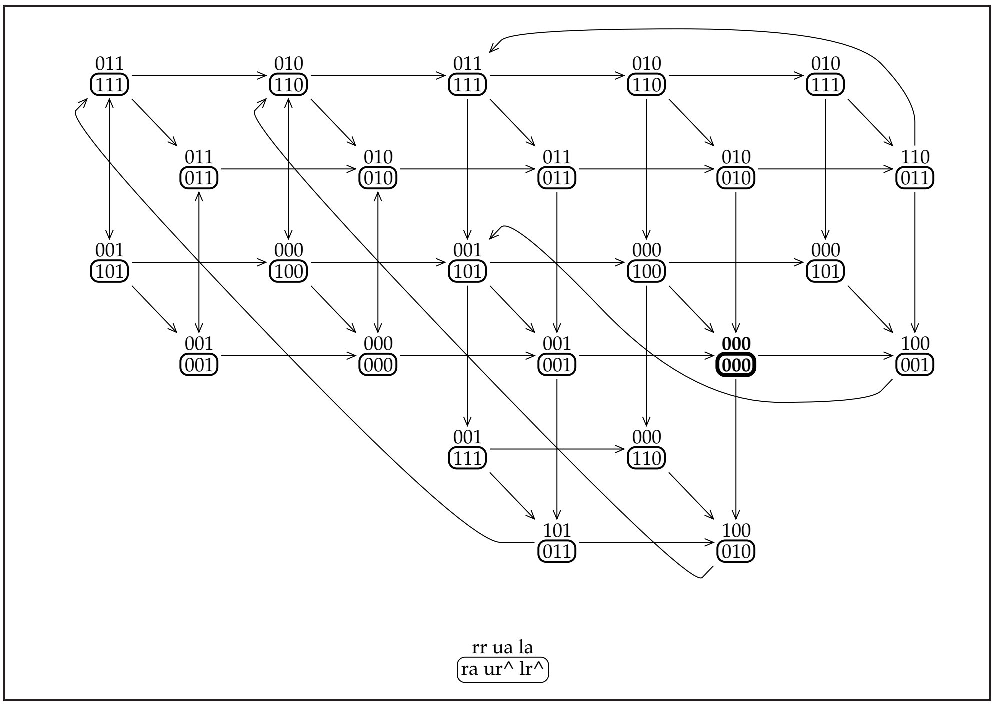


Figure 4.77: Blue diagram for Martin's DME element

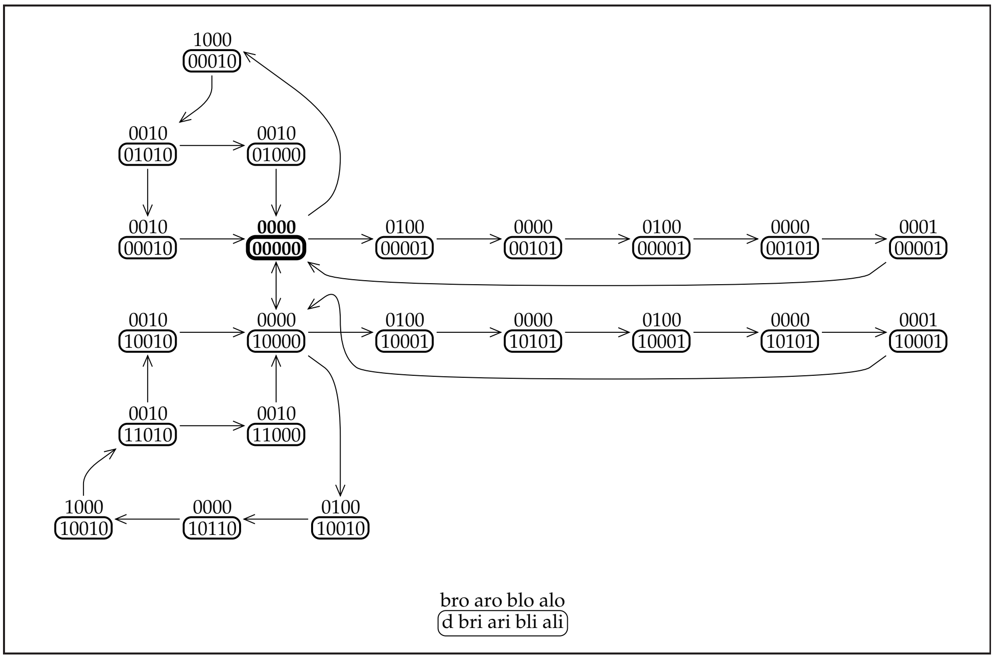


Figure 4.78: Blue diagram for the loadable counter

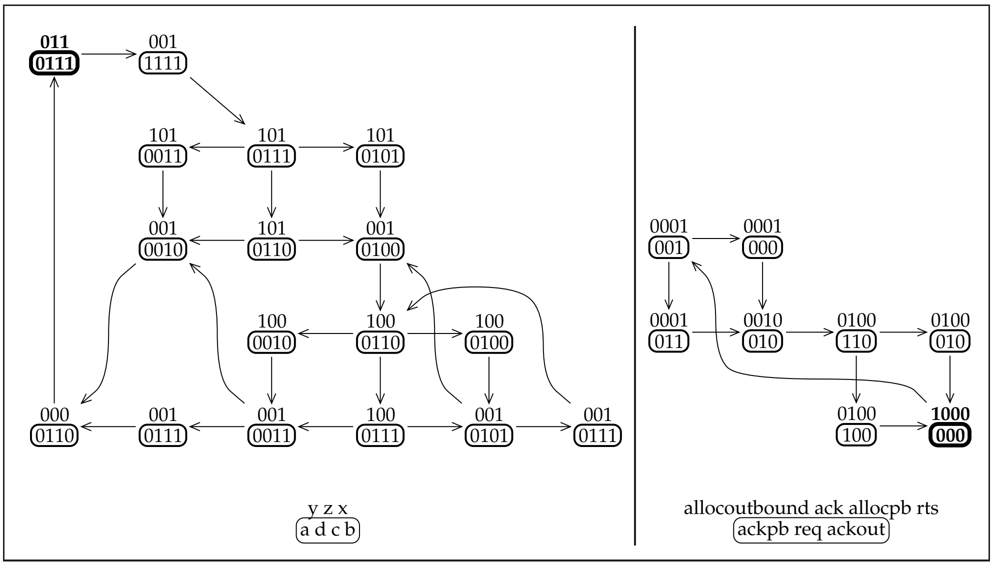


Figure 4.79: Blue diagrams for *isend* and *mp-forward-pkt*

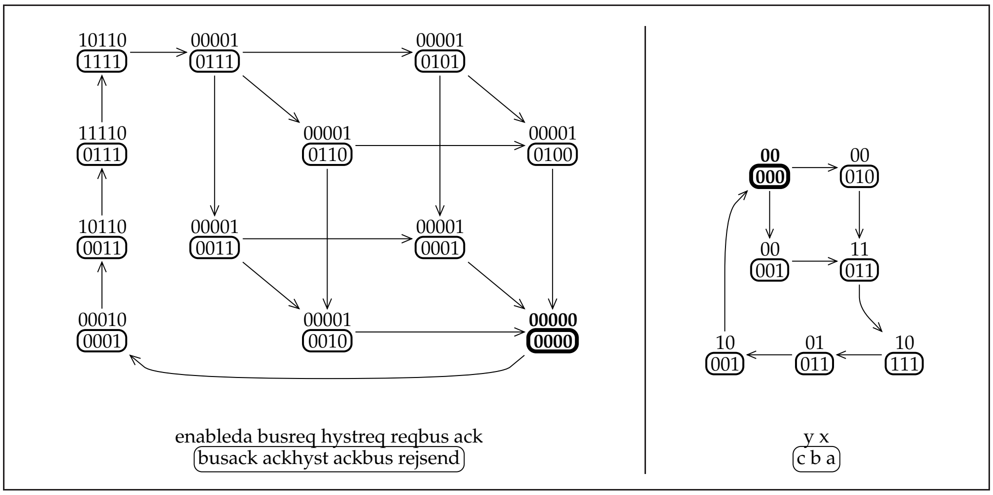
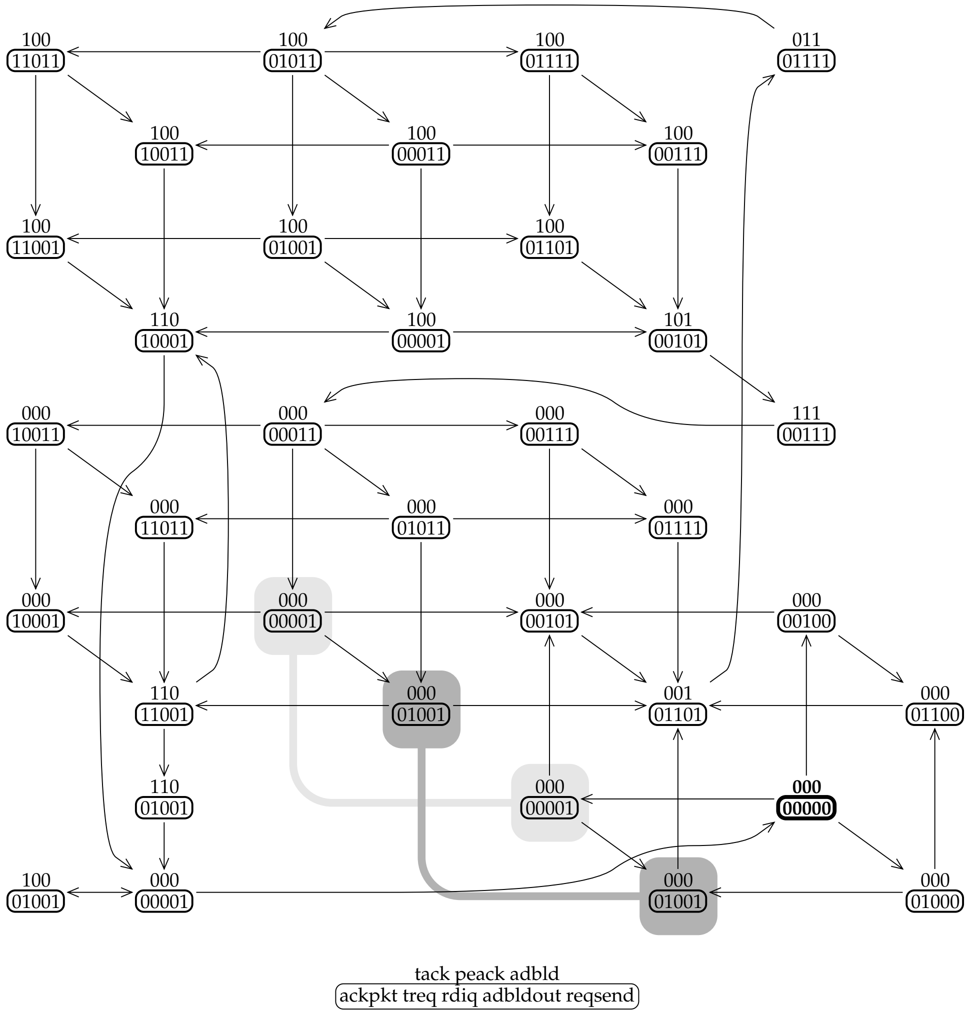


Figure 4.80: Blue diagrams for *nak-pa* and *nowick*



The two pairs of shaded states are the ones that are identified when the `-strong-reduce` option is passed to `L2b`, reducing the diagram to 33 states rather than 35.

Figure 4.81: Blue diagram for *pe-send-ifc*

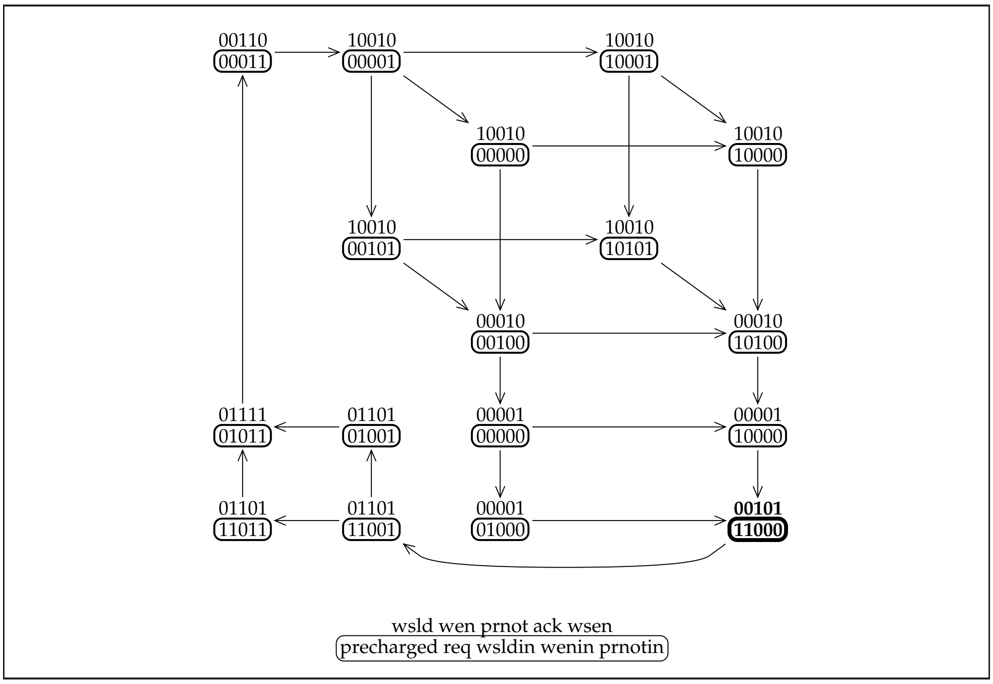


Figure 4.82: Blue diagram for *ram-read-sbuf*

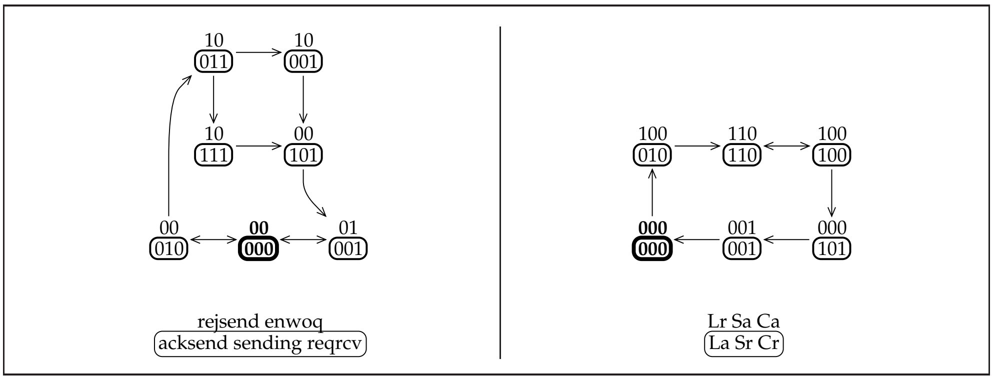


Figure 4.83: Blue diagrams for *rcv-setup* and *rlm*

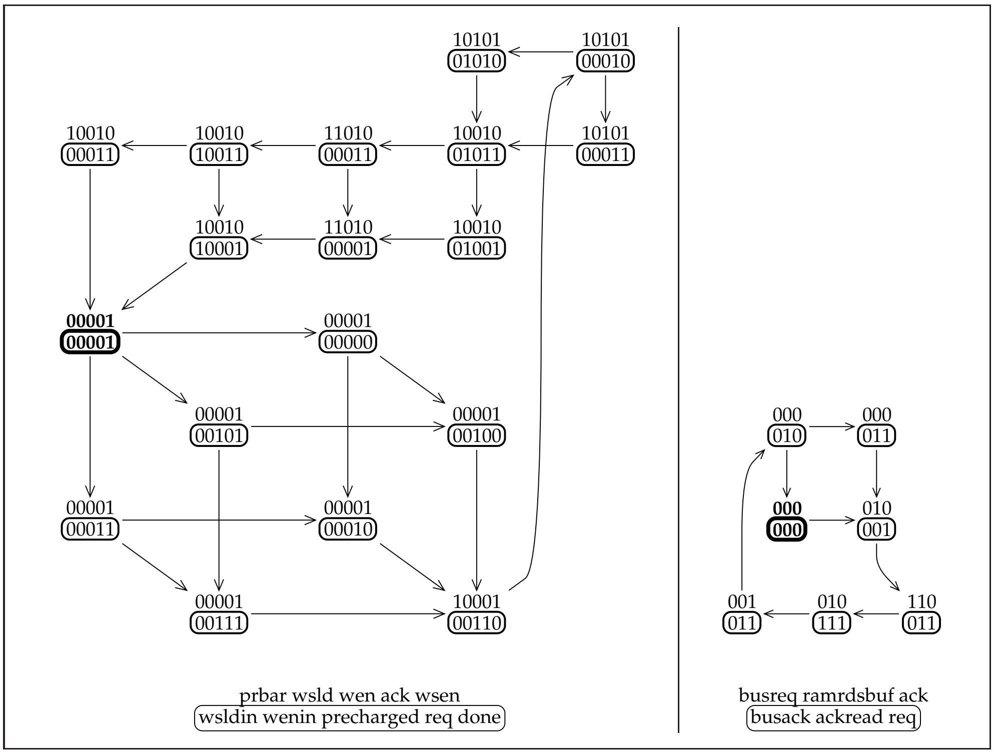


Figure 4.84: Blue diagrams for *sbuf-ram-write* and *sbuf-read-ctl*

Concurrency Reduction 5

I donna suppose you coulda speed things up?
– Inigo Montoya, *The Princess Bride*

Abstract

Concurrency reduction plays a central role in the work described in this dissertation. It is used to produce a large number of circuits from a specification, and the best circuit for a particular task is chosen. This chapter describes the concurrency-reducing operations that are used, first by applying them to a simple example, and then a more complicated example, before giving a brief description of the algorithm used in the prune program. The prune program reads in a starting blue diagram and an environment specification, and produces a file containing all possible blue diagrams which can result from reducing concurrency. A short comparison with earlier work on concurrency reduction is also given.

Structure of this chapter

Section 5.1 describes the concurrency reduction operation, and gives conditions that must be satisfied for it to be used. Section 5.2 gives an example of using concurrency reduction operations on a simple circuit with a trivial environment, which makes it easy to see what the effects of a particular operation are. More general environments need the methods that are described in Section 5.3, which are summarized as an algorithm in Section 5.4. A brief comparison with earlier work on concurrency reduction by Ykman-Couvreur et al. [197] is given in Section 5.5, although the approach presented here has a different goal to the earlier work. Finally, Section 5.6 gives the results of the concurrency reduction process for all the examples in the last chapter.

5.1 Reducing concurrency in blue diagrams

Concurrency reduction in blue diagrams is the action of taking an output transition *t* that occurs as a state *s* is entered, and delaying *t* until the diagram leaves state *s* instead. Figure 5.1 shows an example of a transition o1+, which occurs on entry to state *s*, being delayed until the diagram leaves *s*. Often, the environment will be waiting to observe the delayed output transition before issuing another input

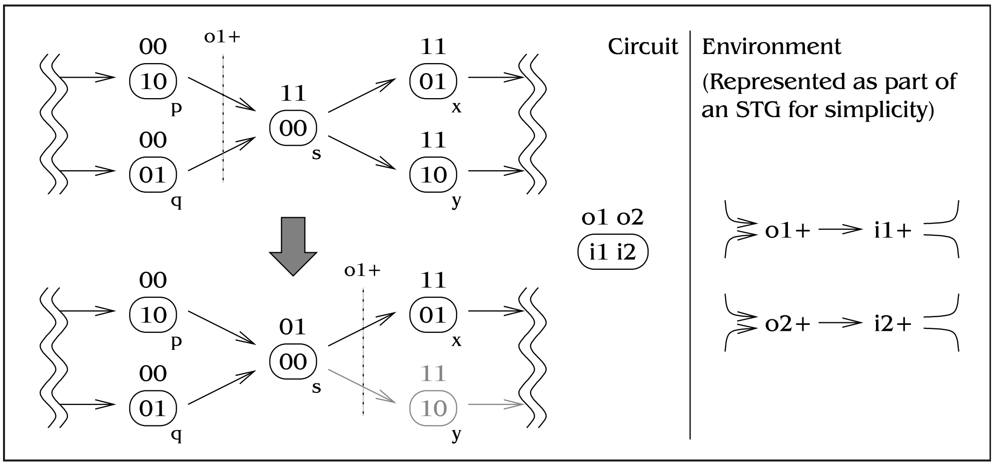


Figure 5.1: The standard concurrency reduction operation

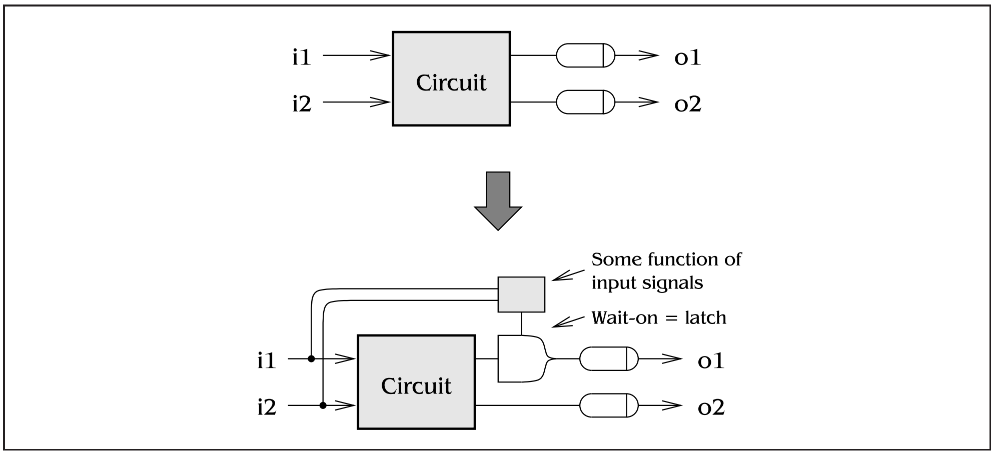


Figure 5.2: The concurrency reduction operation on a circuit

transition; here, i1+ can only happen after o1+, so delaying o1+ will stop the input i1+ being given in state *s*, and will remove the arc from state *s* to state *y*. If *y* has no other predecessors, this will remove *y* and all arcs from *y*, possibly removing other states as well. The action of removing states by delaying output transitions is called *pruning*, by an obvious gardening analogy, and the program that performs concurrency reduction is called prune.

The operation of concurrency reduction has an analogue at the circuit level, as shown in Figure 5.2. Delaying an output corresponds to adding a wait-on or latch in series with one of the unbounded finite delays on the output of the circuit. This latch waits until some combination of inputs is seen before allowing the output transition through. A circuit would not actually be built like this; Figure 5.2 is simply a thought experiment. The circuit in Figure 5.2 cannot be observed to be different from the original circuit in a finite time, because the extra latch delay could always have been caused by the unbounded delay in the original circuit. Equivalently, all traces of signals that are possible when the pruned BD is composed with the environment could also been seen in the original BD, although the reverse is obviously not true.

5.1.1 Conditions that must be satisfied for pruning to occur

In Figure 5.1, the transition `o1+` could be delayed until after state s because every arc into s came from a state where `o1` was low, but `o1` was high in s. This leads to the first condition for pruning being possible:

Condition 1: Output x may be changed to Boolean value v in state s if $x = \text{NOT}(v)$ in state s and $x = v$ in all predecessors of state s.

At a first glance, it appears that a condition on the successors of s is also required. Consider what would happen in Figure 5.1 if the outputs in state x were 01 rather than 11. When taking the path $p \rightarrow s \rightarrow x$ in the upper diagram, `o1+` and `o1-` transitions would be produced, but in the lower diagram on the same path, both these transitions would be lost. However, it turns out that this problem can never occur. Because the interfaces between modules are delay-insensitive, the `o1+` transition when entering state s in the upper diagram must be acknowledged before the circuit is allowed to send an `o1-`, but the only candidate for this acknowledge is the `i2+` transition that causes the move from state s to state x. The environment must have been waiting to see the `o1+` transition before providing the input that would cause the change to state x, so delaying `o1+` will stop the arc from s to x being taken, and hence the problem will not occur. Although state x could not have that value of `o1`, state y could, because the arc from s to y is removed by the pruning operation. This argument can be applied to any situation where a pair of transitions look as though they might be lost.

The pruned diagram must be free from deadlock, so:

Condition 2: The pruned blue diagram, when composed with the environment blue diagram as described in Section 5.3.2, should be free from deadlock.

This condition is difficult to use in the simple example, so will be replaced in the next section by a weaker condition, which is easier to see in the blue diagram:

Condition 2a: All states in the pruned diagram must have at least one arc to another state.

Condition 2a has only been introduced to simplify examples in the text; the stronger condition 2 is used inside prune.

A final, very obvious condition is:

Condition 3: After removing any arcs that cannot be taken, and any states that have no arcs to them, the initial state must still exist.

The initial state needs to be present in all pruned diagrams because the implementation needs a state that it will enter after power-up or on a reset.

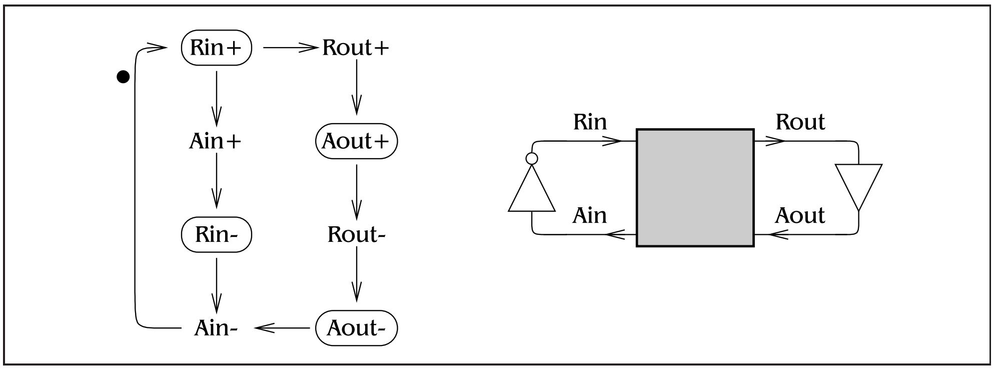


Figure 5.3: Left, STG for a simple pruning example; right, how the circuit will be used

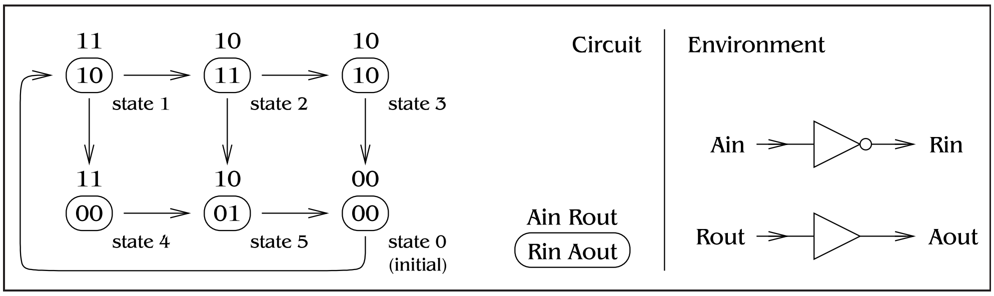


Figure 5.4: Blue diagram and environment derived from Figure 5.3

5.2 Application to a simple example

The prune program performs all transformations that satisfy conditions 1, 2 and 3 above, forming the whole set of pruned blue diagrams. To illustrate this procedure, a simple example will be worked through. Some observations will be made about how these transformations behave, which are useful when producing an general pruning algorithm.

5.2.1 Example used

Figure 5.3 gives the specification of the example circuit, in terms of an STG on the left and the environment behaviour on the right. This STG describes a circuit which partially decouples handshakes on its left and right, so is quite similar to the Furber/Day latch controller. From this STG, L2b produces the blue diagram shown on the left of Figure 5.4; for the time being, the environment will be considered to be that shown on the right of Figure 5.4.

5.2.2 Possible concurrency-reducing transformations

Condition 1 can be used to provide candidate concurrency-reducing transformations: any signal that has a different value in state *s* to its value in all predecessors of *s*. This gives the following possibilities:

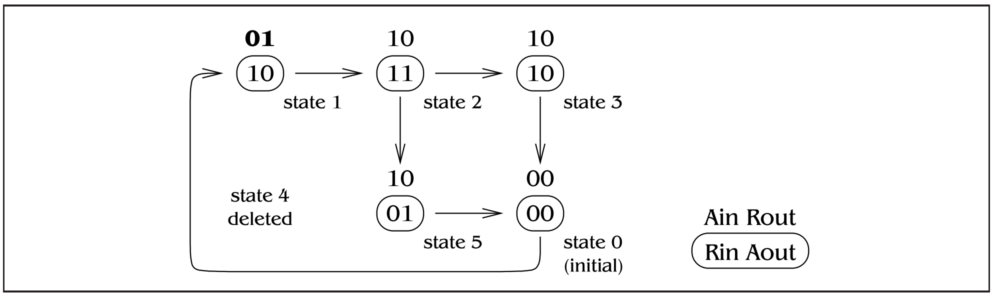


Figure 5.5: Blue diagram after transformation α

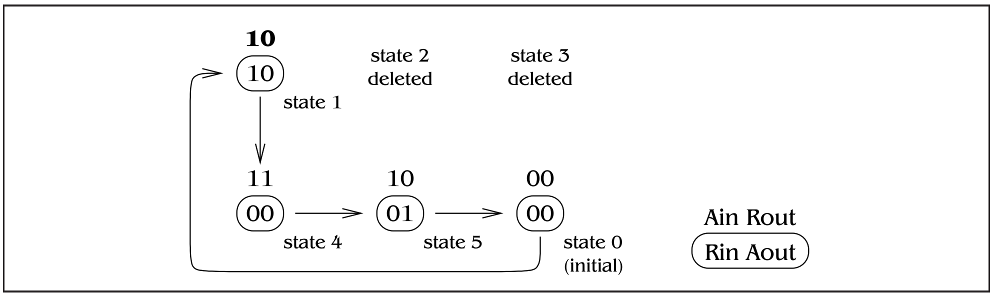


Figure 5.6: Blue diagram after transformation β

- Change the outputs in state 0 from 00 to 10.
  This delays the `Ain-` transition until after the diagram leaves state 0, which stops `Rin+` being given in state 0, causing deadlock and violating condition 2a. After removing all states with no arcs to them, the entire diagram disappears, so this violates condition 3 as well.

- Change the outputs in state 1 from 11 to 01: transformation α.
  This delays `Ain+`, so stops `Rin-` being given and hence removes the arc from state 1 to state 4. State 4 has no predecessors so is removed giving the diagram in Figure 5.5. The changed outputs are marked in a bold font. Conditions 2a and 3 obviously hold, so this is a valid transformation.

- Change the outputs in state 1 from 11 to 10: transformation β.
  This delays `Rout+` and hence stops `Aout+` in state 1, removing the arc from state 1 to state 2, which deletes both state 2 and state 3. This produces the diagram in Figure 5.6. Conditions 2a and 3 also hold.

- Change the outputs in state 2 from 10 to 11: transformation γ.
  This delays `Rout-` until after state 2, so `Aout-` cannot happen in state 2, removing the arc to state 3. This transformation gives the diagram shown in Figure 5.7.

No other output changes are possible in the original diagram that do not violate condition 1, but a further transformation δ is possible in the diagram in Figure 5.5:

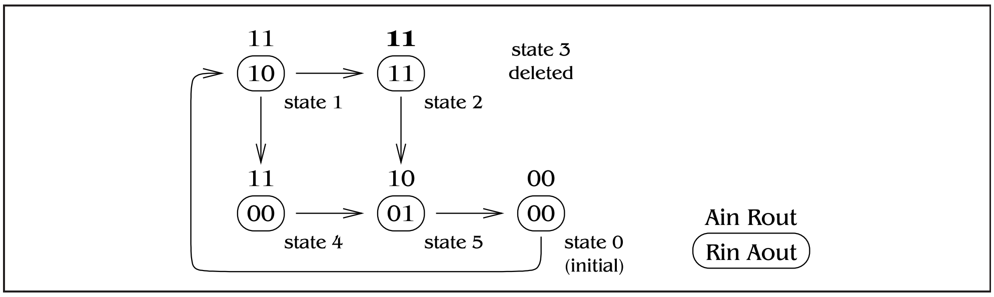


Figure 5.7: Blue diagram after transformation γ

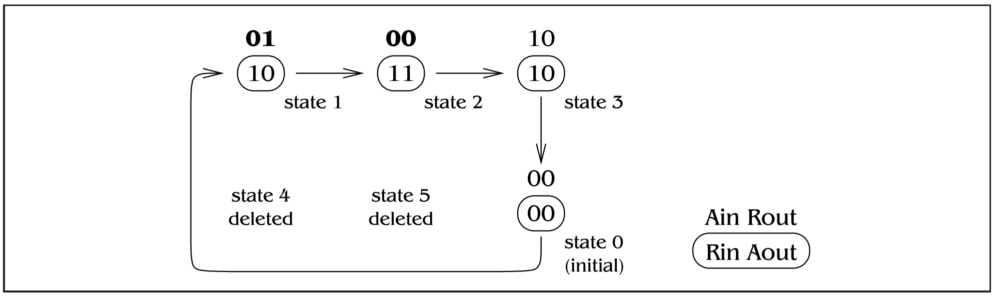


Figure 5.8: Blue diagram after transformation α then δ

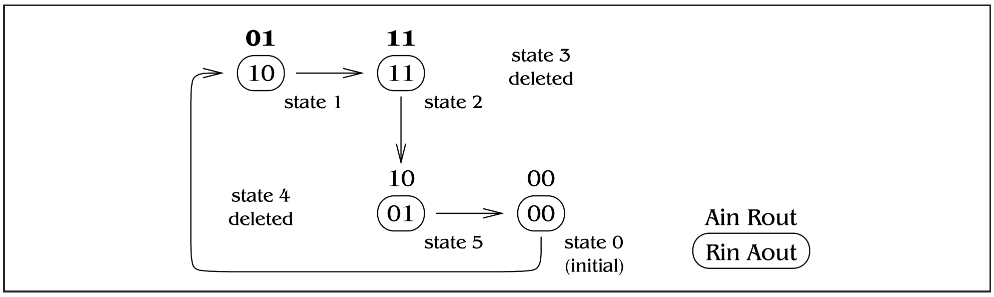


Figure 5.9: Blue diagram after transformation α then γ

- After transformation α, change the outputs of state 2 to 00.
  This stops the arc from state 2 to state 5 being taken, deleting state 5 and giving the diagram in Figure 5.8.

The compound transformation of α then δ will be written $\alpha\delta$.

It is possible to do transformation β on the diagram that results after doing γ (Figure 5.7), but this is the same as doing β on the original diagram. One more compound reduction is also possible: after transformation α, then γ is possible, giving the diagram in Figure 5.9. This can also be derived by doing γ then α.

5.2.3 Observations

Three properties of concurrency-reducing transformations can be seen from the above:

- If x and y are two transformations on a blue diagram, and xy and yx are both possible, then xy and yx give the same transformed diagram. Example: α then γ gives the same result as γ then α. In theory, this leads to a factor of two speed-up in the operation of prune, but in practice the action of checking to see whether xy has already been done to diagram D before attempting operation x on a diagram which was formed by doing y on D takes a significant amount of time, which nullifies any performance gains. For this reason, this observation will be ignored.

- It may be possible to produce the same pruned diagram in two different ways, for example doing $\gamma\beta$ produced the same result as β alone. This is because the state that was changed by γ was removed by β.

- Some reductions may make others possible that were not allowable in the original diagram, such as $\alpha\delta$ being possible above although δ could not be done in the original diagram.

To summarize: duplicate diagrams may occur, and new transformations must be looked for every time a new blue diagram is produced. An algorithm to generate all pruned diagrams from a given diagram must be able to check for duplicates quickly, and must check for new transformations every time a new diagram is found.

5.3 Improved method for a general environment

5.3.1 Problems with the simple example

In the example above, it was easy to determine the effect on the environment of delaying an output transition, because the environment was just a buffer and an inverter. In general, the environment will actually be specified by a blue diagram that is about the same size as that for the circuit. For example, when the STG of Figure 5.3 is given to L2b, the pair of blue diagrams shown in Figure 5.10 is produced. It is possible in this case to reduce the blue diagram on the right of Figure 5.10 to an inverter and a buffer, but this is not possible in general. A way must be found to determine which arcs become untraversable in the blue diagram when a general environment is used.

5.3.2 Solution using a state graph

One way to find the effects of an output change is to:

- Do the output change. As an example, take the transformation γ from before, which was changing the outputs in state 2 to 11.

- Create a circuit-like system by composing the circuit and environment blue diagrams with unbounded finite delays, as shown on the left of Figure 5.11.

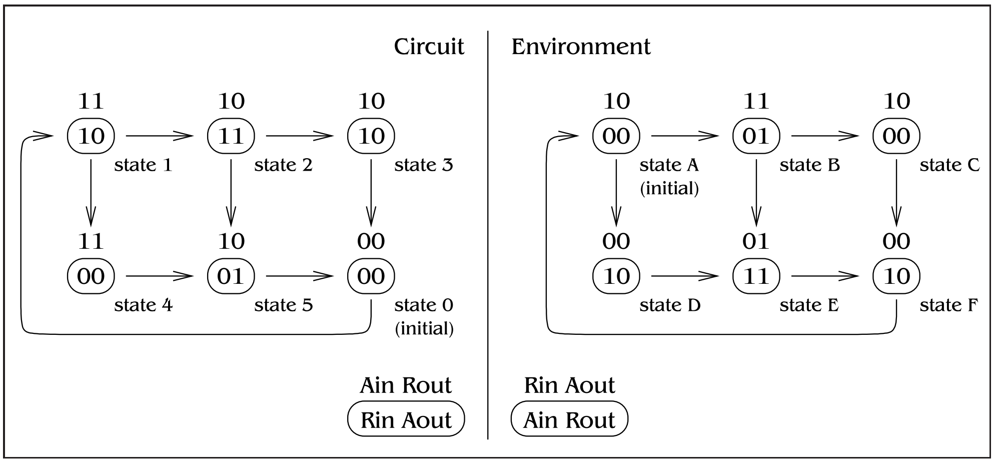
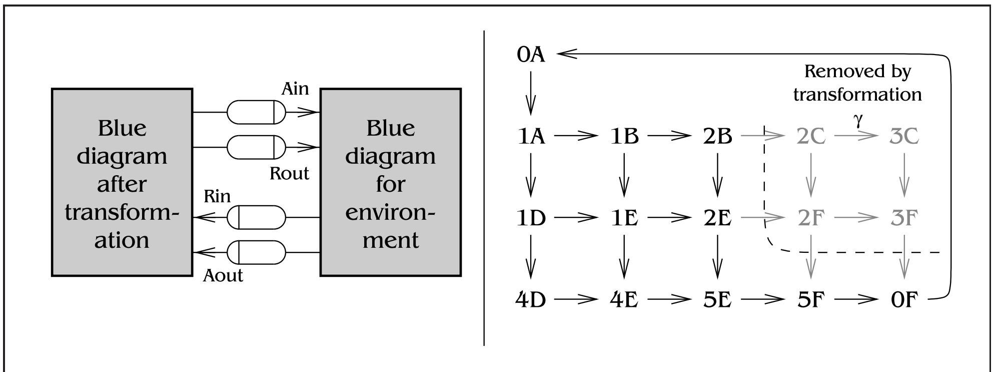


Figure 5.10: An example of a more typical environment: what L2b actually produces.


Blue diagram after transform-ation
Ain
Rout
Rin
Aout
Blue diagram for environ-ment
0A
Removed by transformation
γ
1A
1B
2B
2C
3C
1D
1E
2E
2F
3F
4D
4E
5E
5F
0F

Left, the system used to determine the effect of changing the outputs in a state; right, the graph that results from applying transformation γ to the blue diagrams in Figure 5.10.

Figure 5.11: System and state graph for transformation α

- Simulate this system to find all reachable pairs of states xy where x is a state of the circuit blue diagram and y is a state of the environment blue diagram. Such pairs of states will be called *total states*, because they are states of the whole system. They may be triples or more if arbiter blue diagrams must also be included in the system. The reachability graph of all total states is shown on the right of Figure 5.11, where shaded states in Figure 5.11 are ones that were possible before transformation γ happened, but were not possible after.

- Read off from the graph of total states which states and arcs still exist in the pruned diagram. In Figure 5.11, no total states of the form 3* exist, so state 3 should be deleted, and no arcs exist in the total state graph that go from 2* to 3* or 3* to 0*, so the arcs from state 2 to state 3 and from state 3 to state 0 should be removed in the blue diagram.

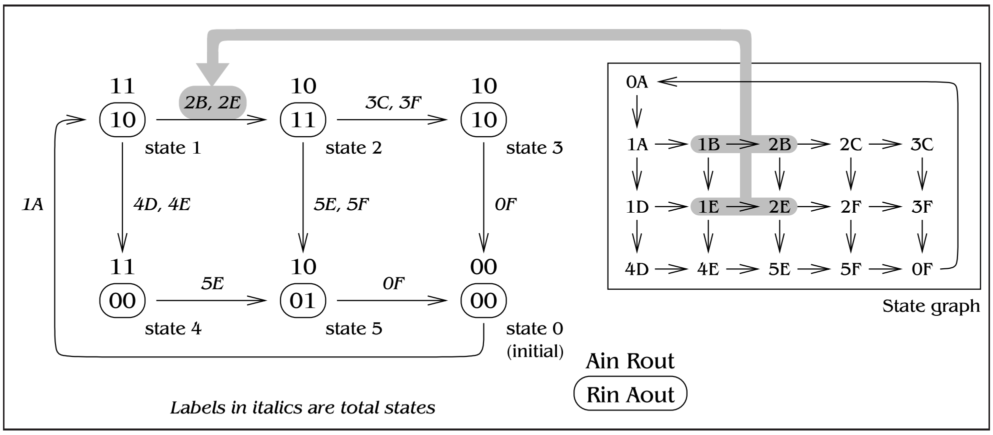


Figure 5.12: Example blue diagram, arcs labelled with total states

The reachability graph in Figure 5.11 is equivalent to the state graph that can be obtained from the original STG, so the total state reachability graph will simply be called a state graph from now on.

5.3.3 Iterative updating of the state graph

Creating a complete state graph every time a concurrency-reducing transformation is performed would be very time-consuming. A method is possible where the state graph is calculated only once, and then local changes made every time an output signal value is changed. The idea is to label each arc in the blue diagram with the total states in the state graph that cause that arc to be taken. Figure 5.12 shows how the arcs are labelled in the example blue diagram from Figure 5.4. The blue diagram changes from state 1 to state 2 when the total state changes from 1B to 2B and 1E to 2E, so the arc in the blue diagram from state 1 to state 2 will be labelled with 2B and 2E; other arcs are labelled in the same way.

Consider again transformation γ, changing the outputs in state 2 to 11. In Figure 5.12, state 2 is entered in total state 2B or 2E. Instead of regenerating the entire state graph after the outputs have been changed in state 2, the system can be *partially resimulated* starting from states 2B and 2E only, and the simulation stopped as soon as the system leaves state 2. This is shown in Figure 5.13. It is found that state 2 is left by changing from 2E to 5E, and that no transitions are possible to total states 5F, 3C or 3F. State 5E is written back to the arc from state 2 to state 5, but the absence of any labels on the arc from state 2 to state 3 means that state 3 should be deleted. After this transformation, state 5 is only entered with total state 5E, whereas before it was also entered with 5F, so state 5 should also be resimulated and any changes propagated forwards. In general, this propagation will only be necessary if the environment blue diagram is non-semi-modular, which should only happen if there is arbitration-type behaviour in the environment that has not been declared with an `arbitrate` keyword. To be safe, resimulation is always performed if arc labels have changed.

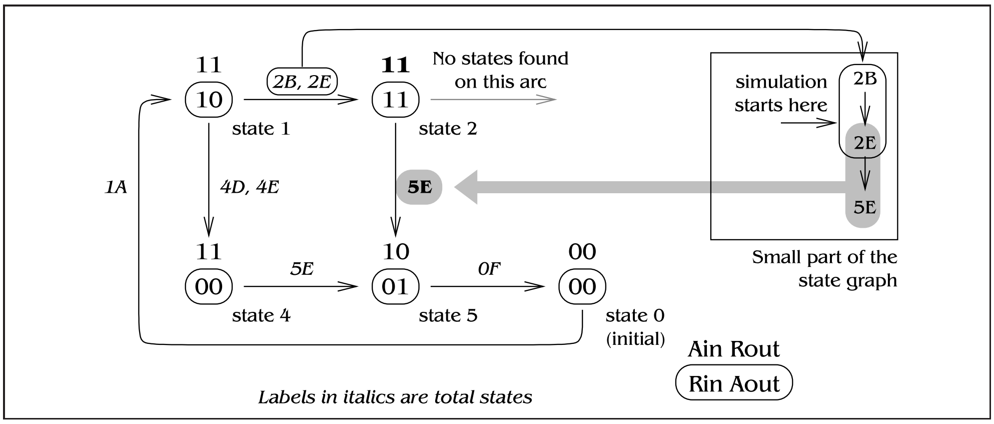


Figure 5.13: Blue diagram after transformation γ, arcs re-labelled with total states

The process of concurrency reduction can only remove total states from the state graph, because there are no possible traces in the pruned diagram that were not possible in the original—neither new states nor new arcs can be added. This means that an arc in a pruned diagram can only be labelled with a subset ($\subseteq$) of the total states that appeared on that arc in the original blue diagram, and consequently, an efficient way to store arc labels of pruned diagrams would be as a bitmapped Boolean array, each bit saying whether a particular total state was there or not. A efficient method of storing pruned diagrams is essential if a large number of diagrams is to be found.[1]

5.4 Description of algorithm

This section gives a pseudocode description of the algorithm that has been outlined in the last section. The two structures needed by the algorithm are:

- O, a set of possible concurrency reducing operations.
  Each operation is a pair (s, o), where s is the state that the operation will be performed on, and o defines the output that will be toggled.

- D, a set of pruned blue diagrams.
  Each diagram is stored as compactly as possible, to squeeze as many diagrams as possible into memory. Only the outputs of the diagram and the labels on arcs need to be stored, because the rest can be inferred from the original diagram.

[1] Actually, this does not make much of a difference. Version 1 of prune used linked list structures for total states on arcs, and was limited to about 3 000 pruned diagrams on a 64MB machine; prune version 2 uses bitmaps, and can keep track of over 50 000 diagrams. Unfortunately, all blue diagrams encountered that have more than 3 000 pruned diagrams happen to have well over 50 000 pruned diagrams as well, so there are no cases that prune v.2 can handle that prune v.1 cannot. Swap space does not help, because the algorithm repeatedly scans through a large subset of the known blue diagrams; when the machine starts to swap, CPU utilization drops to 5% or lower and the program practically stops.

Every time a pruned blue diagram d is added to the set D, the set O is updated with any operations that can be performed on d, if those operations have not been seen before.

This leads to the following algorithm:

```
Create empty $D$ and $O$.
Add the original blue diagram to $D$, updating $O$.
For each diagram $d$ in $D$ {
    For each element $(s, o)$ of $O$ {
        Try doing operation $(s, o)$ on $d$ to get diagram $d'$
        If $d'$ exists and satisfies all three conditions
            Add $d'$ to $D$ if it is not already there, updating $O$.
    }
}

FUNCTION do operation $(s, o)$ on diagram $d$ {
    If $(s, o)$ on $d$ violates Condition 1, abort now.
    Change output $o$ of state $s$ to the new value.
    Collect all total states on entry to the state $s$ in a set $T$.
    Re-simulate the collection of all blue diagrams starting from
        the set of total states $T$. If any state deadlocks at any point,
        set $T' = \emptyset$. Otherwise, let $T'$ be the set of all total
        states that leave the state $s$.
    Update arcs out of the state $s$ with new total states $T'$. If this
        has changed any arc labels to a state $s'$, re-simulate from
        state $s'$ to propagate changes through the diagram.
    Delete any arcs with no labels on them.
    Delete any states with no arcs to them.
    Check conditions 2 and 3 and if it passes, return the new diagram.
}
```

The re-simulation procedure creates a pair of arrays, `before` and `after`, each with one element for each unbounded delay in the system. The current states of all blue diagrams in a total state define all the values of these two arrays, so it is then possible to determine which delay elements are excited. Each of these delay elements can be fired, and the effects on the total state determined to find a successor total state.

The complete algorithm is quite compact, and took only 1 700 lines of C++ to implement. The complete state graph needs to be generated only once, and never stored, so it is not particularly time-consuming for moderately sized examples.

5.5 Comparison with earlier work

Concurrency reduction has previously been explored at the STG level and at the state graph level. STG concurrency reduction was introduced by Vanbekbergen et al.

[183] as a method to solve the USC problem, which is a sufficient but not necessary condition for synthesis. By adding arcs to the STG, the number of state variables required to implement the STG can be reduced. However, this method was limited to live-safe marked graphs with only one rising transition and one falling transition of any signal.

The concurrency reduction work by Ykman-Couvreur et al. [197] operated at the state graph level, which allowed a broader range of specifications to be considered. Their work attempts to remove CSC violations in state graphs by doing a series of transformations, producing a single state graph that requires a smaller number of state signals to synthesize. A concurrency-reduction transformation (x,y) delays an output transition y until after some other transition x with which it was concurrent in the original state graph. Transformations are carried out until the number of CSC violations cannot be further reduced, which tends to produce a state graph that is significantly less concurrent than the original diagram, although constraints can be applied to limit the amount by which concurrency is reduced, especially the concurrency between inputs.

In contrast to Ykman-Couvreur's work, the purpose of the transformations presented here is to produce a large number of possible blue diagrams, each of which can be synthesized. Some of these will require less state variables than the original blue diagram, and will produce smaller and faster circuits for the same reasons as in the earlier work. Some blue diagrams have greatly reduced concurrency, others less so, a few are hardly reduced at all. By producing a large number of possible circuits and picking the fastest or lowest power or smallest, it is hoped that a particularly good circuit can be produced.

Ykman-Couvreur's concurrency reduction transformation does not always make sense when applied to blue diagrams: there are state graph transformations that delay one output until after another output, but enabled output transitions are always concurrent in blue diagrams. The synthesis program, which will be described later, chooses which order to fire outputs in, rather than including this information in the specification. However, comparisons can be made by applying blue diagram transformations on state graphs.

An example piece of a state graph is shown in the centre of Figure 5.14. On the right is an equivalent blue diagram, and on the left is an STG which shows the behaviour of the state graph more clearly. A possible state graph transformation is (`ia+`,`ob+`), which is also an allowable transformation on the blue diagram, producing the state graph and blue diagram shown in Figure 5.15. It is clear that any transformation of the form (*in*∗,*out*∗)[2] will have an equivalent blue diagram transformation: simply take all states from which it is possible to fire *in*∗, and change the value of *out* in those states so that *out*∗ cannot fire.

The blue diagram in Figure 5.15 has had two states altered, so it is the composition of two separate blue diagram transformations, which together have the same effect as the single state graph transformation (`ia+`,`ob+`). The blue diagram after one of the two blue diagram transformations is shown in Figure 5.16. An equivalent

[2] *in*∗ means either *in*+ or *in*−.

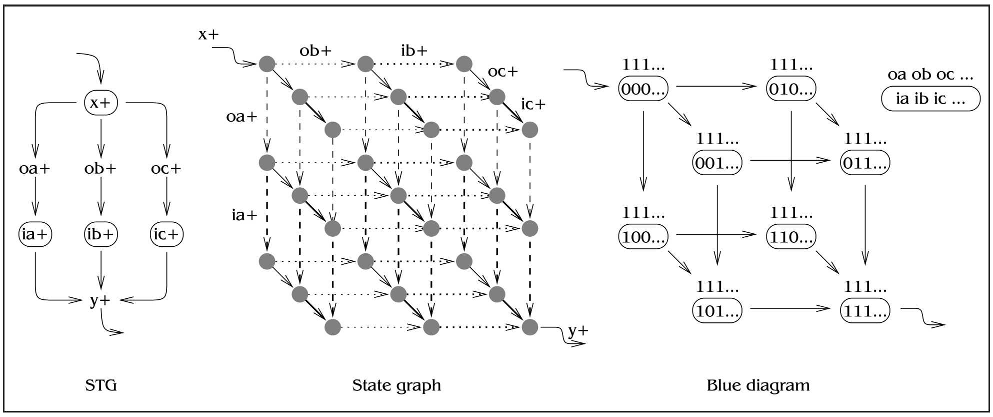


Figure 5.14: Example for comparing the two methods of concurrency reduction

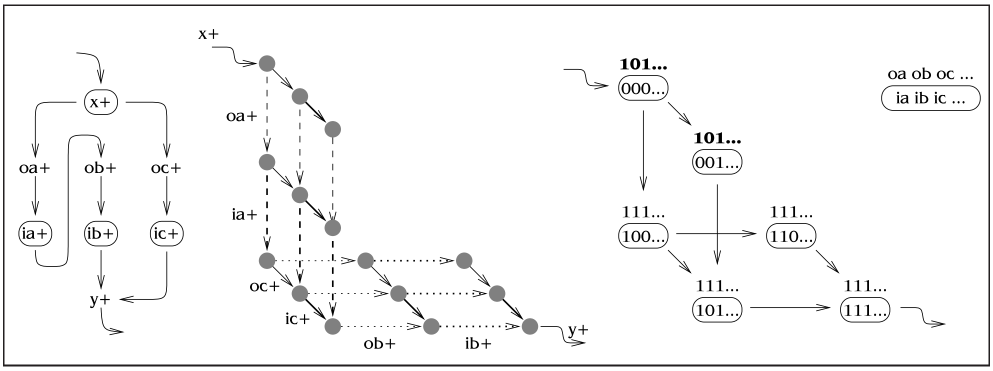


Figure 5.15: Ykman-Couvreur type reduction, applied to Figure 5.14

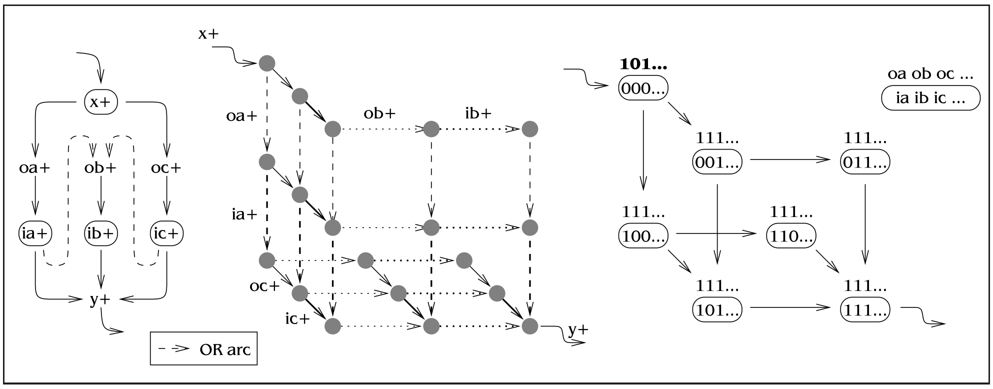


Figure 5.16: Blue diagram reduction that has no Ykman-Couvreur reduction

state graph is also shown, from which it can be seen that no state graph transformation can produce this state graph. A change diagram equivalent to the concurrency reduction state graph is shown on the left of Figure 5.16, from which it can be seen that the state graph is not expressible as an STG. Ykman-Couvreur style reductions always have an interpretation at the STG level, so the reduction in Figure 5.16

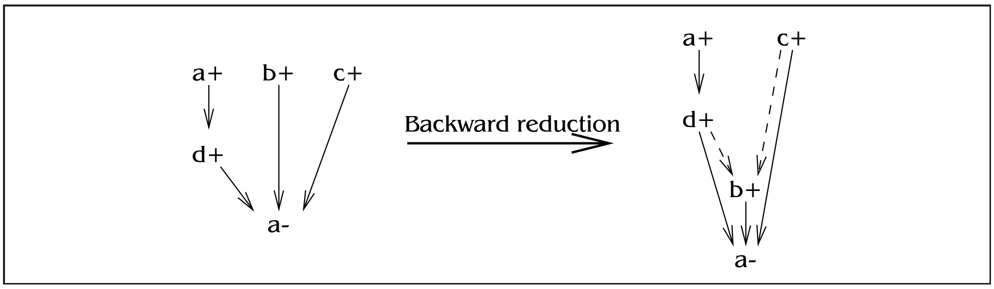


Figure 5.17: A backward reduction from Cortadella et al. [39]

cannot have a equivalent Ykman-Couvreur style reduction. This demonstrates that blue diagram transformations are of a finer granularity than the transformations of Ykman-Couvreur et al., although the difference only occurs when three or more input transitions are each enabled concurrently and individually by output transitions.

While this dissertation was being written, Cortadella et al. [39] published the results of their work on concurrency reduction. Their *forward reduction* algorithm is similar to the work of Ykman-Couvreur et al., but has improved correctness constraints, and can be aimed at logic minimization as well as removing CSC violations. Their *backward reduction* algorithm is more powerful than the forward reduction algorithm, because it allows a finer granularity of concurrency reduction. A backward reduction consists of removing a single arc from the state graph, with some patching to preserve speed-independence. Equivalent transformations on blue diagrams are possible with the algorithms in this chapter, although the patching step is not available; instead, the patching gets done with other blue diagram reduction operations. The situation is similar to the way in which transformation α had to be done before δ to derive Figure 5.8; the work of Cortadella et al. would allow δ but then have to do α to patch the state graph. I believe that the algorithms presented in this chapter are the blue diagram equivalent of backward reduction. It is interesting to note that backward reduction was not implemented by Cortadella et al., because the extra generality of the algorithm could not be clearly interpreted in terms of signal orderings, although there is no such problem with the blue diagram algorithms presented here. It is also interesting to note that the change diagram in Figure 5.16 is very similar to the diagram given by Cortadella et al. as an example of a backward reduction that has no equivalent forward reduction (Figure 5.17).

To summarize,

- Ykman-Couvreur transformations of the form (out_1,out_2) or the equivalent forward reductions of Cortadella et al. do not make sense on blue diagrams; these correspond to implementation choices that will be made by the synth program, rather than changes to the specification of a circuit.

- Ykman-Couvreur transformations of the form (*in*,*out*) do make sense on blue diagrams, and are equivalent to one or more blue diagram transformations.

- Some blue diagram transformations do not have equivalent Ykman-Couvreur style transformations, so blue diagram transformations appear to have a slightly finer granularity. The same is true of the forward reductions of Cortadella et al.; their backward reduction algorithm, which has not yet been implemented, appears to be the STG equivalent of the blue diagrams transformations presented in this chapter.

5.6 Results

Table 5.1 shows the time taken and number of pruned blue diagrams produced by the prune program for each of the examples. The *master-read* example was too highly concurrent to find all concurrency-reduced diagrams—70 000 reduced diagrams were found in 10 minutes before the available swap space was exceeded. This example was instead broken into two parts at the specification stage to create examples *mr1* and *mr2*, as shown in Figure 5.18. In example *mr1*, all transitions above the dividing line were declared as external transitions, apart from `pri` which directly affects the lower part of the STG. Example *mr2* is the opposite. If it is assumed that the number of pruned blue diagrams for *master-read* is approximately the product of the number for *mr1* and *mr2*, then it would have almost 700 000 pruned diagrams, so it is no wonder that prune failed.

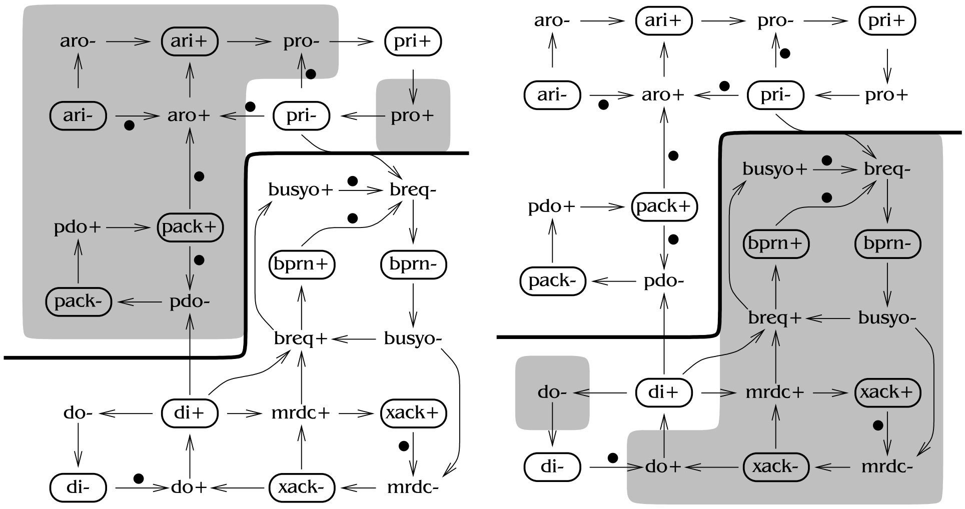
To illustrate the concurrency reduction process, Figure 5.20 gives some pruned blue diagrams from the *atod* example. Arrows between diagrams denote a single concurrency-reducing transformation. The prune program produces the following output for this example:

```
bash$ prune atod
Reading input file ...  done, 11 states and 16 arcs
Number of pruned diagrams found:  34
     6 of size  6     9 of size  7     9 of size 8
     6 of size  9     3 of size 10     1 of size 11
bash$
```

This distribution of pruned blue diagram sizes is typical; there are usually few diagrams that are only a state or two smaller than the original, and relatively few minimal-sized diagrams, with the majority being somewhere in the middle. The distribution of pruned diagram sizes for the latch controller and example *mr1* are shown in Figure 5.19, where the same pattern can be seen.

Example	Runtime (seconds)	Number of pruned BDs	Example	Runtime (seconds)	Number of pruned BDs
latchc	1.46	847	*master-read*	fails	∼700 000?
parallel	1.44	818	↪ *mr1*	7.71	2 310
nacking arbiter	0.08	1	↪ *mr2*	0.75	298
DME	3.39	848	*nowick*	0.08	1
loadable counter	0.15	9	*pe-send-ifc*	0.09	1
alloc-outbound	0.12	1	*ram-read-sbuf*	0.15	15
atod	0.15	34	*rcv-setup*	0.07	1
mp-forward-pkt	0.10	1	*rlm*	0.09	1
isend	0.04	1	*sbuf-ram-write*	0.75	264
nak-pa	0.22	58	*sbuf-read-ctl*	0.08	1

Table 5.1: Results of the prune program



Left, new example *mr1*; right, *mr2*.

The specification files for these two examples are the same as for the original *master-read* example, but with the shaded transitions declared as type external rather than input or output. This gives a pair of circuits which can be composed to give a full *master-read* implementation.

Figure 5.18: The *master-read* example, split into two halves

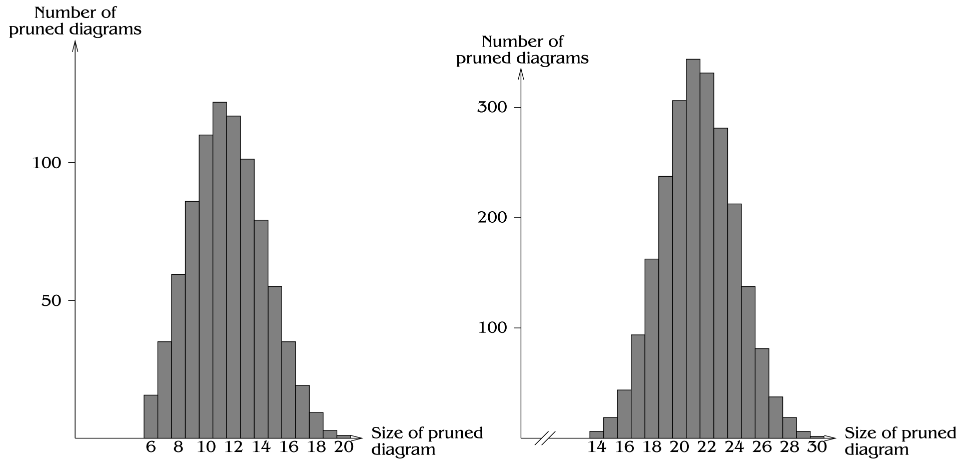


Figure 5.19: Histogram of pruned diagram sizes for the latch controller and *mr1*.

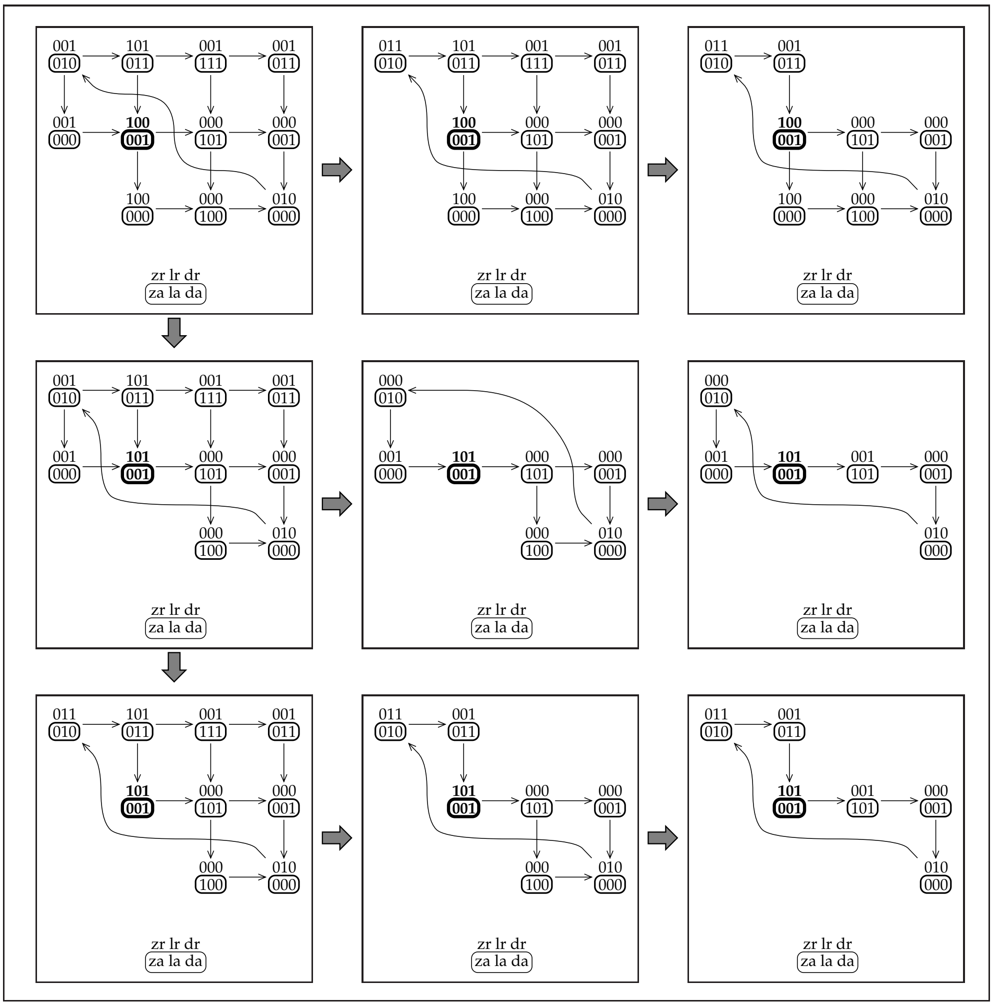


Figure 5.20: Some pruned versions of the *atod* example

Synthesis 6

What day did the Lord create Spinal Tap,
and couldn't he have rested on that day too?
– Marti Dibergi, *This Is Spinal Tap*

Abstract

This chapter describes the synthesis algorithms that are used to create circuits from the set of pruned blue diagrams. Blue diagrams are basically finite state machines, so traditional methods are used, such as those first described by Unger [177]. Several methods are tried for state assignment, all of which are based on Tracey's method [176]. Gates can be created either with or without weak feedback inverters, also called keeper inverters or staticizers. Each of the implementations produced must then be verified, using the techniques described in the next chapter. Comparisons between the different state assignment methods are given in Chapter 8. The algorithms in this chapter have been implemented in the synth program.

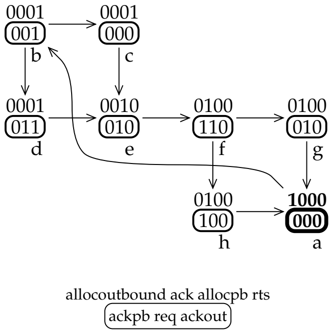
Structure of this chapter

Section 6.1 defines the problem in terms of its start and end points. Flow table minimization and associated concepts such as compatible shrinking and mapping is covered in Section 6.2. State assignment and the creation of truth tables for each output and state variable is dealt with in Section 6.3, where several different methods of finding state assignments are presented. Finally, Section 6.4 gives the methods that were used for creating circuits from truth tables, and gives a short discussion of transistor sizing and how to isolate circuits from capacitative loading outside the circuit.

6.1 Start and end points for synthesis

6.1.1 Start point

Synthesis starts from a flow table description of a Moore-type incompletely-specified state machine (ISSM). For a full description of flow tables, their properties and methods that can be applied to them, see Unger [177]. Each of the set of pruned blue diagrams must be converted into a flow table, which is a very simple task because



		Inputs							
		000		010		100		110	
State	Output		001		011		101		111
a	1000	a	b	-	-	-	-	-	-
b	0001	c	b	**e**	d	-	-	-	-
c	0001	c	-	e	-	-	-	-	-
d	0001	-	-	e	d	-	-	-	-
e	0010	-	-	e	-	-	-	f	-
f	0100	**a**	-	g	-	h	-	f	-
g	0100	a	-	g	-	-	-	-	-
h	0100	a	-	-	-	h	-	-	-

Bold entries in the flow table are ones that have been added because of possible multiple input changes in the blue diagram. In state *f*, inputs `ackpb-` and `req-` are both enabled, so can happen at the same time which moves the flow table directly to state *a*. Dashes in the table denote don't-care entries.

Figure 6.1: Converting the *mp-forward-pkt* blue diagram to a flow table

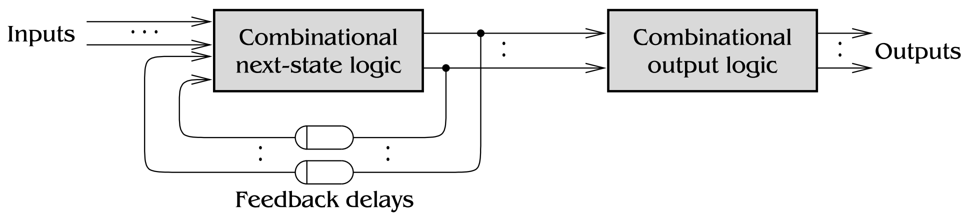


Figure 6.2: Traditional implementation of a Moore machine

blue diagrams are also descriptions of Moore machine. If four states *a*, *b*, *c* and *d* are found in the blue diagram such that $a \longrightarrow b \longrightarrow d$ and $a \longrightarrow c \longrightarrow d$, then the inputs that cause the $a{\longrightarrow}b$ and $a{\longrightarrow}c$ transitions are concurrent, so *d* is also included in the next-state entries for row *a* of the flow table. More states have to be added when three or more inputs can change concurrently. The additional entries are shown in bold for the *mp-forward-pkt* example in Figure 6.1.

6.1.2 End point

Figure 6.2 shows the traditional implementation of a Moore machine, as a block of combinational logic with delays in the feedback path and a block of combinational logic producing the outputs. This style has the problem that outputs can appear before the feedback paths have stabilised, which makes the fundamental mode assumption difficult to uphold. Most synthesis styles today use a network of small state-holding elements:

- SIS [164] uses SR flip-flops with combinational gates to provide the set and reset signals.

- Martin [115, 114] uses a general structure comprising a pull-up tree, a pull-down tree and a weak keeper inverter.
- MEAT [33, 45] uses a general static CMOS gate followed by an inverter. The output of the inverter can be fed back into the static gate, which allows structures such as C-elements to be built. This style can be considered as a generalization of a static C-element.
- ASSASSIN [199] uses a derivative of the SIS approach, using MHS flip-flops.
- Beerel [6] and Kondratyev et al. [94] use a C-element as a flip-flop with combinational trees for the set and reset signals.
- The Amulet group use asymmetric dynamic C-elements [59] and more recently static versions [108].
- The GC synthesis technique described in Yun, Beerel and Arceo [201] appears to use asymmetric C-elements.

One notable exception to this trend is the 3D synthesis tool, described by Yun [203], which uses the traditional arrangement of Figure 6.2.

The approach used in this dissertation will be a combination of the MEAT approach and Martin's structures, because they are very general and together subsume most of the other approaches. Each state variable and output is produced by a single gate followed by an inverter, possibly with a keeper inverter to maintain state, as shown in Figure 6.3. The decision to use keeper inverters on gates is made by the designer, rather than automatically by synth. Inverted outputs are available if necessary, rather than placing bubbles on the inputs to gates. Inputs are also inverted by explicit inverters. The output of any gate can be fed to any other gate. While this approach does not forbid having totally separate output and state variables, it will be assumed that primary outputs are also available as state variables.

6.2 Flow table minimization

The first step in synthesizing a flow table is to reduce the table to an equivalent one with a minimal number of rows. This is done by finding sets of compatible states from the original flow table T, and then creating a new flow table T' with its rows defined by each of these sets of states; this was described in Chapter 2, so will not be described in detail here. As an example, a reduced version of the flow table in Figure 6.1 is shown in Table 6.1, where the sets of states chosen were $\{$a$\}$, $\{$b,c,d$\}$, $\{$e$\}$ and $\{$f,g,h$\}$. Choosing which states to combine into sets is a difficult problem, with many known algorithms for solution. The best method at the moment appears to be due to Puri and Gu [148], so their method is used in synth. The algorithm is briefly described in Section 6.2.1. An alternative method would be to use STAMINA, described by Rho et al. [153], but Puri and Gu's algorithm appears to be faster.

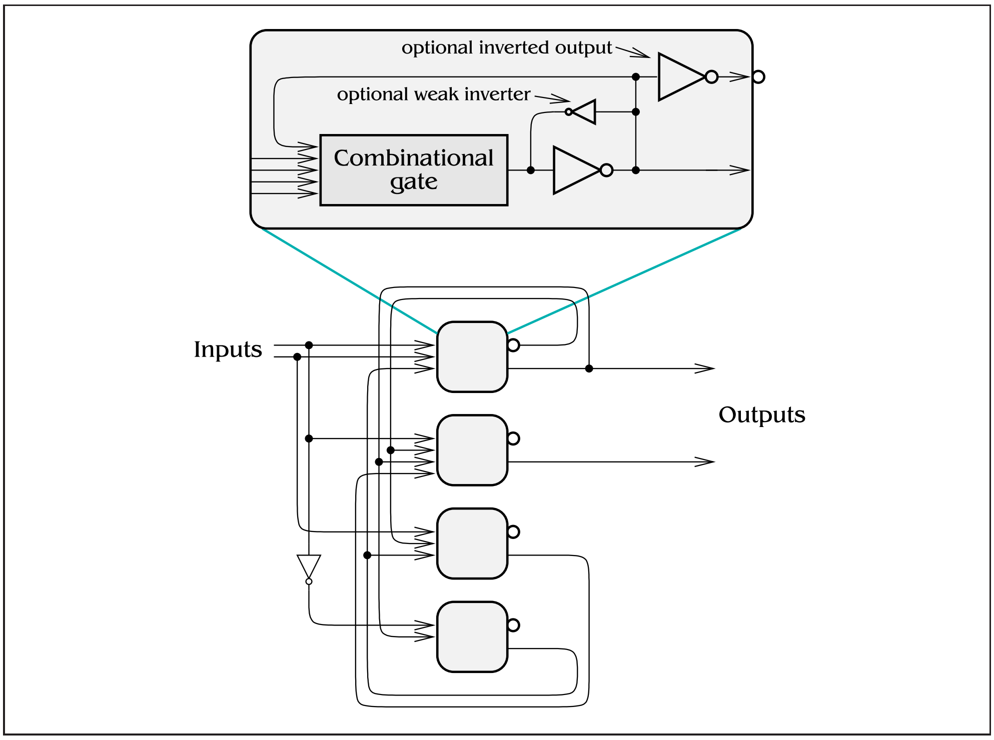


Figure 6.3: Example of the implementation style used in this dissertation

		Inputs							
		000		010		100		110	
State	Output		001		011		101		111
A = {a}	1000	A	B	-	-	-	-	-	-
B = {b,c,d}	0001	B	B	C	B	-	-	-	-
C = {e}	0010	-	-	C	-	-	-	D	-
D = {f,g,h}	0100	A	-	D	-	D	-	D	-

Table 6.1: Reduced table T' for the table T shown in Figure 6.1

		Inputs							
		000		010		100		110	
State	Output		001		011		101		111
A = {a}	1000	A	B	-	-	-	-	-	-
B = {b,c}	0001	B/C	B	D	C	-	-	-	-
C = {c,d}	0001	B/C	-	D	C	-	-	-	-
D = {e}	0010	-	-	D	-	-	-	E	-
E = {f,g,h}	0100	A	-	E	-	E	-	E	-

Table 6.2: Reduced table showing choice in the next-state entries

If the sets of states used to reduce the table are such that one state of the original table is contained in more than one state of the reduced table, then the reduced table may have several possible next-state entries; an example of this is shown in Table 6.2, which is again a reduction of the Table given in Figure 6.1. This reduction is non-minimal, so would never occur, but this sort of choice does happen in larger tables. Two operations may now be carried out, both of which are described by Rho et al. [153]:

- Shrinking the compatibles. In this case, either {b,c} could be changed to {b}, or {c,d} could be changed to {d}.
- Mapping, which is the action of selecting one out of a number of possible next-state entries in the flow table. In either of the rows where B/C occurs, one of B and C would be selected.

The purposes of these steps is to create an ISSM as the output of the minimization procedure. Table 6.2 does not meet the definition of an ISSM, which requires that next-state entries are either defined as a single state, or not defined at all. This is for the benefit of current synthesis approaches. Section 6.2.2 investigates how to perform the shrinking and mapping, and also how to determine whether or not it should be done.

In this dissertation, flow table *reduction* is used to describe the process of finding compatibles that produce a reduced table, and flow table *minimization* is reduction followed by shrinking compatibles and mapping.

6.2.1 Puri and Gu's reduction algorithm

The method used to reduce flow tables in synth was the algorithm described by Puri and Gu [148]. No modifications were needed to this algorithm, so it was simply reimplemented in C++ as it was described in [148]. Some of the routines that are invoked by Puri and Gu's algorithm, such as Unger's pair chart algorithm to find maximal compatibles, were altered. This section will give a brief overview of the method used. The algorithm is very efficient, and is capable of reducing much larger tables than the ones that are encountered in this dissertation.

The first task is to find the maximal compatibles in the original table. All compatible pairs of states are found, and then Unger's pair chart algorithm [177] used to determine the max compatibles. If the number of known compatibles exceeds 1000 during the algorithm, then the computation is terminated and a fall-back algorithm used. The fall-back algorithm creates n compatibles C_i by letting $C_i = \{i\}$ for $i = 1, 2, \ldots n$, and then attempts to add state $((i + j) \bmod n)$ to C_i for $j = 1, 2, \ldots n$ while making sure that every C_i remains a compatible. The set $\{C_i\}$ after this process, with duplicates removed, is necessarily composed of maximal compatibles only, so it is used as an approximation to the actual set of maximal compatibles. The fall-back algorithm is only needed for *master-read*, because there are a 48 states that are almost all pairwise compatible, creating a vast number of compatibles during the algorithm.

The next task is to find the prime compatibles from the maximal compatibles, as described in Chapter 2. An alternative approach, due to Bennetts [8], is to find the prime compatibles directly. In all the examples tested, the prime compatibles are exactly the same as the maximal compatibles, so deriving the primes from the maximal compatibles is particularly efficient. This may be an intrinsic property of blue diagrams, but this has not yet been proved.

Once the prime compatibles are known, a maximal *incompatible* is found. A maximal incompatible is a set of states $\{s_1, s_2, \ldots s_k\}$ such that no two s_i are compatible. Each of the s_i must be covered in the solution, so in any solution there must be at least one compatible c_i that covers each s_i. Let C_i be the set of all compatibles that cover s_i, so c_i must be chosen from C_i. Because the s_i are mutually incompatible, $C_i \cap C_j = \emptyset$ for all $i \neq j$, and hence the choice of which c_i to choose from C_i is independent for each i. This observation is the basis of Puri and Gu's algorithm. The search for a solution can be viewed as a tree, with the choice of which c_i to choose from C_i at the first level, choosing c_2 from C_2 at the second level, and so on. The states s_i are ordered so that $|C_1| \leq |C_2| \leq \ldots |C_k|$ which minimizes the number of nodes in the tree.

An important element of the algorithm is a pair of tree-pruning criteria. These can identify pairs of nodes (x, y) at the same level in the tree, such that if a node below x can produce a solution, then so can a node below y. This renders x redundant, so it can be deleted and lower nodes not generated.

Picking one c_i from each C_i may not produce a solution, so the algorithm carries on down the tree until a solution is found. The leaf nodes are heuristically sorted so that the most promising nodes are tried first.

The algorithm is extremely fast on all the examples tried. The longest time taken for a solution was 0.2 seconds, for the *mr1* example.

6.2.2 Shrinking compatibles

For five of the examples, a single state of the original flow table appears in more than one row of the reduced table. For example, the seventh blue diagram produced by prune for *ram-read-sbuf* has 13 states, and the primes returned by the reduction algorithm are a, bcd, e, f, gl, hj, il, k and m. The state l appears twice in this list, as *gl* and *il*. If either *gl* was replaced with *g*, or *il* with *i*, then the resulting sets of states are still solutions to the flow table reduction problem, but because *g* and *i* are not prime compatibles, these solutions will never be produced by Puri and Gu's algorithm. The action of removing duplicated states from primes is called *shrinking compatibles*, after Rho et al. [153]. Early work by Russo and Palamà [157] attempts to maximise the number of don't care entries, but Rho et al. use a more sophisticated method to determine the best way to shrink the compatibles. Shrinking compatibles does not affect the number of rows in the reduced flow table, but might affect the size and speed of the final solution.

When a state of the original table appears in more than one state of the reduced table, the reduced table will have next-state entries that are not uniquely defined. In the example above, if the next-state entry in the original table was *l*, then in the reduced table, the corresponding entry can be either *gl* or *il*. Synthesis algorithms cannot usually accept tables that have non-unique next-state entries, so something must be done to make the next-state entries unique. Shrinking the compatibles solves the problem for all the examples considered in this dissertation, but if there are still outstanding non-unique entries after that, then *mapping* must be employed. Mapping is simply picking one out of a number of possible next-state entries; a good comparison of mapping algorithms is given in [153].

To investigate the usefulness of shrinking compatibles and mapping, the synthesis algorithms that will be presented later were modified to accept flow tables with several next-state entries. The choice of which next-state entry to use was left as late as possible; usually, half or more of the final circuit was in place before a judgement was made as to which next-state entry to choose. By allowing non-mapped and non-shrunk tables to be implemented, the effectiveness of shrinking and mapping could be explored.

Figures 6.4 to 6.7 show the results of shrinking compatibles in the loadable counter, *mr2*, *pe-send-ifc*, *isend* and *ram-read-sbuf* examples. The squares in these figures denote implementations where compatible shrinking did not occur, circles denote ones where it did occur, and lines from a circle to a square show which square a particular circle was derived from. The horizontal axis indicates the size of the resulting implementation, obtained by counting the total area of the transistors relative to a single N-transistor. The vertical axis is the cycle time for a test circuit using the implementation. The loadable counter had twelve solutions returned from Puri and Gu's algorithm, each of which could be shrunk to obtain four more solutions. This graph suggests that shrinking compatibles usually, but not always, creates a smaller and faster circuit.

Figure 6.5 shows the effects of compatible shrinking on the first five blue diagrams for *mr2*, out of the 298 produced by prune. Again, four new solutions were produced from each original solution, but here, several of the shrunk solutions coincide in the graph. The shrunk solutions are often either bigger or slower or both for this example. The other three examples are also inconclusive; sometimes, shrinking compatibles produces faster or smaller solutions, sometimes it does not.

Synthesizing every possible flow table that can be formed by shrinking compatibles is too time-consuming, but creating the flow tables is relatively fast. It would be

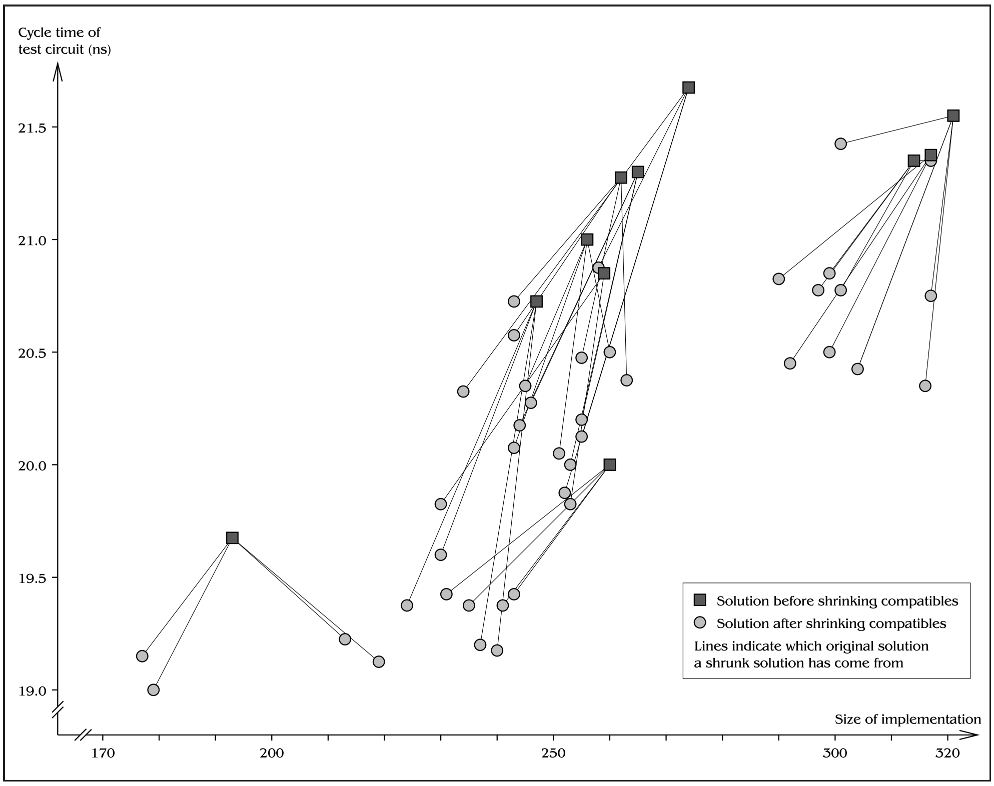


Figure 6.4: Effect of shrinking compatibles on the loadable counter

good to have a scoring function that could be applied to each flow table, and then the table with the best score could be synthesized. This scoring function should reflect the complexity of the final circuit. Four functions were tried:

f_1: Count the number of next-state entries that are not don't-cares.

f_2: Count the number of different next-state entries in each column, and let f_2 be their sum.

f_3: Count the number of different next-state entries in each row, and let f_3 be their sum.

f_4: A derivative of f_3. For each next-state entry x in row r, column c of the table, assign a score k according to:

- if there is another next-state entry of x in the same row r in column c', where c' has the same combination of inputs as column c except that in c', a total of i inputs that were 1 in c are 0 in c', then $k = i$.
- Else if there is no other next-state entry as above, set k to be the number of inputs plus one.

If the flow table has several next-state entries in a single cell of the table, only the lowest cost next-state is counted.

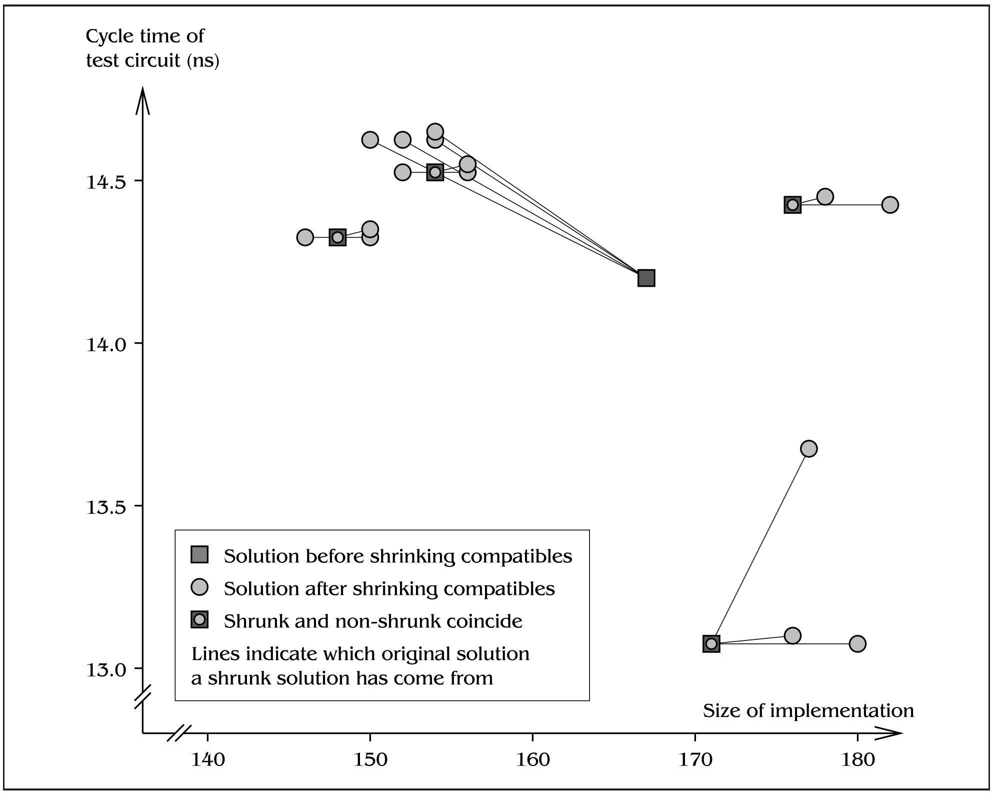


Figure 6.5: Effect of shrinking compatibles on the *mr2* example

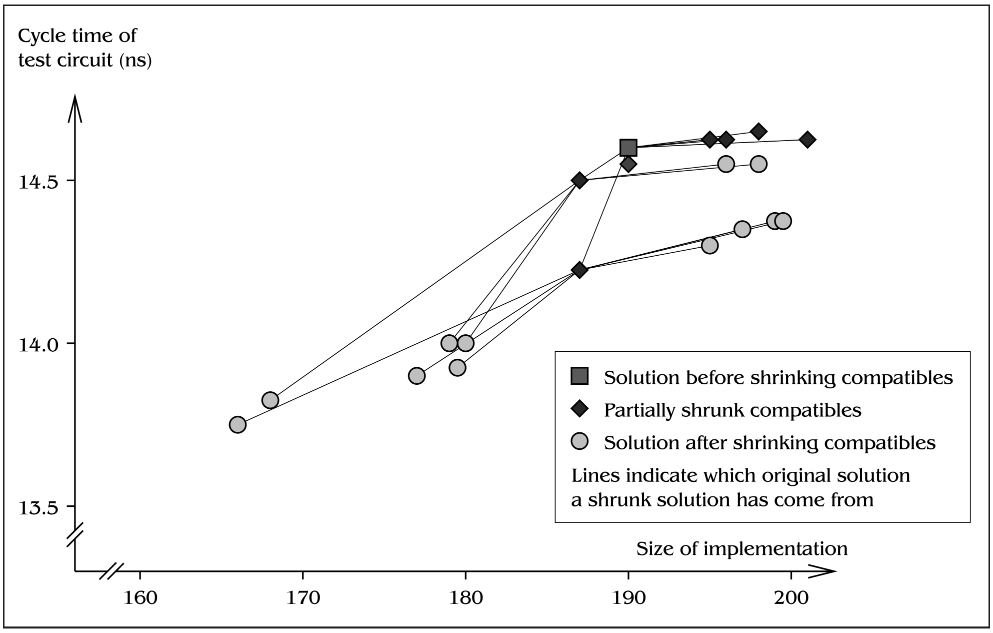


Figure 6.6: Effect of shrinking compatibles on the *pe-send-ifc* example

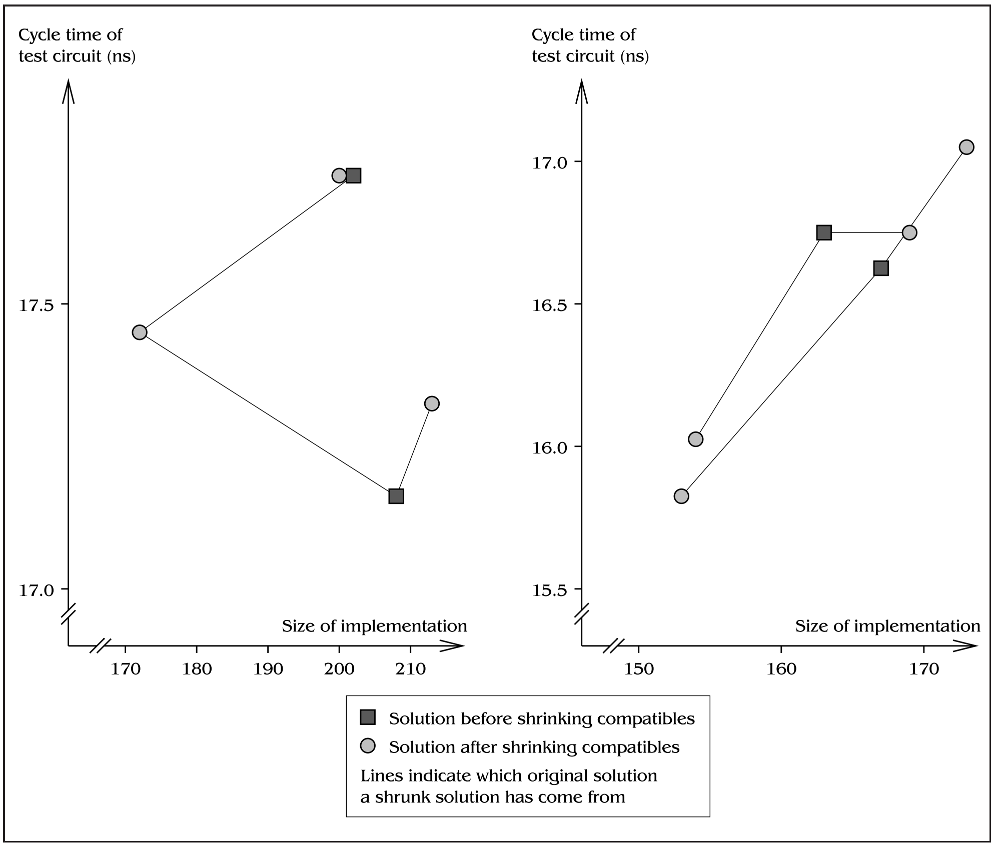


Figure 6.7: Effect of shrinking compatibles on *isend*, left, and *ram-read-sbuf*

The justification behind f_4 is that if in a particular row of the flow table the next-state entry for column 0 was x, then it would probably not change the final solution by very much if the next-state entry in one of columns 1, 2, 4, 8 etc. was x too, because only one input is different between the two columns.

Rather than look at the cycle time and area of implementations separately, a figure-of-merit was defined as the area of an implementation times its cycle time, for which low is good. For each of the four scoring functions f_i, a graph of f_i against the figure-of-merit was plotted; these are shown in Figures 6.8–6.11. It can be seen that a low value of f_1 does not imply a good figure-of-merit, although there is quite a good correlation. There are many loadable counter implementations that get the lowest score for f_1, but only one of these is actually the best. The f_2 function is worse, with all of the loadable counter implementations getting one of two values. The f_3 function is almost ideal: for each example, the lowest value of f_3 also corresponds to the lowest value of the figure-of-merit. The *mr2* and *ram-read-sbuf* examples appear to have a pair of implementations with the lowest value of f_3, but in both these cases, the pair of implementations comes from different blue diagrams, so only one out of the pair will be scored at any one time. Function f_4 has the same problem as f_1: four loadable counter implementations get the same

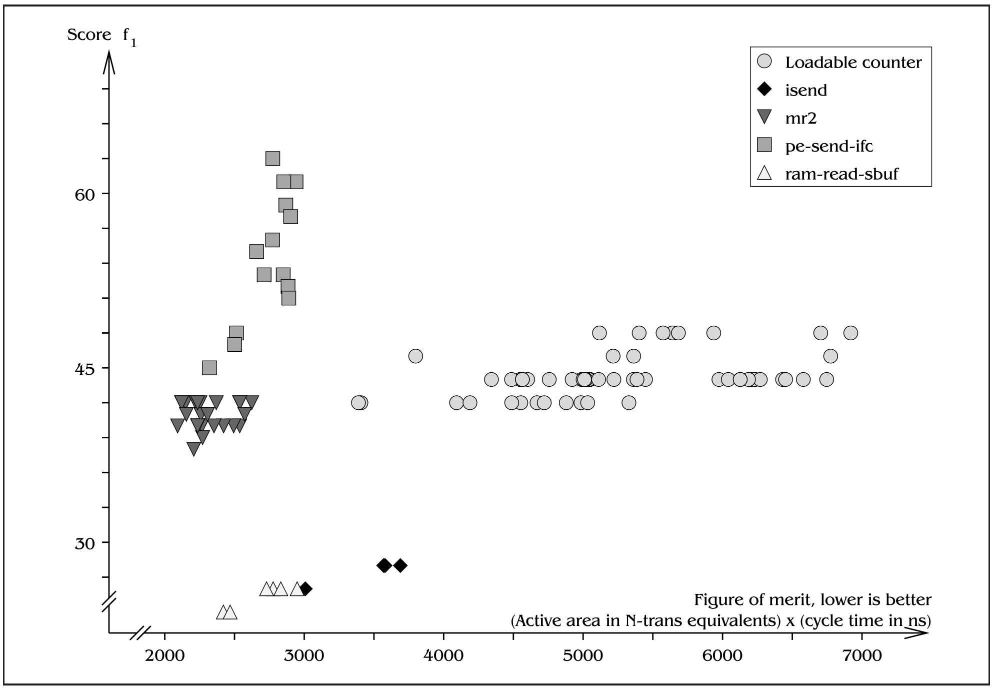


Figure 6.8: The first scoring function against the figure of merit (size × speed)

lowest score, but only one is the best.

These results show that the best way to shrink compatibles is to find the f_3 score of the original flow table and each flow table produced by shrinking compatibles, and then take the table with the best f_3 value.

Mapping

For all the examples that are considered in this dissertation, shrinking compatibles produced a flow table with unique next-state entries, so mapping was never required. When mapping was tried in synth on its own instead of shrinking the compatibles, very poor results were obtained: mapped circuits were always bigger and slower than non-mapped circuits. For this reason, mapping was not implemented in synth. It would be reasonably easy to implement an algorithm which picks next-state entries such that some score function was minimized, similar to the f_i functions above, but without examples to test the code on, it is difficult to find a suitable cost metric.

6.3 Converting the flow table to a truth table

The next stage in the process of creating a circuit is to take the best flow table from the last section, assign a vector of Boolean state variables to each state, and produce a truth table for each state variable. The state assignment and truth table production phases are intricately related, because the state assignment algorithm

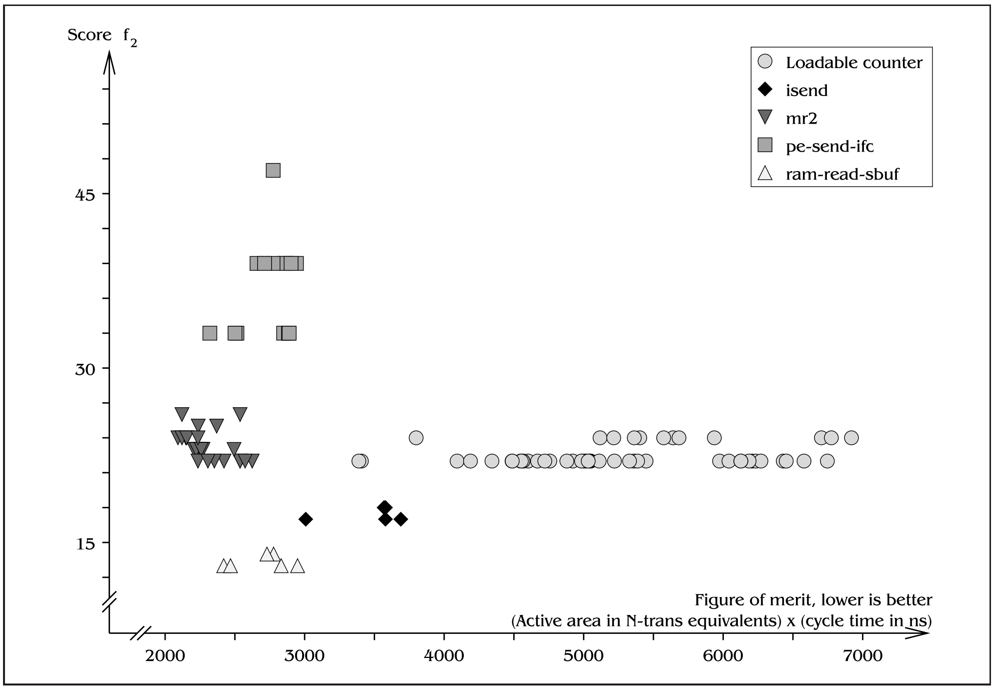


Figure 6.9: The second scoring function against the figure of merit (size × speed)

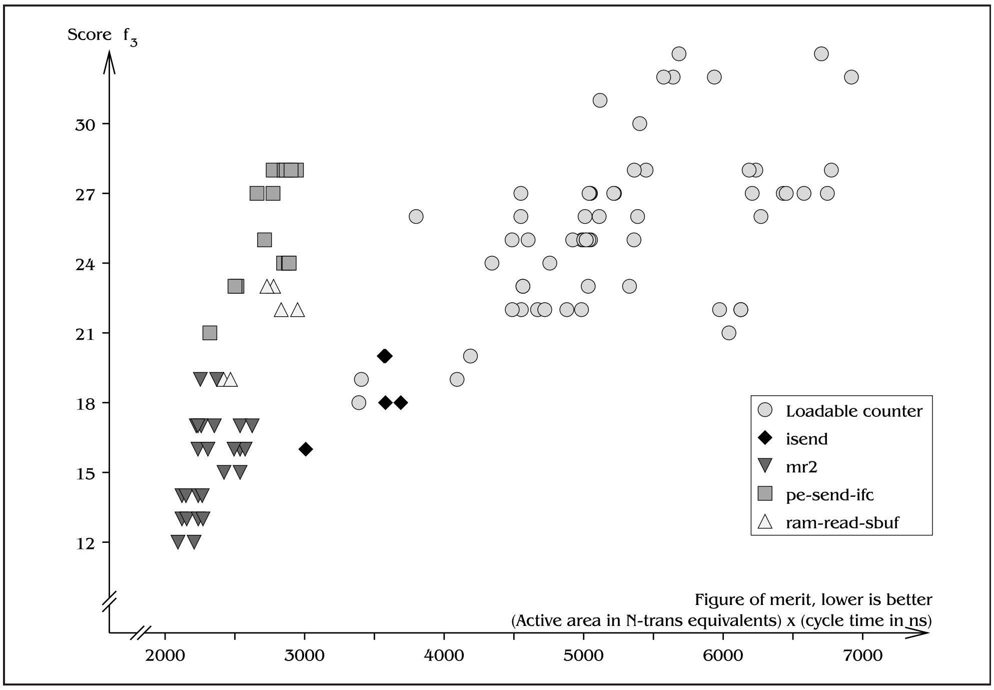


Figure 6.10: The third scoring function against the figure of merit (size × speed)

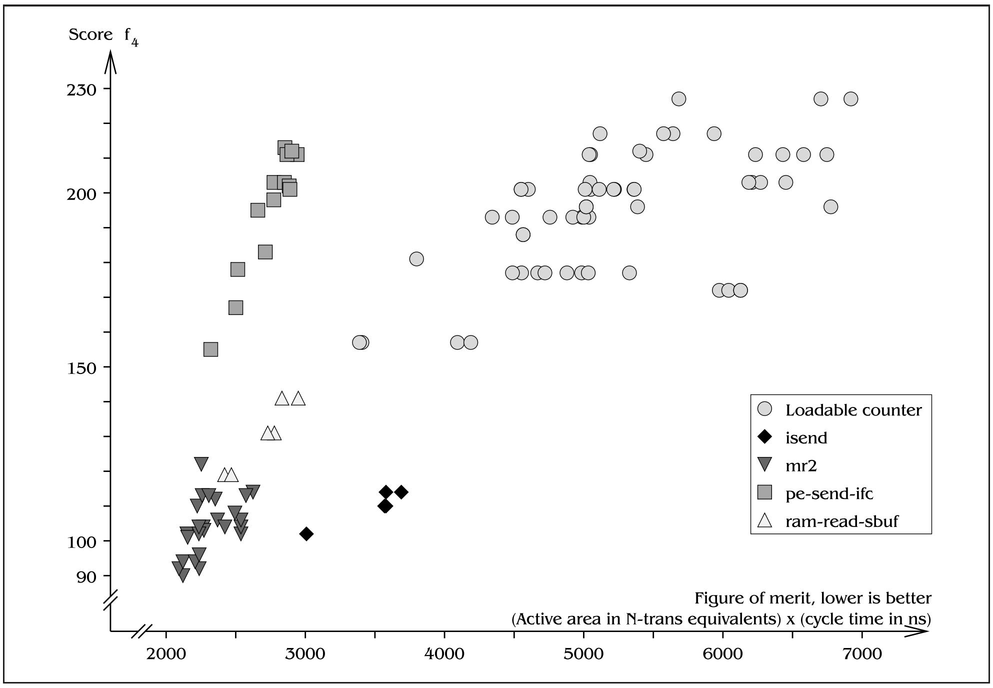


Figure 6.11: The fourth scoring function against the figure of merit (size × speed)

assumes that a particular method will be used to create the truth tables. Three pairs of algorithms will be described, which I will call the Tracey method, modified Tracey method and partial Tracey method respectively. Figure 6.12 gives an overview of how these methods will be used, and five names that can be used to identify the different combinations of the methods. Note that the partial Tracey algorithm for creating truth tables is general enough to accept state assignments from other algorithms. The state assignment algorithms tend to produce several solutions, of which only a few are actually synthesized; the numbers on arrows in Figure 6.12 give how many different assignments are carried forward to be made into truth tables. The second assignment is only used if the first fails for some reason, and the third only if both the first and second fail, and so on.

It was intended that the starred boxes in Figure 6.12 would give smaller circuits than the other algorithms, by producing assignments with fewer state variables. In some cases, the modified Tracey assignments were minimal, so the starred boxes could not produce an assignment with fewer variables; in this case, the two best minimal modified Tracey assignments were used by the partial Tracey truth table generation algorithm. Essentially, the path labelled MP in Figure 6.12 is a fall-back strategy for when the MPP or PP paths fail to produce a circuit. The following four strategies were actually used in synth:

1: TT on its own
2: MM on its own
3: PP, if it fails falling back to MP
4: MPP, if it fails falling back to MP

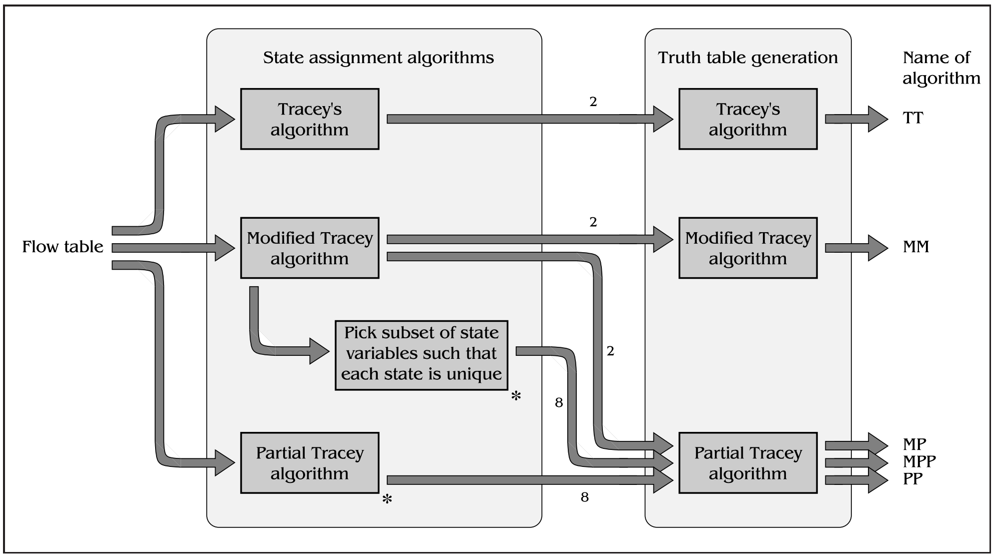


Figure 6.12: Overview of the state assignment and truth table generation algorithms

	I_1	I_2	I_3	I_4
a	a	d	a	a
b	-	b	b	d
c	a	-	b	c
d	e	d	d	d
e	e	d	d	c

Table 6.3: Example flow table to demonstrate Tracey's algorithm

6.3.1 Tracey's algorithm

One of the first asynchronous state assignment algorithms was proposed by Tracey [176]. When it was proposed, it was very time-consuming, but as computers have got faster it has become more tractable. Tracey's algorithm gives unicode single transition time (USTT) assignments, which means that every state has a unique binary code associated with it, and every transition between states happens in a single step, with all variables that need to change rolling together. When several variables change at once, the order in which they change cannot be determined, so the machine can pass through a large number of possible states before reaching the final state. Tracey's algorithm ensures that, regardless of the order in which variables change, the final state will always be the same.

The algorithm will be explained with reference to the flow table shown in Table 6.3. Assume we have a state assignment for this table involving k state variables, and let the assignment for state a be $a_1a_2 \ldots a_k$. In column I_1, state c needs to make a transition to state a, and state d goes to state e. Several state variables may be different between states c and a, so when these state variables change, a number of different states may be entered before the machine settles in state a.

Column	Dichotomies
I_1	$D_1 = ac/de$
I_2	$D_2 = ad/b$, $D_3 = b/de$
I_3	$D_4 = a/bc$, $D_5 = a/de$, $D_6 = bc/de$
I_4	$D_7 = a/bd$, $D_8 = a/ce$, $D_9 = bd/ce$

Table 6.4: Dichotomies produced from the flow table in Table 6.3

Let $[c, a]$ denote the set of states that may be entered when the machine goes from state c to state a, and $[d, e]$ be defined similarly, so

$$[c, a] = \{x_1 x_2 \ldots x_k \quad \textit{where} \quad x_i = a_i \quad \textit{or} \quad x_i = c_i\} = [a, c]$$

$$[d, e] = \{x_1 x_2 \ldots x_k \quad \textit{where} \quad x_i = e_i \quad \textit{or} \quad x_i = d_i\}$$

If there is a state x in both $[a, c]$ and $[d, e]$, then state x may be entered during a $c \longrightarrow a$ transition and during a $d \longrightarrow e$ transition. If this happens, the machine will not know which state should be next after entering state x, so the machine may malfunction. This is a critical race. If there is no such state x, then a critical race cannot occur: the next-state entries of all states in $[c, a]$ in column I_1 should be set to a, all states in $[d, e]$ should go to e, and then the machine must necessarily get to the correct final state. The correctness of the state assignment depends on whether $[a, c]$ and $[d, e]$ have a state in common. Notice that:

$$x \in [a, c] \text{ and } x \in [d, e] \iff \forall i : ((x_i = a_i \text{ or } x_i = c_i) \text{ and } (x_i = d_i \text{ or } x_i = e_i))$$

and therefore

$$\neg\exists x : x \in [a, c] \text{ and } x \in [d, e] \iff \exists i : \begin{array}{l} (a_i = c_i = 0 \text{ and } d_i = e_i = 1) \\ \text{or } (a_i = c_i = 1 \text{ and } d_i = e_i = 0) \end{array}$$

This result is the basis of Tracey's state assignment algorithm. It says that if there is a state variable which takes one value in states a and c, and the other in states d and e, then $c \longrightarrow a$ and $d \longrightarrow e$ transitions in the same column will be free of critical races. Tracey's algorithm finds all pairs of transitions in the flow table that could cause a critical races, and makes sure that there is at least one state variable that prevents the machine malfunctioning. The constraint that at least one variable must be different in a and c to d and e is written ac/de, and is called a *dichotomy*. The full list of dichotomies for Table 6.3 is given in Table 6.4. In column I_2, the transition from a to d must avoid passing through state b, which creates the dichotomy ad/b, and the transition from e to d creates dichotomy b/de. In column 3, a dichotomy is created for each pair of stable states, (b, c), (d, e) and a. Certain dichotomies are redundant and can be removed; for example D_3 will automatically be satisfied if D_6 is, so D_3 can be disregarded. D_5 can also be removed.

Tracey's algorithm is particularly useful for the type of implementation considered in this dissertation, because the outputs of the machine can be used as state variables. Some of the dichotomies may already be satisfied by an output variable;

Max Dichotomy	Dichotomies
$M_1 = abc/de$	D_1, D_6
$M_2 = ac/bde$	D_1, D_7
$M_3 = ade/bc$	$D_2, D_4, \overline{D}_6$
$M_4 = ad/bce$	D_2, D_4, D_8
$M_5 = a/bcde$	D_4, D_7, D_8
$M_6 = ace/bd$	D_7, D_9
$M_7 = abd/ce$	D_8, D_9

Table 6.5: Maximal dichotomies for the flow table in Table 6.3

Flow table					State assignment 1			State assignment 2		
	I_1	I_2	I_3	I_4	abc/de	ad/bce	ace/bd	ac/bde	ade/bc	abd/ce
a	a	d	a	a	0	0	0	0	0	0
b	-	b	b	d	0	1	1	1	1	0
c	a	-	b	c	0	1	0	0	1	1
d	e	d	d	d	1	0	1	1	0	0
e	e	d	d	c	1	1	0	1	0	1

Table 6.6: Final state assignments for the example table

for example, if one of the outputs of the machine in Table 6.3 took the value 1 in state *a* and 0 in all other states, then this satisfies dichotomies D_4, D_7 and D_8 already, so they may be disregarded. This is equivalent to adding more variables to an existing state assignment, which is something that most state assignment algorithms cannot do, such as the algorithms due to Fisher and Wu [56], Kantabutra and Andreou [83], and Tan [173]. Using the outputs as state variables potentially produces a smaller number of state variables and hence a more compact implementation.

The next stage in Tracey's algorithm is to combine dichotomies into *maximal dichotomies*. If a state variable took value 0 in states *a*, *b* and *c*, and 1 in *d* and *e*, then this could be written as the dichotomy abc/de; this variable would satisfy or *cover* both $D_1 = ab/de$ and $D_6 = bc/de$. Dichotomies can be combined into larger dichotomies, and when none of the D_i can be further combined with a dichotomy formed in this way, then it is a maximal dichotomy. Maximal dichotomies M_i are listed in Table 6.5 with the dichotomies they were made from. A bar over a dichotomy means that the states before and after the slash have been swapped, so if $D_6 = bc/de$, $\overline{D}_6 = de/bc$.

The last step in Tracey's algorithm is to pick a minimal-sized set of maximal dichotomies such that each of the D_is that was not deleted is covered by at least one of the chosen maximal dichotomies. Here, either $\{M_1, M_4, M_6\}$ or $\{M_2, M_3, M_7\}$ can be chosen, giving the states assignments shown in Table 6.6.

Tracey's algorithm for state assignment requires that the truth tables for implementing the state variables are derived in a certain way. As was said above, during a transition between two states, for example the transition from *a* to *d* in column I_2,

	I_1	I_2	I_3	I_4
a = 000	000	**101**	000	000
001	-	**101**	-	101
c = 010	000	-	011	010
b = 011	-	011	011	101
100	110	**101**	101	-
d = 101	110	**101**	101	101
e = 110	110	101	101	010
111	110	101	101	101

Table 6.7: Encoded flow table, using state assignment 1

Present state	Next state, input I
a	b or c
b	b
c	c
d	e
e	e

Table 6.8: Example of a non-unique next-state entry

all the intermediate states $[a, d]$ may be entered, so all states in $[a, d]$ must have their next-state entries set to d. This is shown for the first state assignment in Table 6.7, which I will call an encoded flow table, where the states $[a, d]$ in column I_2 are shown in bold. Given this table and an input coding, it is easy to derive truth tables for the state variables.

6.3.2 Non-unique next-state entries

The version of Tracey's algorithm used in synth was modified to allow non-unique next-state entries. If there are two or more next-state entries at a point in the flow table, then a number of dichotomies may be produced of which a limited number need to be satisfied. As an example, consider the single column of a flow table shown in Table 6.8. The transition from state a will produce dichotomies ab/de and ac/de, of which only one needs to be satisfied. Maximal dichotomies are produced as before, but when picking a set of maximal dichotomies, only one of ab/de and ac/de needs to be covered, although both might get covered by chance.

When the encoded flow table is produced, it might be the case that only one of $[a, b]$ and $[a, c]$ are disjoint from $[d, e]$; in this case, it is obvious which next-state entry must be chosen. If both are possible, then as much information is filled in the encoded flow table as can be determined, and Boolean equations derived for the state variables. These Boolean equations define next-state entries for all the rows in the column that is being considered; an example of this is shown in Table 6.9. The next-state entry for state a will be defined, but is unlikely to be a race-free transition to one of the required final states. In Table 6.9, the next-state entry in row a is the

Present state	Next-state derived from Boolean eqs	Next-state for $a \to b$	Next-state for $a \to c$
a = 000	001	0**1**1	**100**
001	001	0**1**1	001
010	010	01**1**	010
b = 011	011	011	011
c = 100	100	100	100
101	100	100	100
d = 110	111	111	111
e = 111	111	111	111
Cost:		3	2

Table 6.9: Finding the cost of the two possible next-state entries

state 001. If the next-state of *a* is declared to be *b*, then three entries in the second column will need to be changed; these are shown in bold in the third column. The fourth column shows the two entries that need to be changed if the next-state is *c*. The number of flow table entries that need to be changed is taken as a heuristic measure of the cost of a particular transition, so the transition with the lowest cost is picked; here, the next-state entry of state *a* will be set to *c*, because that transition had cost 2 rather than 3.

The algorithm in synth employs this heuristic next-state determination in three stages. Boolean expressions are derived for the table, then on the first pass, a next-state entry is only picked if its cost is less than half the cost of all other possible next-state entries. Equations are then re-derived, and on the second pass, a transition is picked if it has a lower cost than the other choices. Any draws are left until the last pass, when hopefully, other changes to the table have broken the tie. Equations are again re-derived, and on the third pass, any ties are broken by choosing the lowest numbered next-state.

6.3.3 Modified Tracey algorithm

One of the problems with Fundamental Mode operation was mentioned in Chapter 2: there are times δ_1 and δ_2 with $0 < \delta_1 < \delta_2$ such that if two inputs I_a and I_b arrive separated by less than δ_1, the inputs are taken to be simultaneous; if the separation was greater than δ_2, they are seen as two separate inputs. If the time separation was between δ_1 and δ_2, then the behaviour is undefined.

The rationale for this is that for some time after an input I_a arrives, the circuit will not have reacted in any way to the input, so if another input I_b arrives, the circuit must react as if they happened together. There will be some period δ_1 where this is true for all inputs I_a. Alternatively, if an input I_a arrives and I_b does not arrive quickly, a sequence of transitions will occur inside the circuit. Then, δ_2 is the time taken by the longest sequence of internal transitions caused by any input. If the separation between two inputs is between δ_1 and δ_2, the circuit may be between states when the second input arrives, which can cause behaviour which was not

		Inputs							
State	Output	000	001	010	011	100	101	110	111
a	1000	a	b	-	-	-	-	-	-
b	0001	c	b	e	d	-	-	-	-
c	0001	c	-	e	-	-	-	-	-
d	0001	-	-	e	d	-	-	-	-
e	0010	-	-	e	-	-	-	f	-
f	0100	a	-	g	-	h	-	f	-
g	0100	a	-	g	-	-	-	-	-
h	0100	a	-	-	-	h	-	-	-

Table 6.10: The *mp-forward-pkt* example again

intended. This can be seen in the flow table of Table 6.3 and the encoded version in Table 6.7. Consider the table to be in state a under input I_1, and change the input to I_2 and then I_3. Looking at the flow table, we would expect the machine to end up in state a or d, depending on the precise timing of the inputs. Looking at the encoded version in Table 6.7, it can be seen that a problem might occur. During the a to d transition in column I_2, the machine might pass through the state 001, and if the machine was in this state when the input I_3 arrived, then the next-state of the machine is undefined.

This problem was tackled by Unger [178], who noticed that when an asynchronous circuit had two or more inputs from different sources, then the time differences between inputs could be arbitrary. Using Tracey's algorithm for state assignment, Unger found that by including additional constraints in the set of dichotomies that needs to be covered, Tracey's algorithm could be made to work for machines with unrestricted input changes. This technique was only demonstrated for the traditional implementation of finite state machines shown in Figure 6.2, but it can also be applied to the implementations considered here. Unger's improved algorithm does not seem to be quite enough to stop misbehaviour[1], but because flow tables derived from blue diagrams have a particularly simple structure, an algorithm based on Unger's idea can be employed.

To explain the algorithm, I will used the *mp-forward-pkt* flow table that was given earlier, shown again in Table 6.10. Consider the $b \rightarrow e$ transition in column 010. Tracey's algorithm produces the dichotomies be/fg, ce/fg and de/fg for this column, which reflect the fact that the sets of states $[b,e]$, $[c,e]$ and $[d,e]$ must each have no intersection with the set $[f,g]$. Unger's modification to this algorithm says that if the machine was in state b in column 001, and the inputs changed from 001 to 000 and then quickly to 010, then the machine could be half-way between states b and c when the inputs became 010. We must therefore be able to tell the difference between any state $x \in [b,c]$ in column 010 and the set of states $[f,g]$ in

[1] Unger's extra dichotomies are not quite enough to make sure that the machine functions correctly; assignments can be constructed that satisfy Unger's conditions but fail on certain combinations of inputs. Unger did publish corrections [179], but this was not one of them.

column 010, so the dichotomy bc/fg must be added. A similar argument adds the dichotomy bd/fg.

However, this is not quite enough. It is not sufficient to be able to tell the difference between any state $x \in [b,c]$ and $[f,g]$, because from every such x, a transition to state e must be made. We must ensure that $[x,e]$ and $[f,g]$ are disjoint for each $x \in [b,c]$. Define $[S,T]$ for sets S and T as:

$$[S,T] = \{\text{states } x \text{ such that } \forall i : (x_i = s_i \text{ for some } s \in S \\ \text{or } x_i = t_i \text{ for some } t \in T)\}$$

so that

$$\bigcup_{x \in [b,c]} [x,e] = [[b,c],e] = [b,[c,e]] \overset{def}{=} [b,c,e]$$

In the example flow table, we must make sure that $[b,c,e]$ is disjoint from $[f,g]$, so we should add the dichotomy bce/fg. A similar argument also adds bde/fg. When creating the encoded flow table, all states in $[b,c,e]$ in column 010 should have their next-state entries set to state e, and the same for all states in $[b,d,e]$. The general algorithm is that for all stable states n and t in a column, and for all multiple input changes $h, i, \ldots n$ and $p, q, \ldots t$ that may enter states n and t, the dichotomy $hi \ldots n/pq \ldots t$ should be added. These dichotomies will cover all the Tracey dichotomies, so the Tracey dichotomies can be left out.

Tracey also gave a second algorithm [176], which produces dichotomies for every column of the form $s_1 s_2 \ldots s_k s/t_1 t_2 \ldots t_l t$, where $s_1, s_2, \ldots s_k$ are all the states which have a next-state entry of s in the column under consideration, and similarly for $t_1 \ldots t_l$. This algorithm is faster than Tracey's first algorithm but may produce an assignment with more state variables. The algorithm proposed here can be seen to be part-way between Tracey's first and second algorithms.

6.3.4 Partial Tracey algorithm

Tracey's algorithm often picks more state variables than approaches that allow a transition to happen in a number of steps, such as Fisher and Wu's algorithm [56] or typical speed-independent approaches. Because of this, an algorithm was written that still allows state variables to be added to an existing assignment, but only adds enough state variables to ensure that all states have a unique encoding, while satisfying as many Tracey constraints as possible. Every Tracey constraint that is satisfied allows a transition of the flow table to happen as a fast, single step, but some transitions may need to be routed through intermediate states in two or more steps.

The algorithm used is similar to Tracey's algorithm. For any two states x and y that have the same values of all outputs, the dichotomy x/y is added to a set $E = \{E_1, \ldots E_{|E|}\}$ of *essential dichotomies*. If an essential dichotomy is not satisfied by the state assignment, then two states will be indistinguishable, which is clearly wrong. The Tracey dichotomies are then calculated as above, and placed in a set $N = \{N_1, \ldots N_{|N|}\}$ of non-essential dichotomies. If a non-essential dichotomy

ab/cd is not satisfied, then one or both of the transitions $a \rightarrow b$ and $c \rightarrow d$ will need to be routed via an intermediate state. Associated with each member of N is a count of how many times it has been seen as a Tracey dichotomy, so that the algorithm can make a decision to route transitions that occur often in a single step at the expense of transitions that only happen infrequently.

The set of maximal dichotomies $M = \{M_1, \ldots M_{|M|}\}$ can be computed as in Tracey's algorithm from both N and E. The task then is to find a minimal set of elements of M such that all members of E are covered, and the maximum number of members of N are covered. The method used was based on a solution for the knapsack problem suggested by Dr. A. C. Norman. Rather than use an exhaustive search or some modification of such a method, Dr. Norman's suggestion is to form $|M|$ solutions $S_1 \ldots S_{|M|}$ by starting off solution S_i by including M_i in the set of maximal dichotomies picked, and then picking further maximal dichotomies in each set by greed. A score function was defined for a set of maximal dichotomies as the number of members of E that were covered times 2^{16}, plus the number of members of N that were covered. The maximal dichotomy which raises the score function most is then repeatedly picked until all members of E are covered, and this set of maximal dichotomies marked as a solution. More members of M are then picked, until the number of maximal dichotomies picked is one less than the number of modified Tracey state variables. The reason for also creating solutions with more than the minimum number of state variables is that it might not be possible to route all transitions in the flow table derived from the minimal solution; having an extra state variable or two might allow all the transitions to be routed properly. If no solutions can be found with less state variables than the modified Tracey algorithm, the MP algorithm in Figure 6.12 is used instead.

The above algorithm produces a large number of possible state assignments, which is also true to a lesser extent of the other two algorithms. Which assignment should be picked is dealt with in the next section.

Creating the encoded flow table is similar to the method used for multiple next-state entries in the Tracey algorithm described above. Whenever two or more state variables need to change during a transition, there is a choice of whether they should change together, if possible, or one at a time. For example, when going from state 00 to 11, the choices are 00→01→11, 00→10→11, or the Tracey-type transition 00→11 with 01→11 and 10→11. When only one variable changes, there is no choice. The algorithm proceeds similarly to the first algorithm: all transitions that have no choices are filled in, Boolean expressions derived, and then a three-step procedure carried out where low-cost routes through the flow table are taken in preference to high-cost routes. If only one route is possible for a transition, then that route is taken as soon as possible to avoid the route being blocked by another transition. It is possible that the encoded table cannot be filled in correctly, in which case the state assignment is rejected.

Scoring function	Cycle time	Size of circuit
19	17.4890 ns	172
20	17.4845 ns	191
21	17.5675 ns	199
22	18.0455 ns	205
22	16.9867 ns	209

Table 6.11: Result of scoring function for state assignments for *isend*

Pruned BD no.	Scoring function	Cycle time	Size of circuit
0	17	19.1455	177
1	20	19.9898	179
1	22	19.7032	198
2	23	21.8089	185
2	29	21.6518	211
3	22	19.7329	184
4	23	19.2386	184
6	23	20.1866	184
7	23	22.3785	190
7	29	22.8651	220
9	19	21.2238	176
9	23	20.6630	198

Table 6.12: Result of scoring function for state assignments, loadable counter

6.3.5 Choosing the best state assignments

All of the above state assignment algorithms can produce several possible state assignments. It may be the case that certain assignments may produce a circuit which fails verification, or the encoded flow table cannot be filled in for the case of the partial Tracey algorithm, so several state assignments may be synthesized, but it would be good to order the assignments so that heuristically good circuits will be produced first. This requires a scoring function for the state assignments. Several scoring criteria were tried, but the results were mixed. The best function was the sum of the number of state variables and outputs that were different between states a and b for all transitions $a \longrightarrow b$ in the flow table. Some results are shown in Tables 6.11 and 6.12, where it can be seen that a low score corresponds to a small implementation, but not necessarily a fast one. These two circuits, *isend* and the loadable counter, were chosen because they had several state assignments with the modified Tracey method, but many circuits for the loadable counter failed verification.

6.4 Converting truth tables to circuits

6.4.1 Derivation of the P and N trees

Section 6.1.2 gave the form of circuits that will be considered, as a CMOS complex gate followed by an inverter, with an optional weak keeper inverter. If the P and N trees of the complex gate are derived separately, then an SOP/SOP gate results as described in Section 2.6.3, which is immune to static hazards. It is important that the P tree and N tree do not both conduct at the same time, partly because this wastes power, but more importantly because the power estimation tools that will be described later will fail to pick this up. It is permissible for neither tree to conduct for some input combinations.

The usual tool for deriving Boolean expressions from truth tables is *espresso* [153], which is freely available from the University of California at Berkeley. One way to find a pair of expressions for the P and N trees from a truth table containing values from $\{0,1,X\}$ is:

- Derive the P tree directly using *espresso*.
- Fill with 1's all entries in the truth table that are covered by the expression for the P tree.
- Invert the truth table and run *espresso* to derive the N tree expression.

Equivalently, the N tree can be derived first and the P tree second. These methods differ, because the second time *espresso* is run, the flow table has fewer don't-care entries so the problem is a little more constrained. Because it is not obvious which order the trees should be derived in, both orderings are tried and the one that produces that smallest gate chosen. When judging the size of the gate, P transistors are assumed to be twice the size of N transistors, although the exact figure used makes little difference.

The *espresso* program produces minimal sum-of-product expressions, such as $ab+acd$, rather than attempting to combine terms into a simplified expression such as $a(b+cd)$. This simplification appears to be usually done with a tool *full-simplify*, but that was not available for download from Berkeley, so a small simplification function was written using a partial search followed by a greedy approach.

An alternative to *expresso* was written, using a modified Quine-McCluskey algorithm [123, 150]. This algorithm is a little faster than *espresso* on the small problems that are encountered in this dissertation, and gives solutions of about the same quality. The speed difference may only be due to the fact that *espresso* is a separate program which needs to be launched as a child process, which takes a comparatively long time. On large problems, the Quine-McCluskey algorithm takes much longer than *espresso*. This algorithm was written so that a few ideas could be tested, such as:

1. Whether deriving the N and P trees simultaneously by constraining the covering problem part of the Quine-McCluskey algorithm would provide a better solutions than separately deriving the P and N trees.

2. Whether combining prime implicants before the covering problem is solved could give a better solution. For example, if *ab* and *acd* were prime implicants of cost 2 and 3 respectively, is it worth combining these into a single PI-like term $a(b + cd)$ of cost 4?

3. Whether giving an n-variable prime implicant a cost of n^2 rather than n would encourage many short transistor stacks, which provide better drive to other gates.

It transpires that none of these ideas gives significant benefits. The first two give gates that are smaller by about 0.5% on average, but increase the run time by a factor of 20 to 100 times. The third modification, interestingly, has absolutely no effect; it seems that a minimal solution is still a minimal solution under any reasonable cost function. Using n^3 or even 2^n as a cost function also makes no difference.

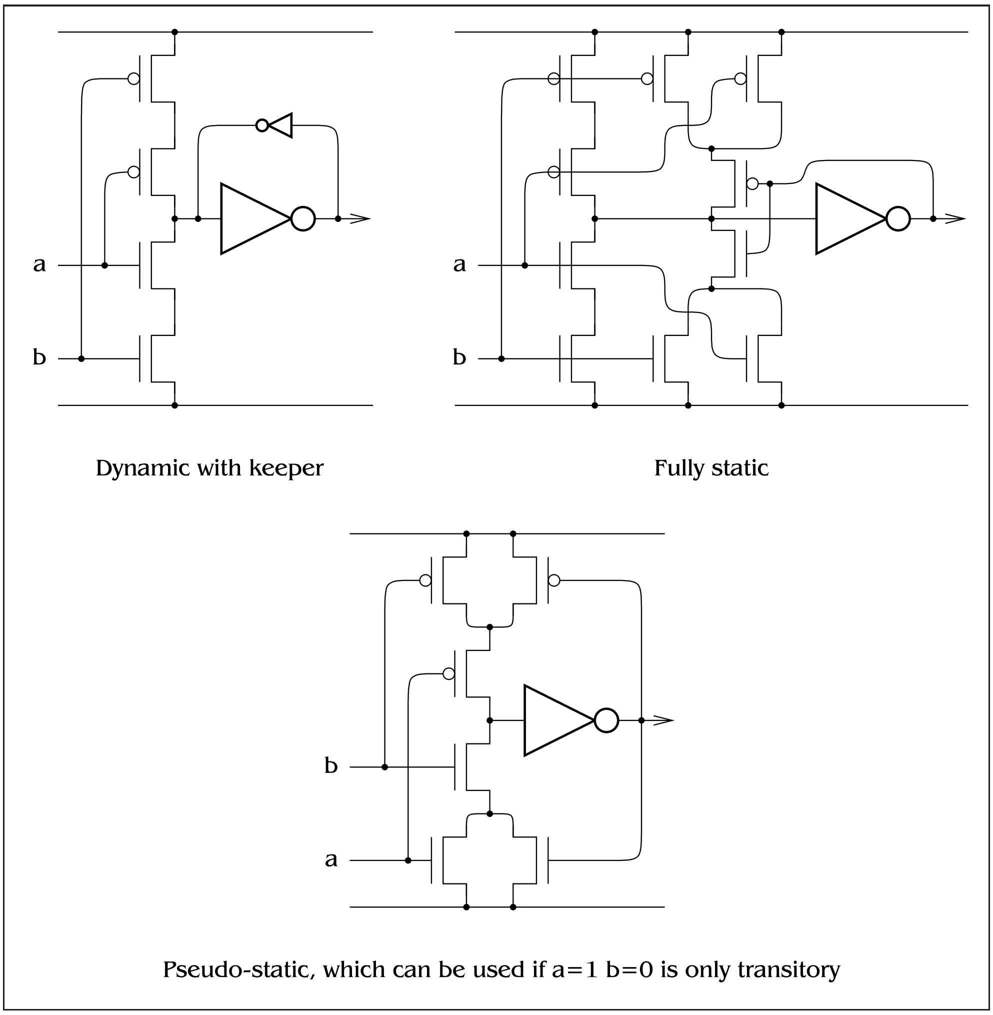
6.4.2 Types of gate created

As was noted in Chapter 2, two types of CMOS gates are typically used by asynchronous tools: dynamic gates with keeper inverters, or fully static gates. An example of a two-input C-element implemented as each kind of gate is shown at the top left and top right of Figure 6.13. Although the dynamic version looks smaller and probably faster, the weak inverter is quite large and slows the circuit down, so there is not much to pick between the two circuits. In practice, for small circuits the static version is the fastest and smallest, but for large gates with many inputs, the dynamic version is smaller and faster.

Consider what would happen if it was known that the input combination $a = 1$, $b = 0$ to a two-input C-element would only persist for a small number of gate delays. Such conditions can occur when a and b are state variables internal to the circuit, and the only time that $a = 1$ and $b = 0$ is during a state transition $ab = 00 \rightarrow 10 \rightarrow 11$. In this case, the implementation at the bottom of Figure 6.13 could be used, which is much faster and smaller than the other two circuits. Such gates can be described as *pseudo-static*, because they only rely on dynamic charge retention for a few gate delays, and never in a stable state of the circuit.

Creating pseudo-static gates requires that the truth tables take values from the five-membered set $\{\mathbf{0}, 0, X, 1, \mathbf{1}\}$, where **1** is a *strong logic one* and 1 is a *weak logic one*, similarly for **0** and 0. Strong logic values are those that must be driven, such as stable states of the circuit and when a state variable is changing its value. Weak values are where the circuit is changing rapidly, so dynamic charge retention will be sufficient to hold the value. An example is the three-step transition $\mathbf{0001} \rightarrow \mathbf{1}001 \rightarrow 1\mathbf{1}01 \rightarrow \mathbf{1111}$, where the stable start and final states are strong values, as are any variables that are changing state. When creating P and N trees for a pseudo-static gate, the five-valued truth tables are translated to ordinary three-value tables as in Table 6.13, where is it assumed that the output of the complex gate is inverted.

Pseudo-static gates can be viewed as being between dynamic gates with keepers and fully static gates, as Table 6.14 shows. Because the P and N trees are derived



Dynamic with keeper

Fully static

Pseudo-static, which can be used if a=1 b=0 is only transitory

Figure 6.13: Three ways of implementing a C-element

Symbol	State of P-tree	State of N-tree
1 = Strong 1	off	on
1 = Weak 1	off	don't care
X = Don't care	don't care	don't care
	(But not both on)	
0 = Weak 0	don't care	off
0 = Strong 0	on	off

Table 6.13: The meaning of strong and weak values at the transistor level

Type of gate	Output defined for				Needs keeper
	places where the variable changes	stable states	reachable states	unreachable states	
Static (s)	Yes	Yes	Yes	Yes	
Not quite static (s2)	Yes	Yes	Yes		
Pseudo-static (ps)	Yes	Yes			
Dynamic (d)	Yes				Yes

Table 6.14: Comparison of static, pseudo-static and dynamic gates

separately for a gate, there may be input combinations that do not cause the output to be driven, producing what is labelled in Table 6.14 as an *s2* gate: one that is not static, but behaves as a static gate for all input combinations that are not don't-cares. Some results obtained by using each of the four kinds of gates *s*, *s2*, *ps* and *d* will be given in Chapter 8.

When creating a dynamic gate with keeper, synth always checks to see whether the resulting gate is actually static, and if so removes the keeper inverter. A similar degenerate case is when the complex gate turns out to be just an inverter; in this case, the combination of the complex gate followed by an inverter can be replaced with a wire. This only happens if a flag is set in synth, because the implementation will then be able to respond in zero time, which may break timing assumptions in other modules.

Currently, the choice of which type of gate to use is left up to the designer. A possible extension is to employ a heuristic that can determine whether a pseudo-static or dynamic gate with keeper would be best in a certain circuit, by looking at both and comparing their size and speed.

6.4.3 Other considerations

Isolation from outside effects

The MEAT design style [34] attempted to keep wire forks away from the interfaces between modules, by placing buffers before and after modules. If a wire is forked on entry to a module, then a slow ramp voltage on that input could cause gates to switch at noticeably different times, causing the circuit to fail. If a wire at the output of a module is forked back into the module, then a large load on that output may cause the output to be a slow ramp, causing the same kind of problem. Whenever slow ramps or large loads may be encountered, inverters should be placed before all inputs and after all outputs. When very slow input ramps could occur, Schmitt trigger inverters could be used. This is a trivial modification to the synthesis algorithm, because it simply inverts all input and all outputs.

Transistor sizing

Transistor sizing is something that is often done at the end of the design process, using simulators such as SPICE to shave a few percent off the delays in a circuit. A method to create a good first approximation to an optimal sizing of transistors, called Logical Effort, was given by Sproull and Sutherland [169]. This requires using several different sizes of transistors in a single gate, which will render the timing algorithms that will be described in the next chapter almost useless. The timing algorithms rely on a large number of gates looking relatively similar, so the same timing information can be used for many gates; this is made much easier by building all gates with a certain size of P and N transistor.

However, it is possible to make the inverter that follows the complex gate (see Figure 6.3) a different size from every other gate, because this will only require the timing simulator to learn about one new kind of gate. In SPICE trials on small circuits, it was found that double-size inverters gave the lowest cycle times, producing circuits that were about 10% faster than using single-size inverters. Only another 2% was gained by using Logical Effort in the examples tried, although this is almost certainly a best-case result for my approach; on more complex circuits, Logical Effort could probably give more substantial improvements.

Although synth does not use Logical Effort or other transistor sizing algorithms for every circuit before timing simulation, there is no reason why the final implementation cannot be sized correctly.

Timing and Verification 7

One . . . two . . . Wait!
On three, or after three?
– Murtaugh, *Lethal Weapon 3*

I find your lack of faith disturbing.
– Darth Vader, *Star Wars*

Abstract

This chapter discusses previous approaches to gate-level timing, and finds that these will be either too slow or too inaccurate to be used in this dissertation. A new model of slope waveforms in circuits is presented, which allows an accurate gate-level simulator to be written. This is then combined with previous work on binary bi-bounded simulation to produce a technology-specific verification algorithm.

Structure of this chapter

Previous timing and simulation strategies are described in Section 7.1. Section 7.2 compares the accuracy of these strategies with a new method proposed in this dissertation. The new method is used in a timing simulator described in Section 7.3. Finally, verification of circuits is described in Section 7.4.

7.1 Previous timing strategies

This section gives a quick overview of techniques that have been used for timing of CMOS circuits. It is not exhaustive, and probably could never be; a good timing algorithm would have great commercial advantages, and is unlikely to be documented in academic journals.

7.1.1 Analogue simulators

The earliest predictors of circuit timing information were the analogue simulators, such as SPICE [138] and ASTAP [187]. These programs numerically solve systems of differential equations expressing the voltages and currents in the test circuit, and take a long amount of time for circuits of only a few hundred transistors. Despite this disadvantage, SPICE has become almost ubiquitous, and it even now used in preference to other simulators even though it is over twenty years old. This popularity is due in part to the fact that the transistor models in SPICE are very detailed, so accurate technology files can be produced for any given fabrication process. SPICE is so well-trusted that other simulators are usually judged by how close their predictions are to those of SPICE.

Several methods have been recently tried to speed up the execution of analogue simulators. Ruan et al. [156] used piecewise constant I-V curves for all circuit elements. Piecewise constant currents imply piecewise linear voltages, so the simulation becomes more event-based in character, because it is easy to tell when a linearly-changing voltage will exceed a particular value. Devgan [50] proposed the use of charge-voltage simulation rather than current-voltage, which can speed up the inner loop of SPICE by a factor of two. Other improvements and approximations were also suggested that gave signification speed-ups.

7.1.2 Event simulators

Analogue simulators were found to be too slow for tasks such as transistor size optimization, where repeated re-simulation of a circuit is required. Fast methods of estimating gate delays had to be found. Some simulators, such as DIANA and SPLICE, combined the old analogue techniques for critical paths with newer, event-based methods for all other gates, but these are not really fast enough for optimization problems. Rather than the transistor-level approach of SPICE, methods were developed that consider gates to be atomic units in a circuit, a practice known as macromodelling.

Three main types of delay estimation can be identified, based on what the model assumed about the effects of the input waveform to a gate:

- Assume the input waveform can be ignored. This was the approach taken by Berkelaar et al. [13], because they needed linear equations for the power vs. delay optimization that they were doing. They used the formula $\tau_{gate} = \tau_{int} + c.C_{load}$, expressing the gate delay solely in terms of the internal no-load delay τ_{int} plus a constant times the load capacitance.

- Assume the input waveform depends only on a property of the previous gate. Hedenstierna et al. [72] made several simplifying assumptions to the equations governing a CMOS inverter, such as assuming the input was a linear ramp and that a primitive MOSFET model could be used. They found that the delay t_{ds} of an inverter is equal to $t_{ds} + t_{di}$, where t_{ds} is the step-input delay of the inverter, and t_{di} is proportional to t_{ds-1}, the step-input delay of the previous gate. This model was used by Hoppe et al. [75] to optimize circuits for power and speed, and extended by Hallam et al. [68] by making the dependence on t_{ds-1} linear rather than proportional.

- Assume the input waveform can be approximated by a particular curve, and parameterize the gate behaviour in terms of the rise time of this curve. Typical curves are a linear ramp through the 10% and 90% points [185, 196], or through the 20% and 80% points [17, 81]. Other curves have also been used, such as exponentials [17] and linear ramps with exponential tails [122].

Of the three approaches, the last seems to be capable of giving the best results. Again, approaches to delay determination using slope-based methods can be split into a number of categories:

- Yang and Holburn [196] used a semi-analytic method to determine gate delay. The input waveform could be of three types: fast, so that the input waveform did not affect gate delay or the output waveform, hence an RC delay approximation could be used; slow, where a direct solution of the transfer equation of the gate could be used; or intermediate, which used parts of both other solutions. Vemuru and Smith [185] later split the slowest classification into two, improving accuracy.

- Brocco et al. [17] measured the input slope vs. output slope and gate delay characteristics, and found that the graphs were fitted well by two and three piece linear approximations. A set of approximations is stored for each different kind of gate that is used in a circuit. Very similar graphs were also given by Compass Design Automation, when talking about their new commercial simulator [36].

- Jun, Jun and Park [81] gave a method for estimating the delay of all kinds of gates with only one set of parameters. They found that the graphs of gate delay τ_d vs. input slope τ_{in} were almost linear, so assumed that $\tau_d = a\tau_{in}+b$, and then expressed a and b as quadratics in the output load C_{out} and the effective width of the conduction path inside the gate β. This gave a set of 60 parameters that could be used to determine the delay of any gate.

It is difficult to compare accuracy of these methods; even though most were compared to SPICE simulations, the choice of MOSFET model will affect the validity of the results. An example is the work of Hedenstierna et al. [72], where good agreement with SPICE results was reported. The only SPICE model available at the time was the primitive level 1 model; these same Shichman-Hodges equations were the ones that were used by Hedenstierna et al. in their work, so it is not surprising that their results were close to SPICE. The level 6 SPICE model will be used in this chapter for all simulations, which is a model applicable to submicron and deep submicron transistors, and is described in [159]. This may give results that are different from the papers listed above, which were mostly written before 1990, when the level 5 (BSIM2) and level 6 SPICE models became available.

7.2 Development of an accurate timing model

This section covers the development of the gate-level timing approach that is used in this dissertation. Accuracy is paramount, because a large number of circuits must be timed, with small variations in delays between them. Section 7.2.1 looks at the different methods of estimating the effects of the input waveform slope on the delay of a gate, to determine whether any existing methods will be sufficiently accurate.

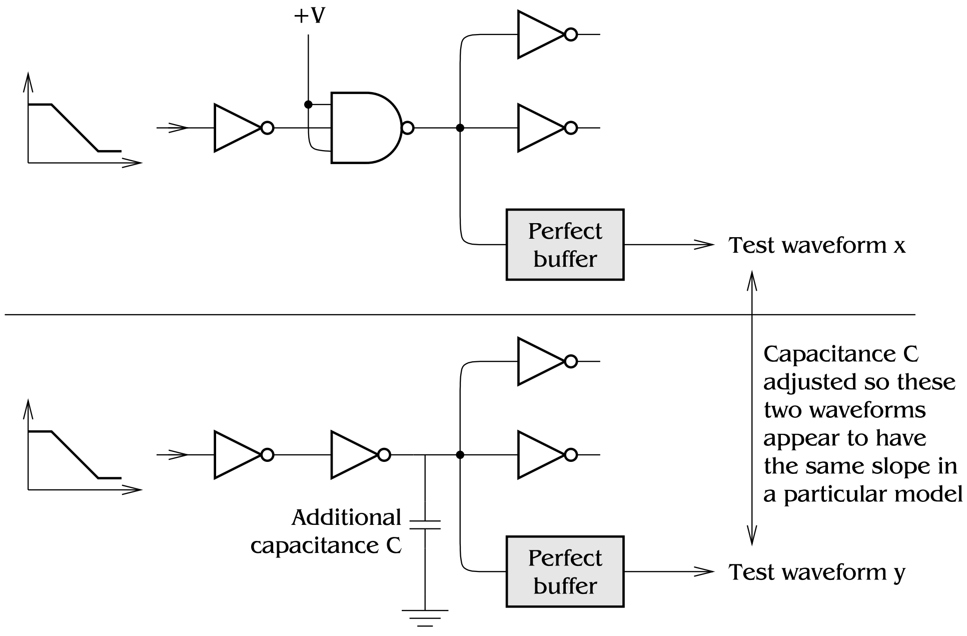
7.2.1 Evaluation of input slope models

The delay of a CMOS gate depends on its geometry, the load on its output, and the waveform which it receives from the previous gate. Any effective gate-level simulator must have some way of determining the slope of the input waveform, and determining its effects on the gate delay and output slope. Four ways of measuring the input slope were found in the literature:

1. Assume all necessary knowledge about the input slope is captured by the previous gate's step-input delay [68, 72, 75].
2. Measure the time between the 10% and 90% points of the input waveform, and use that as the slope [185, 196].
3. Measure the time between the 20% and 80% points of the input waveform, and use that as the slope [17, 81].
4. Do a least-squares fit of a linear waveform with an exponential tail [122].

Each of these methods attempts to capture the important properties of a waveform in a single real number, so the effectiveness of each method will ultimately depend on how much information is not reflected in this number. Consider a falling waveform x produced by a 3-input NAND gate, which has three N-transistors in the conducting path. The gate is loaded by two inverters, and driven from an inverter which itself has a linear ramp voltage applied; this is shown at the top of Figure 7.1. Each of the methods 1–4 will be able to assign a particular slope s_x to waveform x. Now consider replacing the NAND gate by an inverter, producing waveform y, but adding an additional capacitative load so that the slope s_y of this waveform is the same as the original slope s_x of the NAND gate. This is illustrated at the bottom of Figure 7.1. If it was indeed the case that the slope values s_x and s_y captured all the relevant information about the waveforms x and y, then any gate G driven by waveform x should have exactly the same delay G_x as the delay G_y when it is driven by y. By measuring the difference between G_x and G_y for a variety of gates G and loading conditions on the output of G, it should be possible to see how good each of the methods 1–4 are.

Two further methods will also be compared, which I believe are new to this dissertation. When macromodelling gates, a simulator does not need to *create* a waveform with a given slope; it is only necessary to be able to *measure* a slope parameter from a waveform. A method is needed that will derive a single number from a waveform, that gives a maximal amount of information about what effects that waveform will have on the delay on the gate it is driving. An ideal measuring device would therefore seem to be a CMOS gate, giving the following method for determining the slope of a waveform:


+V
Perfect buffer
Test waveform x
Capacitance C adjusted so these two waveforms appear to have the same slope in a particular model
Additional capacitance C
Perfect buffer
Test waveform y

The box labelled "perfect buffer" is a SPICE model for unity-gain voltage controlled voltage source. It copies the voltage on its input, presenting no load to the NAND gate or inverter.

Figure 7.1: NAND gate and inverter used to produce test waveforms

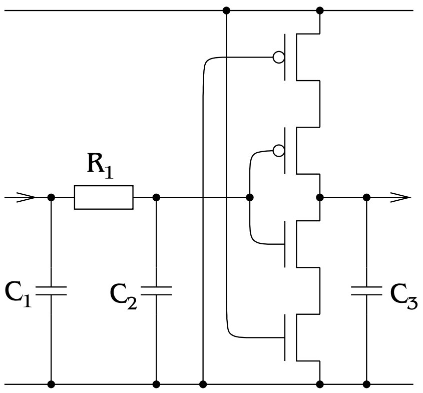
5. Feed the waveform into an inverter I loaded with two other inverters, and take the delay of I as an indication of the slope of the waveform.

It was later found that an inverter is a particularly atypical gate, being the smallest possible CMOS gate, so the gate shown in Figure 7.2 was used instead, leading to method 6:

6. Feed the waveform into T, the typical gate of Figure 7.2, loaded with two other such gates, and take the delay of T as an indication of the slope of the waveform.

To compare these six methods, each must be used to measure the output slope s_x of the NAND gate in Figure 7.1, then a linear search or interval bisection used to determine the value of C in the lower half of Figure 7.1 that makes the slope s_y equal to s_x. The measured values of s_x, initial value of s_y with $C = 0$ and required values of C are given in Table 7.1.

Next, waveforms x and y must be fed into some example gates with different loadings, and the delays of these gates measured and compared. The gates that were be used are given in Figure 7.3; they were each be loaded by one, three and five inverters. For each of gates A–D and each load, Table 7.2 gives the delay G_x of the test gate when driven with waveform x, and for each method 1–6, the difference between G_x and the G_y. It can be seen that errors of 7–12% are inevitable



This gate is equivalent to a tri-state inverter in the enabled state. All gates in this test have a small length of polysilicon attached to their inputs, modelled here by the π-network R_1, C_1 and C_2, and a small length of metal1 on their output, modelled by a pure capacitance C_3. Both lengths were 20 times the gate width, assuming a particular set of process parameters.

Figure 7.2: A more typical gate than an inverter

	Value on NAND s_x	Value on inverter with C=0, s_y	Additional capacitance C needed to make $s_x = s_y$
Method 1 (previous gate delay)	0.57279 ns	0.23006 ns	37.151 fF
Method 2 ($t_{90\%} - t_{10\%}$)	1.31403 ns	0.46932 ns	45.637 fF
Method 3 ($t_{80\%} - t_{20\%}$)	0.91976 ns	0.33579 ns	47.408 fF
Method 4 (fit linear/exp tail)	0.85538 ns	0.30654 ns	45.169 fF
Method 5 (delay of inverter)	0.48941 ns	0.29594 ns	68.699 fF
Method 6 (delay of typical gate)	0.66273 ns	0.44229 ns	65.841 fF

Table 7.1: Additional capacitance required to make $s_x = s_y$ for methods 1–6

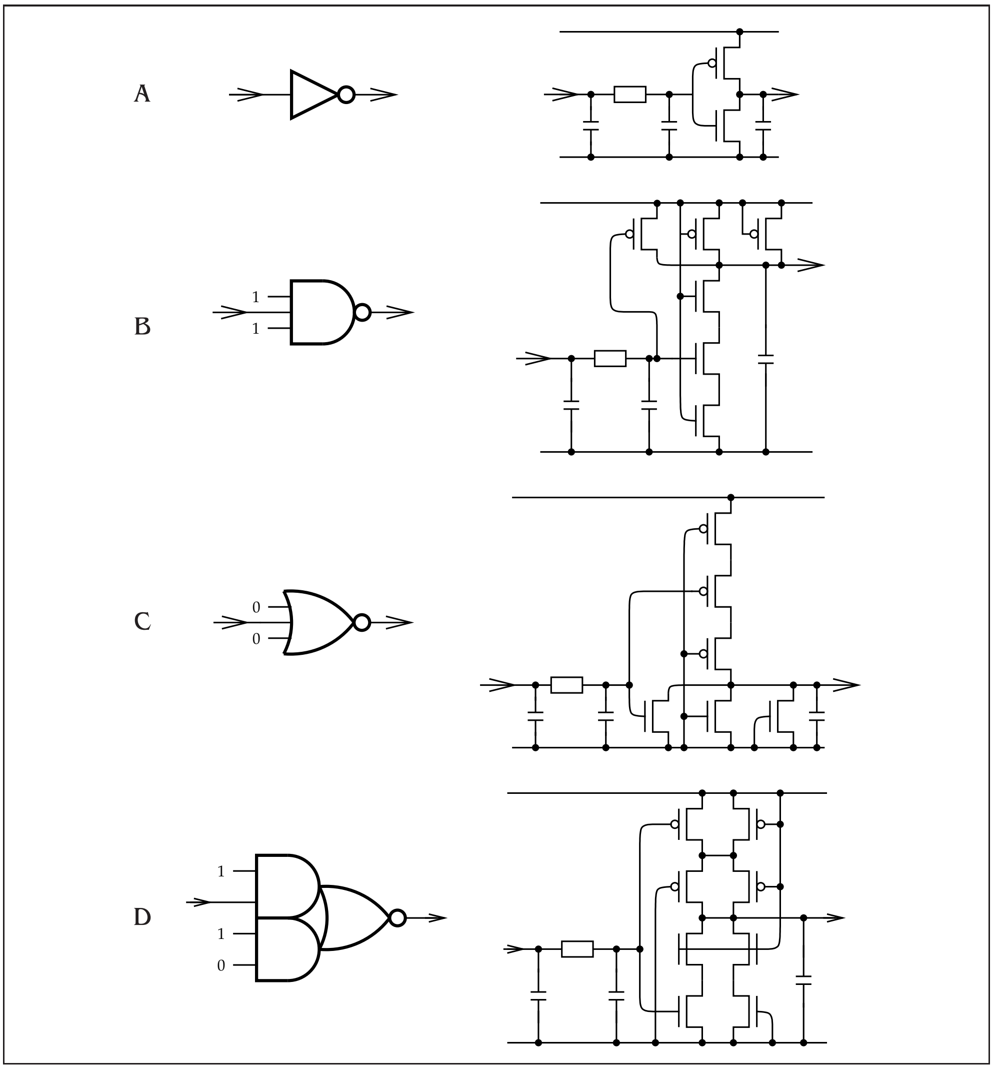


Figure 7.3: Four example gates used and their circuits

with the existing methods, but the two proposed methods for measuring waveform slopes have errors of around 2%. These figures give an indication of the possible accuracy of a gate-level simulator based upon each of the six methods, so imply that a simulator using methods 5 or 6 would perform significantly better than one using methods 1–4. Method 6 was used for the simulator described in this chapter.

7.2.2 Effects of discrete gate modelling

Any gate-level simulator assumes that a circuit can be split up into its constituent gates, and when timing algorithms are applied to these gates in isolation, the resulting gate delays are approximately equal to the gate delays in the original circuit. This section will take an example circuit and break it up into single-gate pieces,

Gate &		Actual	Method 1	Method 2	Method 3	Method 4	Method 5	Method 6
Load		delay	prev gate	10%–90%	20%–80%	lin/exp	inverter	typ. gate
A	1	364.65 ps	−17.0 %	−12.5 %	−11.6 %	−12.7 %	−1.1 %	−2.5 %
	3	574.27 ps	−14.6 %	−10.1 %	−9.2 %	−10.4 %	+0.9 %	−0.4 %
	5	731.52 ps	−12.8 %	−8.4 %	−7.6 %	−8.7 %	+2.4 %	+1.2 %
B	1	375.95 ps	−13.2 %	−10.8 %	−10.3 %	−10.9 %	−5.8 %	−6.3 %
	3	583.67 ps	−12.5 %	−9.1 %	−8.4 %	−9.3 %	−1.4 %	−2.3 %
	5	743.71 ps	−11.3 %	−7.7 %	−7.0 %	−7.9 %	0.9 %	−0.1 %
C	1	662.79 ps	−14.5 %	−9.8 %	−8.9 %	−10.1 %	+2.3 %	+0.9 %
	3	996.99 ps	−9.5 %	−6.3 %	−5.7 %	−6.5 %	+2.5 %	+1.4 %
	5	1 333.08 ps	−6.9 %	−4.6 %	−4.1 %	−4.7 %	+1.9 %	+1.1 %
D	1	651.66 ps	−14.5 %	−9.8 %	−8.8 %	−10.1 %	+2.2 %	+0.8 %
	3	903.45 ps	−11.1 %	−7.1 %	−6.3 %	−7.4 %	+3.3 %	+2.0 %
	5	1 143.67 ps	−9.0 %	−5.6 %	−4.9 %	−5.8 %	+3.2 %	+2.2 %
Average magnitude of error			**12.2 %**	**8.5 %**	**7.7 %**	**8.7 %**	**2.3 %**	**1.8 %**

Table 7.2: Discrepancies between gate delays when driven by "identical" waveforms

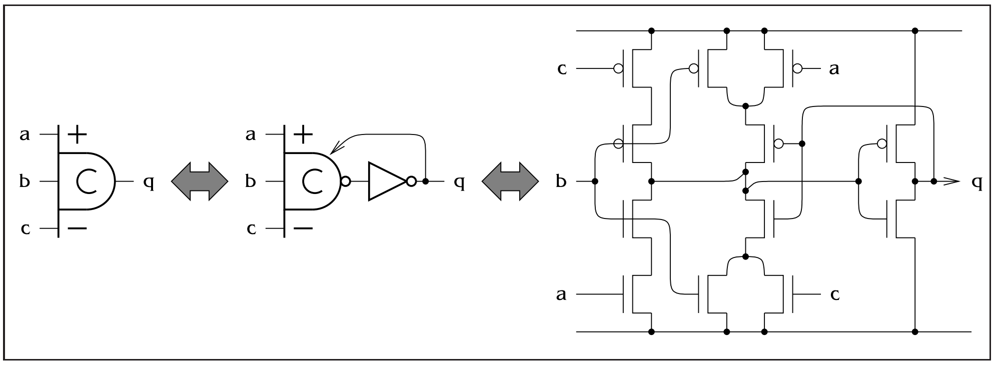


Figure 7.4: Static C-element symbol that will be used, and a CMOS implementation

looking at the errors that are introduced along the way.

The example circuit used in this section is the Furber/Day fully decoupled latch controller [59], with some modifications:

- Fully static C-elements are used, such as that shown in Figure 7.4. This is partly because Davis suggested that fully static gates are faster than dynamic ones [45], but mostly because I thought that dynamic gates with keeper inverters would be harder to handle in a gate-level simulator. It turns out that the theory presented here also works well with dynamic gates.

- Inverted outputs from a C-element are taken from the output of the combinational part of the C-element, rather than hanging another inverter off the output inverter. This is merely to give a more interesting example.

A somewhat artificial set of inputs is used to get a transition that will pass

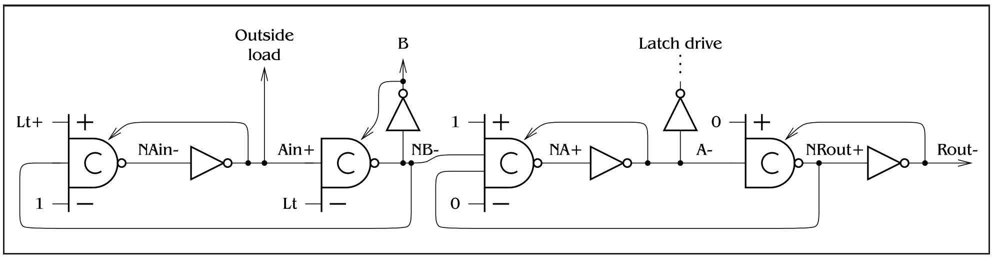


Figure 7.5: Example circuit from [59], redrawn to highlight interesting transitions

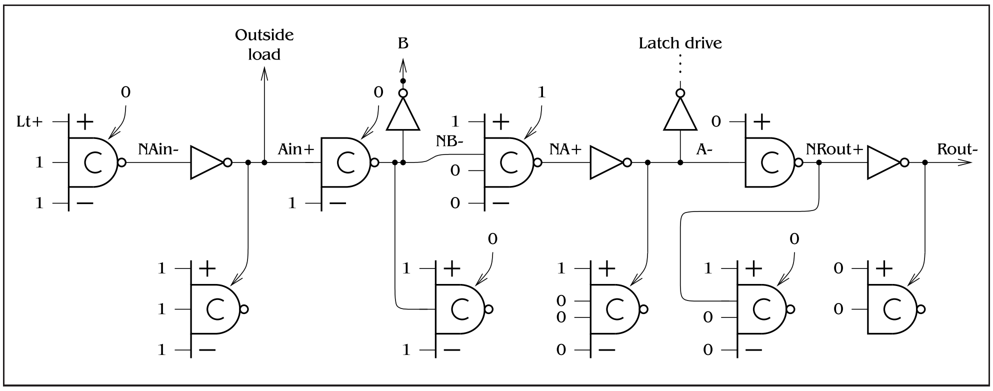


Figure 7.6: Straight-line version of Figure 7.5

through all four C-elements, although this set of transitions can occur in practice: Rin and Aout are held at logic 1, while a rising edge is observed leaving the latch drive buffer at Lt. The circuit from [59] is redrawn in Figure 7.5, highlighting the following set of transitions that will occur: Lt+ → NAin- → Ain+ → NB- → NA+ → A- → NRout+ → Rout-. NAin refers to the signal on the internal node of the C-element that produces the Ain signal.

The first step towards isolating the gates in this example is to turn the circuit into a linear sequence of gates, each of which drives the next gate plus some additional load. This could be viewed as "straightening" the circuit. In Figure 7.5, the gate that creates the NB signal drives the next gate in the chain, and an inverter, but it also drives an earlier gate that creates the NAin signal. In the straight-line version of this circuit, shown in Figure 7.6, the NB gate now drives a copy of the NAin gate instead. It would be expected that the gate delays in the straight-line version would be almost the same as in the original; after all, every time a gate switches, it has the same values on all inputs in both versions of the circuit, and it drives the same gates again with the same values on all inputs. Table 7.3 gives the gate delays in both versions, where it can be seen that this is not the case. Gate delays vary by an average of 7 %, but the total delay through the circuit is only out by 1.5 %. This pattern has been observed on several test circuits. This result indicates that individual gate delays may be highly inaccurate in a gate-level simulator, although the errors tend to cancel in long chains of gates, giving a reasonable accurate final result.

	Delay in original	Delay in straight-line	Difference
Lt+ → NAin-	246 ps	254 ps	+3.0 %
NAin- → Ain+	335 ps	302 ps	−9.6 %
Ain+ → NB-	399 ps	419 ps	+5.1 %
NB- → NA+	488 ps	554 ps	+13.5 %
NA+ → A-	377 ps	350 ps	−7.3 %
A- → NRout+	304 ps	317 ps	+4.3 %
NRout+ → Rout-	240 ps	227 ps	−5.4 %
Total	2 390 ps	2 425 ps	+1.5 %

Table 7.3: Effect of straightening the example circuit

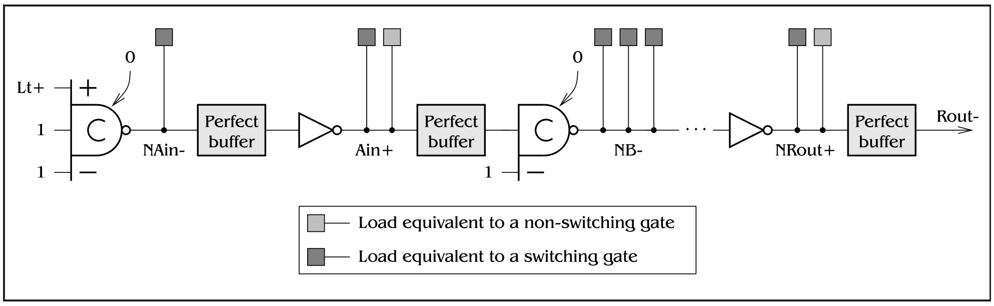


Figure 7.7: Example circuit broken up by perfect buffers

When the circuit has been straightened out, then the second step can occur: treating each gate in isolation. Each gate will have a known input waveform and a known loading. At this stage, the load on each gate will be approximated, but the input waveform will be an exact copy of the output of the previous gate. This is illustrated in Figure 7.7, where each gate has been separated from the adjacent gates by a perfect buffer, and a load added on the output of the gate equivalent to the load it had before. Two kinds of load on a gate G were identified: gates that will switch depending on the output of G, and ones that will not. Gates that switch present a higher load on their input than ones that do not, because of the Miller effect [76, page 102]. The effects of replacing each kind of load with either an inverter, the typical gate used before, or a pair of P and N transistors simulating a non-switching gate are given in Table 7.7. These results show that it is not worth making the distinction between switching and non-switching loads. Earlier—probably less trustworthy—results showed that it was worth making this distinction, so the timing models used in the synth program were written to expect separate parameters for switching and non-switching loads, and it is that method which will be described in this chapter. Having two types of load makes the procedure only slightly more complicated, and makes little difference to the results.

To summarize this section, circuits experience large errors in individual gate delays when they are broken up into constituent gates, but these errors tend to cancel out in a long chain of gates. In the example circuit, errors occur of up to 13 % in individual gate delays, but the errors in the delay of the chain of gates are

Gate substituted for a switching load ■—	Gate substituted for a non-switching load □—	Delay through the chain of gates	Error relative to original circuit
I	I	2 664 ps	11.5 %
I	P	2 568 ps	7.4 %
T	P	2 372 ps	0.8 %
P	P	2 407 ps	0.7 %

I: Inverter
T: Typical gate of Figure 7.2
P: Pair of transistors with their source and drain connected, representing a non-switching gate, as shown on the right

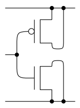


Table 7.4: Effects of different substitute gates on the delay of the example circuit

1.5 % and 0.7 % for the two stages described, which partly cancel to give an overall error of 0.8 %. This implies that any gate-level simulator can only hope to get to within about 2 % of SPICE; of course, SPICE is not accurate to 2 %, so the distinction is academic.

7.2.3 Estimating gate delays

The next task is to produce a method which can calculate the delay d of a gate G and its output slope s_o, when given the input slope s_i, measured as in method 6 of Section 7.2.1, and the load l on the gate. The method used was to find a number of basis functions $f_n(l, s_i)$, and to express the delay for a particular gate as a sum $\sum_n a_n f_n(l, s_i)$ for a set of constants a_n, which are determined from SPICE runs. This is similar to the work by Jun, Jun and Park [81], except that they used quadratics rather than general basis functions, and they found a single set of constants for all gates, whereas this work finds these constants on a per-gate basis.

To determine the dependence of the gate delay and output slope on the load and input slope applied to the gate, SPICE simulations of the four example gates in Figure 7.3 were carried out, with a variety of input slope and load conditions. The resulting graphs of gate delay and output slope against input slope and load are shown in Figures 7.8–7.11.

The curve for gate delay as a function of the load on the gate (Figure 7.8) looks linear, but the worst-case error in fitting a straight line is 2.5% with an RMS error of 0.82%. This worst-case error is a little higher than was hoped for, so a linear approximation will not be quite good enough. A quadratic approximation could be used, as in the work of Jun et al. [81], giving an RMS error of 0.25%; the problem with this is that the delay against load curve will tend towards a straight line as the load gets large, but a quadratic will not show this behaviour. A better approximation would be a linear expression plus something that tends to zero as the load get big, such as $\frac{1}{load+k}$ for some constant k. Choosing k arbitrarily to be 2 gives an RMS error of 0.11% when fitting the delay against load curve, and gives the right behaviour for large loads. This argument can also be applied to the graph of output slope against

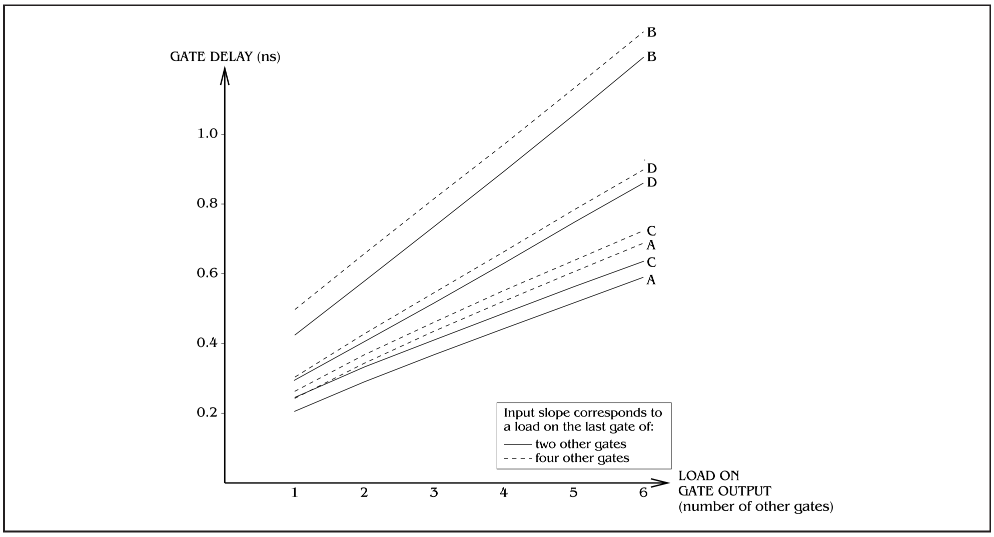


Figure 7.8: Graph of gate delay against output load

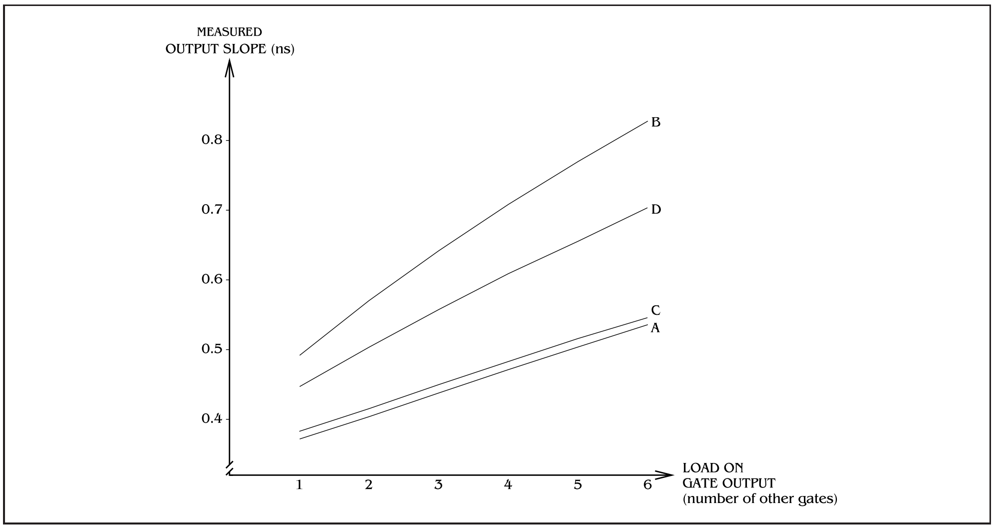


Figure 7.9: Graph of output slope against output load

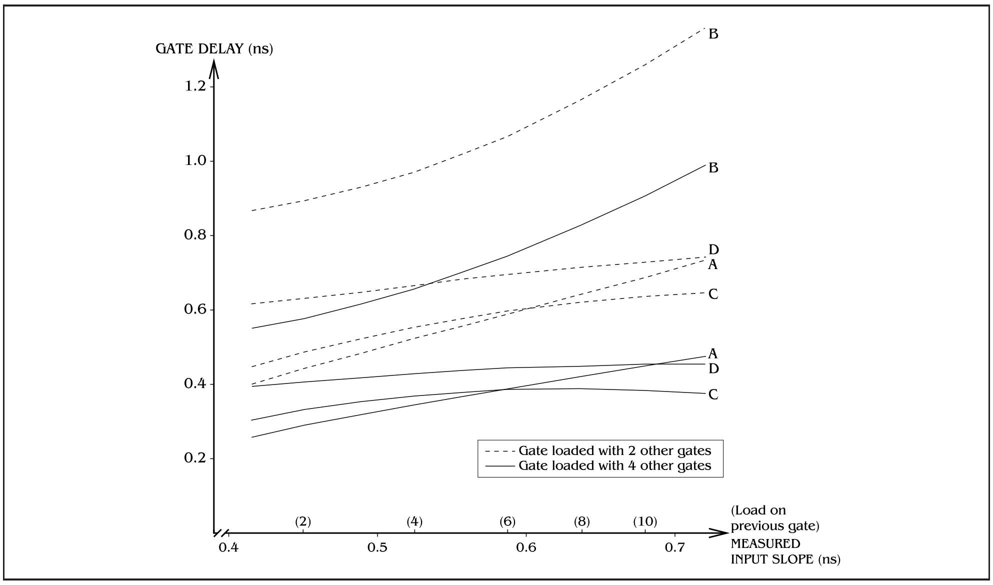


Figure 7.10: Graph of gate delay against input slope

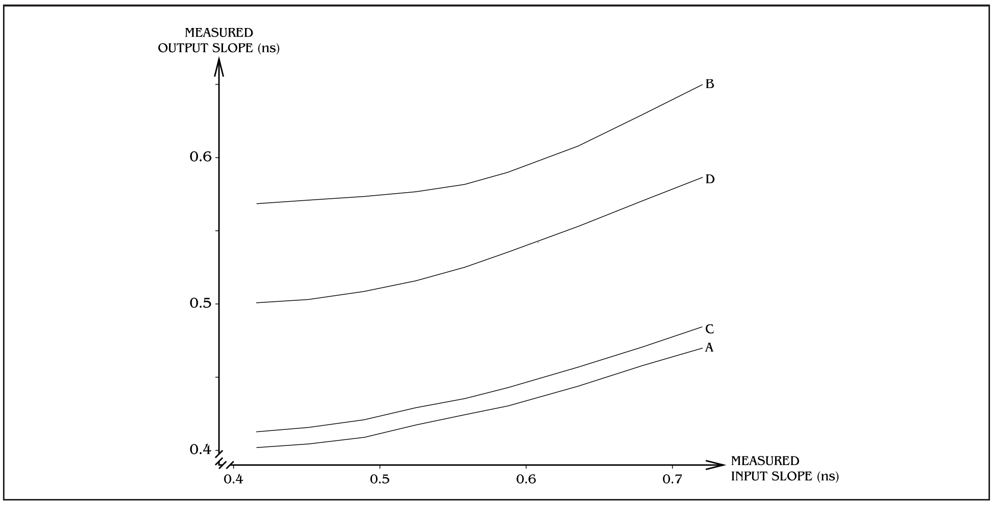


Figure 7.11: Graph of output slope against input slope

load, shown in Figure 7.9.

The graph of output slope against input slope, shown in Figure 7.11, appears to fit the same argument as above; it appears to be linear for large values of the input slope, with a correction to be made for small slope values. The RMS error in Figure 7.11 is smaller when a function of the form $\alpha + \beta s_i + \gamma \frac{1}{s_i + \lambda}$ is used than if a quadratic was used, where λ is a fixed parameter about the same size as a typical gate delay.

The graph of gate delay against input slope does not fit this argument: Figure 7.10 does not appear to be linear for large input slopes. Figure 7.12 shows

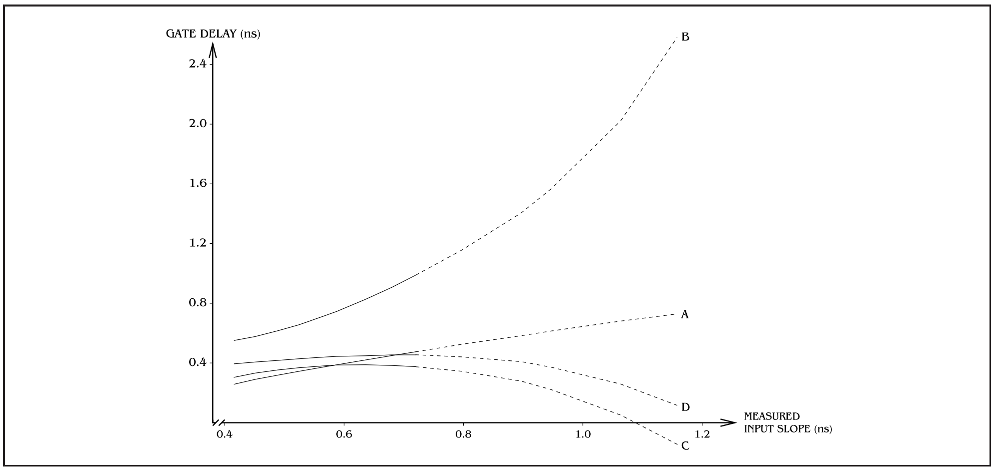


Figure 7.12: Graph of gate delay against extreme values of input slope

the effects on the gate delay of extreme values of the input slope, far greater than will be encountered in normal circuits; even at huge values of the input slope, the curve does not become linear. The argument above breaks down for this curve, but it happens that fitting a curve of the form $\alpha + \beta s_i + \gamma \frac{1}{s_i+\lambda}$ gives a slightly better RMS error on moderate values of the input slope, 0.20%, than fitting a quadratic, 0.21%.

The obvious step now is to take the approximating basis functions for the input slope s_i and load on the gate l, and form a product basis of nine functions:

Basis functions for l: $\quad 1 \quad l \quad \frac{1}{l+k}$

Basis functions for s_i: $\quad 1 \quad l \quad \frac{1}{s_i+\lambda}$

$$\text{Basis functions for } (l{,}s_i)\text{:} \quad \left\{ \begin{array}{ccc} 1 & l & \frac{1}{l+k} \\ s_i & ls_i & \frac{s_i}{l+k}\dagger \\ \frac{1}{s_i+\lambda} & \frac{l}{s_i+\lambda}\dagger & \frac{1}{(s_i+\lambda)(l+k)}\dagger \end{array} \right.$$

By doing a least-squares fit of these basis functions to a number of SPICE results, it was found that the terms marked with a dagger were very small, and when they were left out, the difference in predicted gate delays changed by a tiny amount, less than 0.1%. The other terms could not be neglected.

Up to now, switching and non-switching loads on gates have been treated the same, because the graphs of output slope and gate delay against load would look roughly the same regardless of what kind of load was used. To allow for separate l_{sw} and l_{nsw} parameters, representing the switching and non-switching loads respectively, the l and ls_i basis functions were split into two. The $\frac{1}{l+k}$ term is small, so it was left alone. This gives the final set of basis functions used, where $k = 2$ and λ is a typical gate delay:

$$\{1, l_{sw}, l_{nsw}, s_i, l_{sw}s_i, l_{nsw}s_i, \frac{1}{l_{sw}+l_{nsw}+\lambda}, \frac{1}{s_i+k}\}$$

Eight basis functions f_n require eight sample points from SPICE runs to determine the constants a_n. Each sample consists of running a SPICE simulation of the gate G with a given value of the parameters l_{sw}, l_{nsw} and s_i. The value of s_i cannot be affected directly, but instead it is produced by varying the load l_{prev} on the previous gate to the one under test, and measuring the value of s_i from the SPICE run. It would be good to use typical values of the three parameters, such as $l_{sw} \approx 1$, $l_{nsw} \approx 2$ and therefore $l_{prev} \approx 3$. The average value of l_{sw} varies between about 0.9 and 1.2 depending on the complexity of the specification, so 1.0 is a good estimate. The average value of l_{nsw} varies from about 0.9 to 2.5, so the choice was more arbitrary for l_{nsw}. Many obvious or symmetric choices of the eight sample points lead to singular matrices when trying to find the a_n constants, so the eight points were found essentially by trial and error. The points used were:

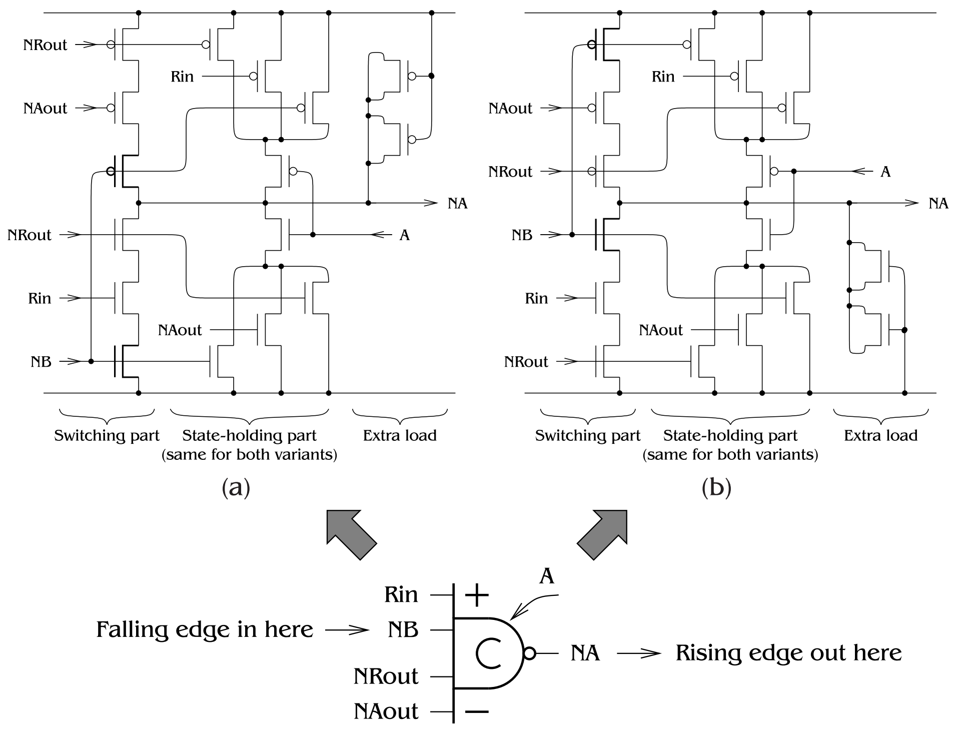
$$(l_{prev}, l_{sw}, l_{nsw}) = \left\{ \begin{array}{llll} (1,1,0) & (1,1,2) & (3,0,2) & (3,1,0) \\ (3,1,4) & (3,2,2) & (5,1,2) & (5,2,4) \end{array} \right.$$

When doing the SPICE run, it is also possible to find relatively cheaply the load that gate G presents on other gates. Three circuits are produced that time the delay of the typical gate of Figure 7.2, when loaded with (a) a single typical gate, (b) two typical gates, and (c) the gate G. The load of the gate G relative to a typical gate can then be calculated by linear interpolation. The result is typically a figure in the range 0.95–1.15; its use has been observed to halve the errors in some example circuits. The process also makes splitting the load into switching and non-switching components even less significant, because all switching loads can now be parameterized in terms of any other constant load; unfortunately, this was not spotted until after the simulation program was written.

7.2.4 Finding equivalent gates

The above method for calculating gate delays requires a SPICE run and parameter fitting for every different gate that is seen in a given circuit. These SPICE runs take typically 30–60 seconds each, so building a disk and memory cache of the results of SPICE runs is necessary for good performance. Even so, the number of different CMOS gates that can be produced by the synthesis algorithm is large, and it would be inefficient to run SPICE on a gate that is almost the same as a gate that has already had its delay parameters determined. If a way can be found to infer the parameters of one gate from another gate that has already been simulated, then this would reduce the number of SPICE runs required.

Several methods in the literature [68, 81, 166] assume that the delay of a gate will only depend on the transconductance of the conducting path, and not on the ordering of the transistors in the conducting path as long as any extra switching capacitance is compensated for. This means that the C-element producing signal NA in the example used above (Figure 7.5) can be implemented as either of the two gates in Figure 7.13, and the differences in gate delays should be minimal. In SPICE tests, the differences were actually quite large; gate (a) switched in 617 ps, gate (b) in 558 ps, a difference of 10.6%. Gate (a) also produced a slower output

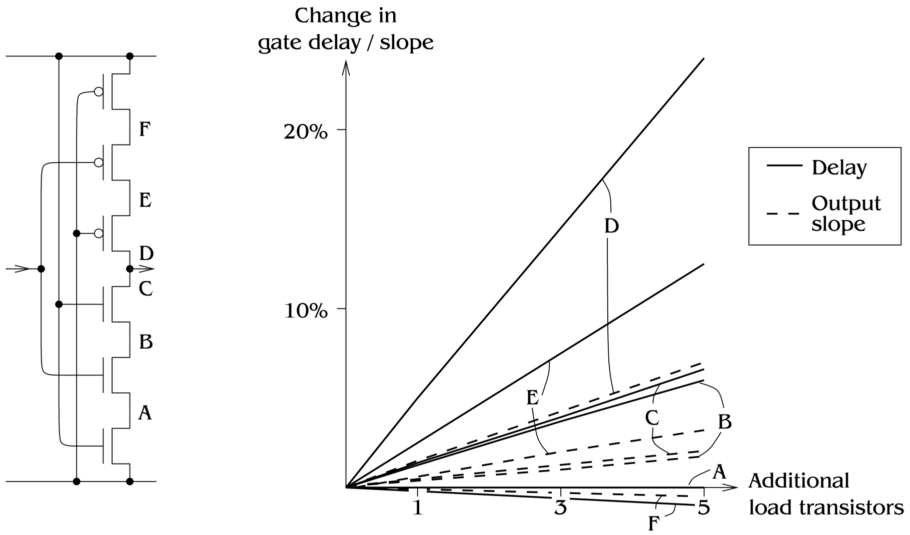


The only difference between these gates is the ordering of the transistor stacks. The transistors that cause the gate to switch are drawn bolder than the others. Both gates charge their loads through three series P-transistors, both gates have three N-transistors on a conducting path to ground, and the extra loads on the right of each gate make sure that the number of P-type and N-type source and drain regions switched by each gate is the same. However, gate (b) is 10.6% faster than gate (a).

Figure 7.13: Two gates with the same transconductance and loading, but different delays

ramp, causing the next gate to switch 4.3% slower than with gate (b). All other gate delays differed by less than 2%. This error was not cancelled out by other changes in gate delays, resulting in a total error for the chain of seven gates of 55 ps, or 2.2%. An error of 55 ps due to a single gate which only has a delay of ≈600 ps is totally unacceptable for an accurate simulation algorithm. The ordering of transistor stacks clearly does matter. This result also applies, somewhat surprisingly, to the conducting path that is being switched off as well as the one that is being switched on.

When a CMOS gate switches, some transistors will be part of a conducting path to power or ground, but others will be simply acting as capacitative loads. It would be hoped that these load transistors, which may either be on, off or switching, would have linear effects: an *n* transistor load at a particular point will have an effect roughly equivalent to *n* times the effect of a single transistor. Figure 7.14 shows the effect of 1, 3 and 5 extra transistors on a gate at various points on the



Left, circuit used to evaluate the effect of non-switching transistors. The drains of 1, 3 or 5 off N-transistors were connected to A, B and C; P-transistors were used for D, E, F. A falling transition was applied to the input of the gate, causing a rising transition of the output.

Right, the resulting increases in gate delay and output slope are linear to within an amount equal to 0.2% of the gate delay.

Figure 7.14: Effects of non-switching transistors off the conducting path

conducting paths to power and ground. A graph derived using transistors in the "on" state looks almost identical, but scaled by a factor of 1.5–2.0 depending on the location of the extra transistors, so it can be deduced that additional transistors not on the conducting path act purely as extra loads on the gate. A third possibility for extra transistors is that they are also connected to the input that is changing, so they switch during the output transition. These transistors act as being part-way between on and off transistors.

Although the extra loads appear to be linear, the magnitude of these loads is difficult to determine. Points A and F in Figure 7.14 do not charge or discharge during a rising transition of the output; additional load at those points therefore makes little difference to the gate delay and output slope. The effects of loads at the other four points must be determined from SPICE runs. For the gate in Figure 7.14, an additional twelve circuits would be added to the SPICE circuit used for parameter fitting. These would correspond to extra on, off and switching loads at each of points B, C, D and E. Each of these twelve circuits is used to determine a load on the output of the circuit that is equivalent to that kind of transistor at that point. This equivalent load is then added to l_{nsw} when calculating the gate delay.

7.2.5 Caveats

The usefulness of the synthesis tool presented ultimately lies in its ability to provide reasonably accurate timing information about generated circuits, which relies upon a stable copy of SPICE. SPICE 3f4 was used in early trials, but this fails to simulate a large number of gates, even though there is nothing wrong with the circuit as given to SPICE. Additions of 1GΩ resistors between various points in the circuits would often cure spurious errors from SPICE. A later version of SPICE, patched to 3f5, was downloaded from `http://www.redhat.com`; this version seemed much more reliable, and only failed on a handful of gates. The problem appears to be the adaptive stepsize control within SPICE: SPICE decides that no stepsize is small enough to accurately simulate the gate in question, and terminates the simulation. As a result, the proposed tool is unable to simulate a particular gate and must abandon any circuits containing that gate. The more stable version of SPICE is no longer available for download, so this may seriously affect the usability of the tool presented in this dissertation. The stability of commercial versions of SPICE has not yet been determined.

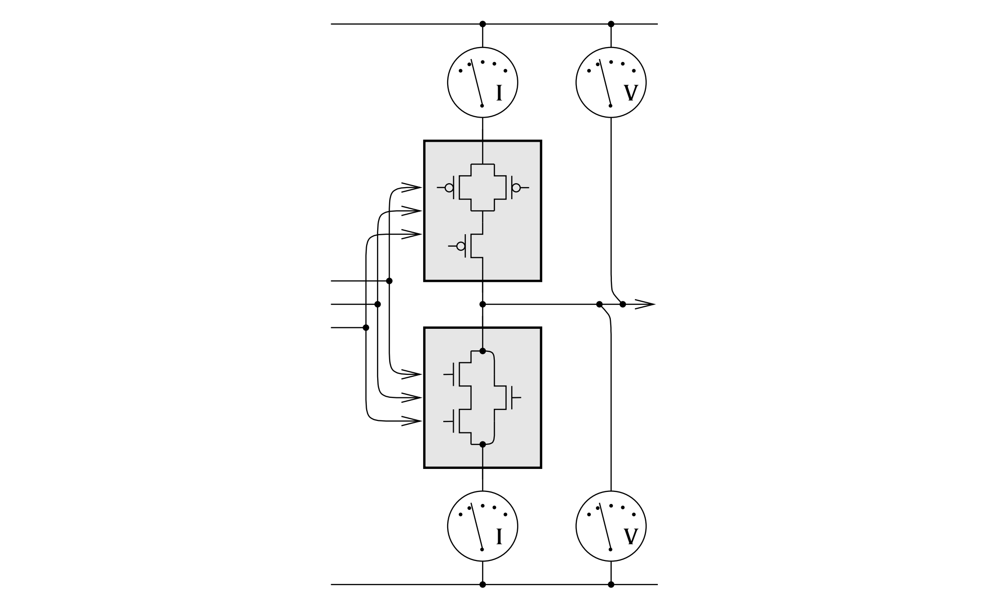
7.2.6 Power estimation

The delay and output slope of a gate are already determined by SPICE runs, so it was decided that power should also be estimated in this way. The major source of power dissipation in small CMOS circuits is I^2R heating in the transistors of a gate, although interconnect accounts for more of the power in larger circuits. To find the power dissipated by a gate, ammeters were included in every gate simulated, as in Figure 7.15, and the *VI* product across both transistor stacks was integrated and summed. This measure of power includes short-circuit current, unlike XPOWER [2] and the work of Kudva and Akella [96], but will fail in the presence of substantial subthreshold leakage current. It is assumed that power has the same dependence on input slope and output load as the gate delay and output slope. The power of a circuit is not critical, because it turns out to be almost a linear function of the active area, which is easy to compute. For critical low-power applications, it would be good to replace this power estimation algorithm by a better one.

7.3 Finding a speed measure for an implementation

It is not always obvious what defines a "fast" example of a particular circuit; for example, is a fast latch controller one that has a small `Rin+`⟶`Rout+` delay, so can propagate data forwards quickly, or is it one that has a low cycle time when connected in a pipeline? If the latter, is there a processing delay between stages? Only the circuit designer will know what particular speed measure is important, so there must be a way for the designer to tell synth how to measure speed. This is the purpose of the `.timing` file.

In order to find out how fast a particular implementation is, a test wrapper must be placed round the implementation and then the whole system simulated. This test



$$\text{Power} = \int_{\text{time of interest}} (V_p I_p + V_n I_n) dx$$

Figure 7.15: Circuit to determine the power consumed by a gate

wrapper may be as simple as applying a rising edge to a single input while keeping other inputs constant and measuring when a particular output transition occurs, or it may involve a complicated external circuit and a cycle time measurement. An example test wrapper for the latch controller example is shown in Figure 7.16. The cycle time of the latch controller is measured when it is in a pipeline with a processing delay of 20 ns and a precharge delay of 5 ns, because this was the arrangement used in Furber and Day's paper [59]. The latch driver buffers are modelled as four series inverters, as is the end of the pipeline. The start of the pipeline uses a special NOTRESET signal provided by the simulator, which makes a single rising transition at time $t = 0$. RESET is also provided, as are constant logic 0 and logic 1.

Once a speed measure has been found—here, the cycle time of the latch controller in the given circuit—the `latchc.timing` file can be created. As before, the format of the file will be made as close to Verilog as possible, but there will be significant limitations, because the full functionality of Verilog is not required. Only the following features are supported in the `.timing` file:

- `wire` declarations, which create wires in the circuit. Example: `wire a,b;` creates wires called `a` and `b`.
- Assignments of values to wires using NOT (`~`), AND (`&`) and OR (`|`) Boolean operators. The value will be computed with a complex gate, and will honour multiple NOT operators in series. For example, `assign d = ~~~((a & b) | c)` will create a complex gate for `~((a & b) | c)`, and will then add two more NOT gates to the output.

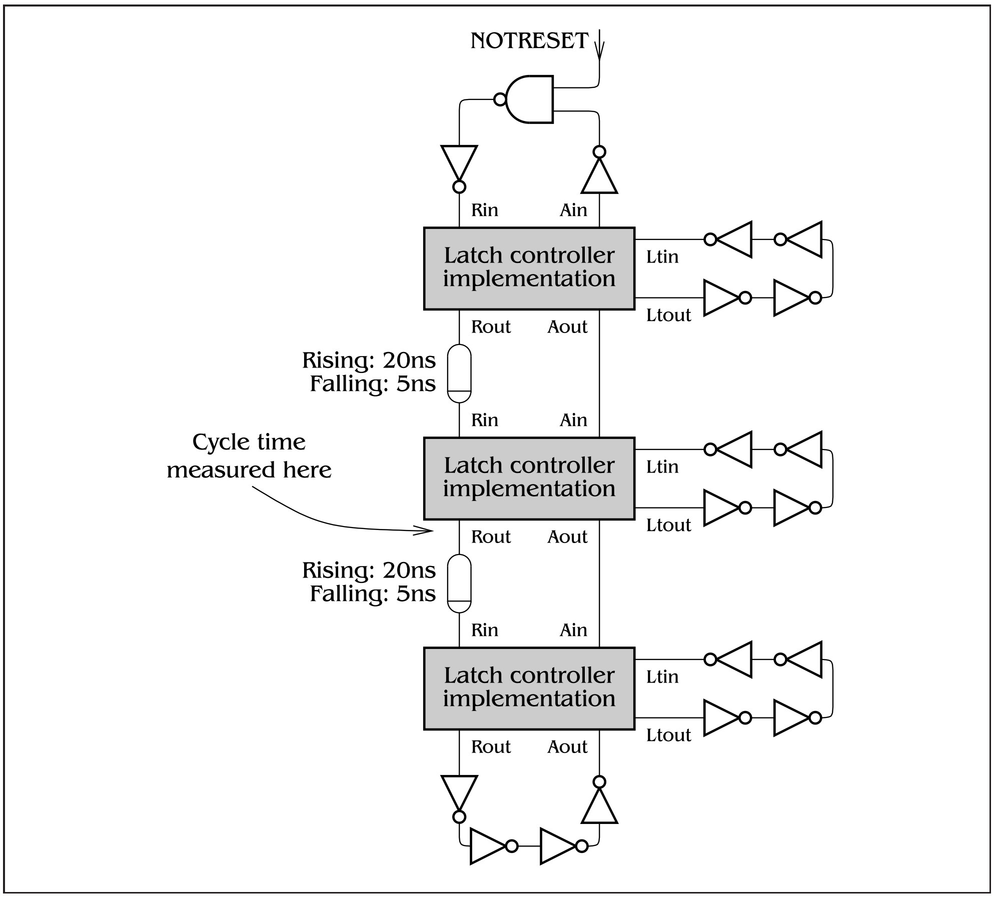


Figure 7.16: Example circuit used for timing purposes: Latch controller

In the latch controller example, a delay was needed between adjacent stages to simulate processing logic in the pipeline. A construct is needed to introduce such a delay, because physically building the delay from basic gates would be tedious. The construct #<*rise,fall*> is available, which creates a model of a delay. The rise and fall delays can be specified in gate delays, such as 2 `gd`, or in nanoseconds, as 5 `ns`. The Verilog-like construct #*delay* can be used if the rise and fall delays are the same.

- Creation of subcircuits, but the only kind of circuit that can be created is the circuit that is being timed. In the file `latchc.timing`, it is possible to write:
  `latchc` ⟨*name*⟩ `(a,b,c,d,e,f);`
  but it is not possible to create an instance of a nacking arbiter, for instance. Parameter renaming is allowed, for example using `.Rin(x)` to connect wire `x` to the `Rin` terminal of the latch controller.

```
timing
  wire   rin1, ain1, rout1, lto1, lti1,
         ain2aout1,
         rin2, rout2, lto2, lti2,
         ain3aout2,
         rin3, rout3, aout3, lto3, lti3;

  latchc one (.rin(rin1), .rout(rout1), .ain(ain1), .aout(ain2aout1),
              .ltout(lto1), .ltin(lti1)),
         two (rin2, ain2aout1, rout2, ain3aout2, lto2, lti2),
         three (rin3, ain3aout2, rout3, aout3, lto3, lti3);

  aout3= ~~~~rout3;
  rin1 = ~ ~(NOTRESET & ~ain1);
  lti1 = ~~~~lto1;
  lti2 = ~~~~lto2;
  lti3 = ~~~~lto3;
  rin2 = #<20ns,5ns> rout1;
  rin3 = #<20ns,5ns> rout2;

  cycle rout2+;
endtiming
```

Figure 7.17: The timing part of the file latchc.timing

Two additional keywords are provided to specify how the circuit is to be timed:

- cycle ⟨*transition name*⟩ measures the cycle time between adjacent named transitions.
- time ⟨*trans1*⟩,⟨*trans2*⟩ measures the time elapsed between a pair of named transitions.

The file latchc.timing is shown in Figure 7.17, to illustrate the use of most of these features. This produces the circuit in Figure 7.16. The timing wrappers used for the parallel component, loadable counter, DME circuit and nacking arbiter are shown in Figure 7.18–7.21. The timing wrapper for the nacking arbiter produces alternate requests on lr and rr, because of the properties of a Seitz arbiter. These alternate requests will produce rising transitions on ly, rn, ln, ry, ln, rn and then the sequence repeats, so the cycle time can be determined by looking at adjacent ly+ transitions.

7.3.1 Action when timing wrapper is not known

Example timing wrappers are not given for the SIS benchmarks examples, so some other fall-back timing method needs to be provided for these cases. The blue diagram for the environment was used as an approximation to the actual environment behaviour. The gate-level simulator was written to accept blue diagram models of a circuit behaviour, with given delays on outputs, as well as normal CMOS gates. An

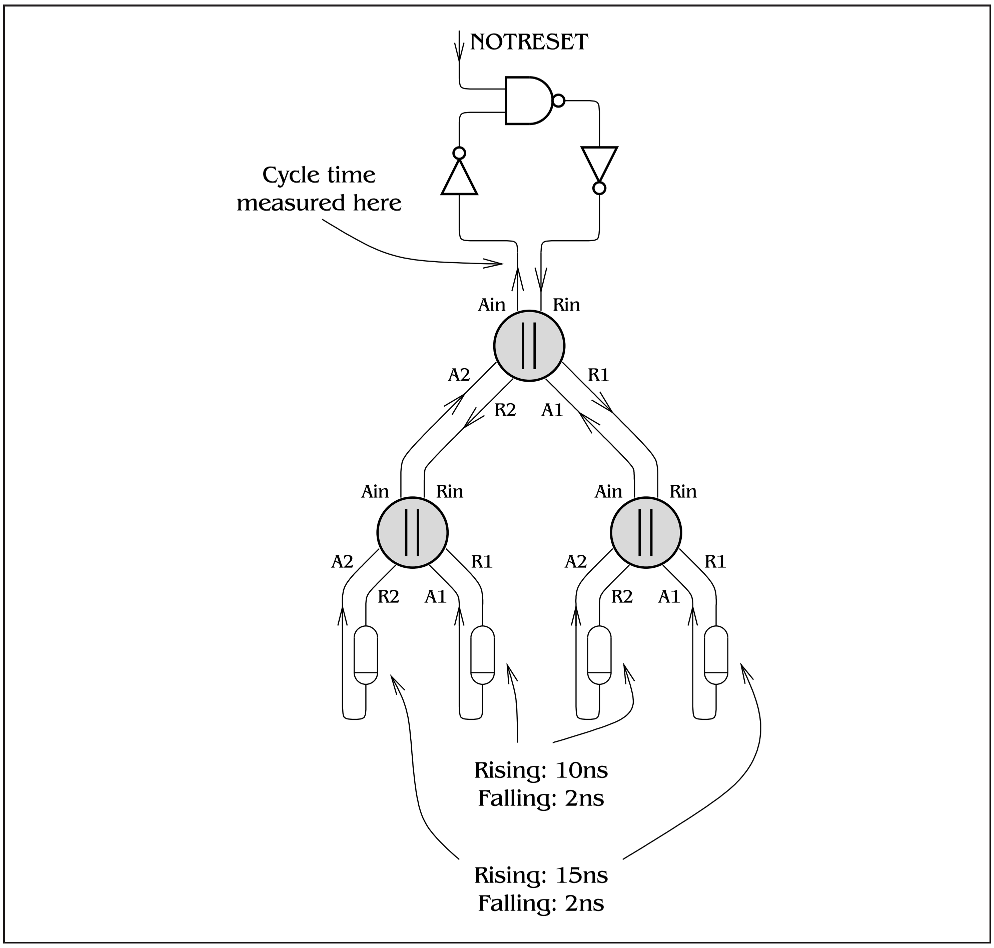


Figure 7.18: Example circuit used for timing purposes: Parallel component

approximation to the behaviour of the latch controller example could be specified in the file `latchc.timing` as:

```
timing estimate
  default delay 4gd;
  rin delay 20ns,5ns;
endtiming
```

This causes the blue diagram model of the environment to have a delay of 4 gate delays on all outputs, apart from `Rin`, which has a rising delay of 20 ns and a falling delay of 5 ns, which emulate a processing delay between this stage and the previous stage. If no `.timing` file is provided at all, a default delay of 2 gate delays is assumed on all inputs to the circuit.

The effect of approximating the environment behaviour is difficult to quantify. In the latch controller example, the fastest circuit using a blue diagram model of the environment will be a long way from the fastest circuit using the cycle time of a pipeline as a measure of speed, because in the latter case, the speed of the circuit

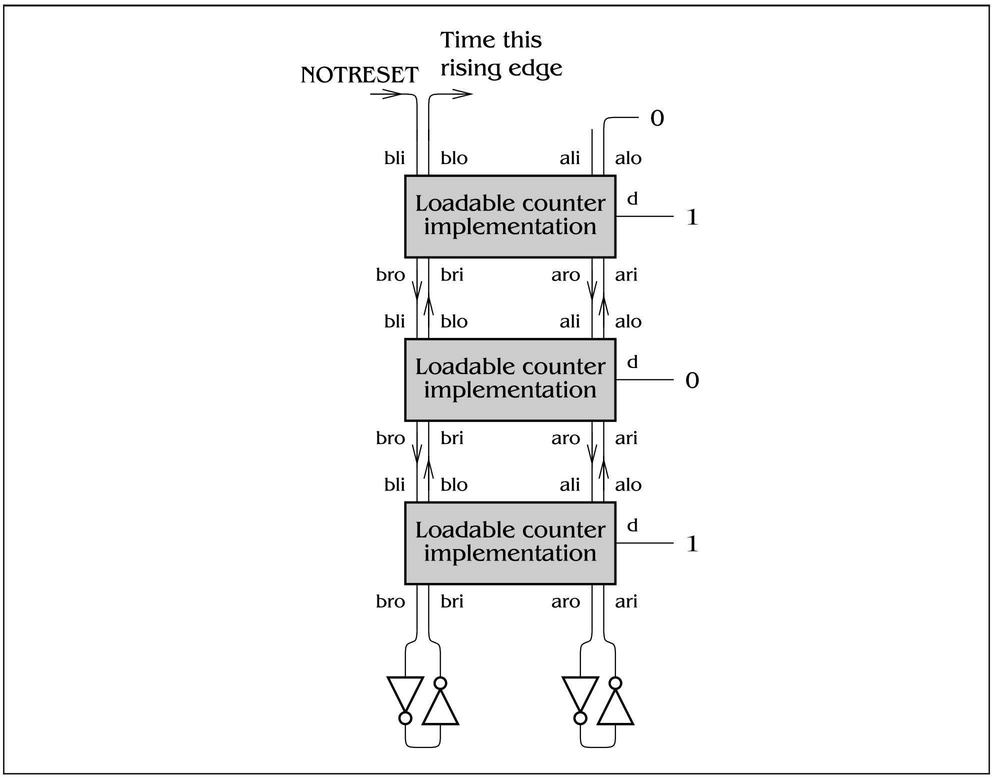


Figure 7.19: Example circuit used for timing purposes: Loadable counter

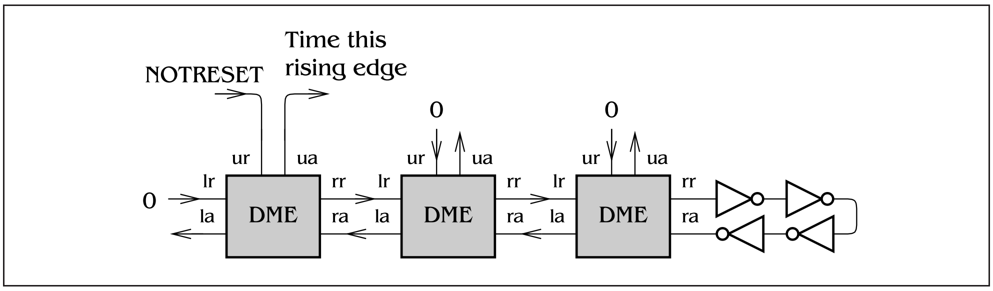
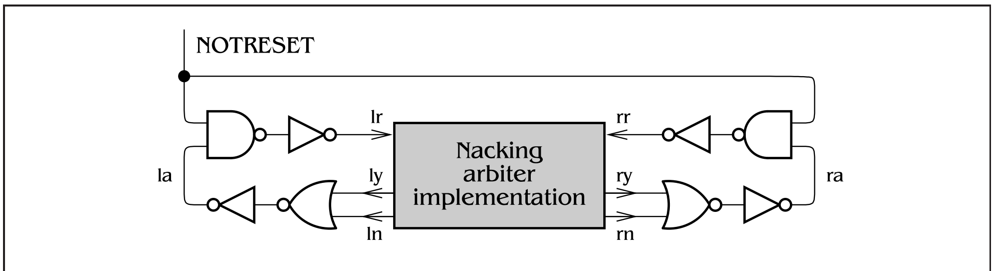


Figure 7.20: Example circuit used for timing purposes: DME



la = ly + ln ra = ry + rn
lr = la.NOTRESET rr = ra.NOTRESET

Figure 7.21: Example circuit used for timing purposes: Nacking arbiter

depends heavily on the decoupling between pipeline stages, but in the former, this is not the case. This will also be true of the DME, parallel and loadable counter examples. The nacking arbiter circuit, and many of the SIS examples, are not intended to be connected to other copies of themselves, so the blue diagram model provides an adequate model of the environment behaviour.

7.4 Verification

7.4.1 Reasons for verification

Post-synthesis verification is often used in asynchronous circuit synthesis. Verification of speed-independent circuits is as much about checking the tools as the circuit itself, because SI synthesis algorithms should produce circuits that are correct by construction; in this case, verification serves only to highlight program bugs. Verification of non-SI circuits, such as timed circuits [7, 137] or burst-mode circuits [34, 33, 45, 140, 203] is more about checking whether a circuit malfunctions due to imperfect synthesis algorithms, and then patching the circuit to remove the problem. The circuits produced in the previous chapter fall into this last category; they need to be checked because they may have hazards, deadlock, livelock or a host of other undesirable behaviours.

7.4.2 Types of verification

Verification of speed-independent circuits is comparitively easy in principle, although in practice may be very time-consuming. The basic idea is to allow any excited gate to fire at any time relative to other gates, and keep track of all states that the circuit passes through. The most well-known SI verifier is Dill's AVER [53].

SI verification will not work for the circuits produced in this dissertation, because the circuits produced often assume that the environment cannot respond "too fast" to an output; ie. it might not matter if the environment takes 2 ns, 20 ns or 200 ns to provide a particular input after the output that triggers it, but 1.8 ns may break the circuit. A simulator is needed that can use a range of delay values for each gate rather than a fixed nominal value. All possible states of the implementation can be found while assuming that all gates delays can vary by 20% or 50% or more, and then the behaviour of the circuit checked against the specification.

A pair of simulation algorithms that use ranges for the delays of gates are presented in Brzozowski and Seger [21]. The subject is known as *bi-bounded delay analysis*, to distguish it from *up-bounded* analysis where all delays are between zero and some constant, and the unbounded analysis that is used for speed-independent circuits. The more computationally efficient bi-bounded algorithm uses a ternary model of signals on wires: signals can takes values from the set $\{0, 1, \Phi\}$, where Φ denotes an uncertain value, which can be either 0 or 1 depending on the precise delays in the circuit. The value Φ is not the same as X, which usually denotes a value which is part-way between a 0 and a 1. The rule that gives the outputs of a gate with minimum delay d_{min} and maximum d_{max} is:

- When any input changes, the output will change to Φ in a time d_{min} unless it is already at or changing to Φ.
- At a time d_{max} after all inputs become defined at 0 or 1, the output will change to either 0 or 1 as appropriate for those inputs.

The problem with ternary bi-bounded simulation is that it is inherently pessimistic. It returns the most information that can be said about a circuit at a particular time t; if a node x could be either 0 or 1 depending on the precise gate delays at time t, then the ternary simulation algorithm will assign node x the value Φ, because the most that can be known about the node is nothing. This is useless when a closed cycling circuit is being considered. For example, consider verifying a circuit with a cycle time of t_{cyc}, but allowing gate delays to vary by a factor k. At time $t = 0$, the circuit is in a particular state. It will be in this state again sometime between $t = (1 - k)t_{cyc}$ and $t = (1 + k)t_{cyc}$, because of the variations in gate delays. The nth time it is in this state will occur for t in the interval $[(1 - k)nt_{cyc}, (1 + k)nt_{cyc}]$. For some n, it will be the case that $(1 + k)nt_{cyc} < (1 - k)(n + 1)t_{cyc}$, which means that the circuit could be at any point in its cycle for a certain time $t_1 \in ((1 + k)nt_{cyc}(1 - k)(n + 1)t_{cyc})$, and hence the most that can be known about the circuit is nothing. This will always happen with circuits that cycle; after a time, ternary simulation will assign the value Φ to all nodes in the network. In practice, this occurs very fast, especially in circuits involving arbiters.

7.4.3 Binary Bi-bounded Delay Analysis

The other algorithm given for bi-bounded delay analysis in Brzozowski and Seger [21] uses binary rather than ternary. It is much more computationally expensive than ternary analysis, so was said to only be useful for small circuits. Empirically, circuits in this dissertation appear to fit their definition of small. This section will describe the binary bi-bounded delay model, or BBD model for short. It will be described in sufficient detail to allow the algorithm to be implemented, in a way which is less formal than the treatment given in [21]; the emphasis will be on how to create a program to do the simulation, rather than presenting it in a way which allows a formal proof that it works. Full details of why it works can be found in Dill [52] and Lewis [107], but they are beyond the scope of this dissertation. Some additions to the algorithm are given in the next section, which make it more suitable for the verification task required.

First, some properties of intervals must be defined. Let τ be an integer interval, ie. a 4-tuple $(t_{lower}, t_{upper}, C_{lower}, C_{upper})$, with t_{lower} being an integer or $-\infty$, t_{upper} being an integer or $+\infty$, and C_{lower} and C_{upper} being Boolean flags that determine whether the ends are closed (true) or open (false). According to the usual convention, closed ends are written as "[" or "]" and open ends as "(" or ")", so an example interval would be [1,2). Intersection, addition and negation of intervals are defined in the obvious way, as:

If $I_1 = (t_{l1}, t_{u1}, C_{l1}, C_{u1})$ and $I_2 = (t_{l2}, t_{u2}, C_{l2}, C_{u2})$,

$$\text{then } I_1 \cap I_2 = (t_l, t_u, C_l, C_u) \text{ where } (t_l, C_l) = \begin{cases} (t_{l2}, C_{l2}) & : t_{l1} < t_{l2} \\ (t_{l1}, C_{l1}) & : t_{l1} > t_{l2} \\ (t_{l1}, C_{l1}.C_{l2}) & : t_{l1} = t_{l2} \end{cases}$$

$$\text{and } (t_u, C_u) = \begin{cases} (t_{u1}, C_{u1}) & : t_{u1} < t_{u2} \\ (t_{u2}, C_{u2}) & : t_{u1} > t_{u2} \\ (t_{u1}, C_{u1}.C_{u2}) & : t_{u1} = t_{u2} \end{cases}$$

$$\text{and } I_1 + I_2 = (t_{l1} + t_{l2}, t_{u1} + t_{u2}, C_{l1}.C_{l2}, C_{u1}.C_{u2})$$

$$\text{and } -(I_1) = (-t_{u1}, -t_{l1}, C_{u1}, C_{l1})$$

Subtraction of an interval τ is defined as addition of $-\tau$. All gates in the circuit are assumed to have their possible delays specified by an interval, where gate j may fire in the interval $[d_j, D_j)$ after its inputs cause it to become excited. Note that the above operations can all be performed efficiently, even though they represent infinite sets of points.

A plausible approach to bi-bounded simulation would be to simply keep bounds on the times that excited gates could fire, but this can be shown to be insufficient. Consider the circuit shown in Figure 7.22, which has three excited gates at t=0, with firing times bounded by $[1, 2)$, $[3, 4)$ and $[3, 4)$ for gates 1, 2 and 3 respectively. Gate 1 must fire first, at some time $t \in [1, 2)$; we can move the circuit forwards to the time t_{g1} at which gate 1 fires by subtracting $[1, 2)$ off of the other two intervals, which gives that gates 2 and 3 can both fire in the interval $t - t_{g1} \in (1, 3)$. This is true, but not the whole story. Either of gates 2 and 3 can fire at time 1.1 or 2.9 after gate 1 fires, but it is not possible for gate 2 to fire at $t = t_{g1} + 1.1$ *and* gate 3 to fire at $t = t_{g1} + 2.9$, because this would imply a time separation of 1.8 time units between gates 2 and 3, which was not possible in the original firing intervals. The bounds between firing intervals on different gates have been lost, so bounds should also be kept on the differences between the firing times of all pairs of excited signals. Brzozowski and Seger [21] state that this is sufficient to carry out a bi-bounded delay analysis.

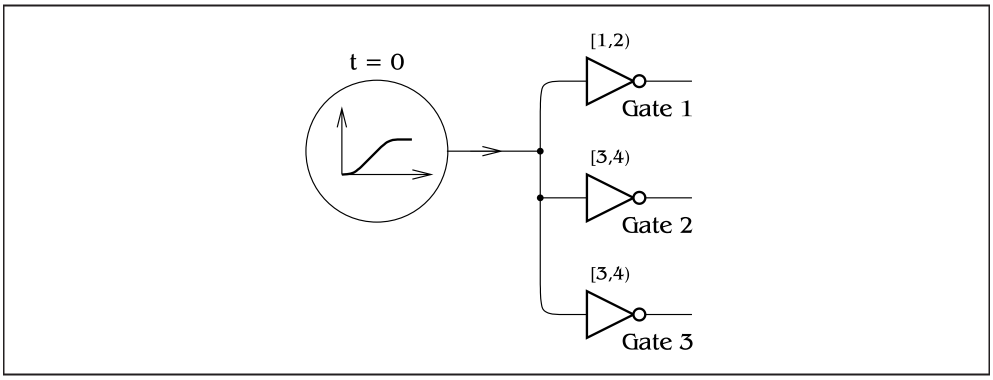


Figure 7.22: Example circuit used to illustrate the BBD algorithm

The BBD algorithm must keep track of:

- F, the bounds on the firing times of all gates,
- G, the bounds between the firing times of all gates,
- c, the current state of all wires in the circuit.

Given a triple (F, G, c), the BBD algorithm advances time until one or more gates may fire, and then fires those gates, producing a new triple (F', G', c'). Several different successor triples may be produced from a single triple, because gates may fire in different orders. The algorithm in Brzozowski and Seger is used on circuits which will eventually stabilize, but it also works in circuits that show cyclic behaviour if some minor modifications are made. Firing a number of gates at one time rather than just one gate gives oscillation when considering cross-coupled NAND gates, which models the metastability phenomenon. In the circuits in this dissertation, all arbitration behaviour is concealed by analogue circuits in a Seitz arbiter, so metastability will not occur. The algorithm described here will only fire a single gate at a time, so it is a slight simplification of the algorithm in Brzozowski and Seger.

When implementing the algorithm, it is convenient to combine F and G into a single matrix Σ, by creating a fictitious gate number 0 representing the current time. Σ has entries σ_{ij}, each of which are intervals, defined by $r_i - r_j \in \sigma_{ij}$ for $0 \leq j < i \leq N$, where N is the number of gates, r_i and r_j are the firing times of gates i and j respectively, and $r_0 = 0$, the current time. Any entry σ_{ij} where one or both of gates i and j are excited is defined as $(-\infty, \infty)$.

Given a matrix Σ, it is a problem to determine whether there is a set of gate delays $(r_1, r_2, \ldots r_N)$ that satisfy the inequalities represented by $r_i - r_j \in \sigma_{ij}$. If a set of delays exists, then Σ is a *feasible* matrix, if not, an *infeasible* one. To determine whether Σ is feasible, the algorithm in Figure 7.23 was proposed, which is a modification of the Floyd-Warshall all pairs shortest path algorithm. If $\theta(\Sigma, c)$ has any empty intervals, then Σ is infeasible, otherwise $\theta(\Sigma, c)$ is a canonicalized version of Σ that can be used to compare two matrices with each other.

```
Function $\theta(\Sigma, c)$
    Matrix $\Sigma' = \Sigma$
    For $k$ = 0 to N, such that gate $k$ is excited or $k$=0
        For $i$ = 1 to N such that $i \neq k$ and gate $i$ is excited
            For $j$ = 0 to $(i - 1)$ such that $j \neq k$ and gate $j$ is excited
                $\sigma'_{ij} := \sigma'_{ij} \cup (\sigma'_{ik} + \sigma'_{kj})$, where $\sigma'_{pq}$ is defined as $-\sigma'_{qp}$ if $p < q$
    Return $\Sigma'$
EndFunction
```

Figure 7.23: A modified Floyd-Warshall algorithm to determine feasibility

The starting state for the algorithm is the pair $(\theta(\Sigma, c), c)$ with c being the initial state of wires in the circuit, and $\sigma_{i0} = [d_i, D_i)$ for all excited gates i and all other $\sigma_{ij} = (-\infty, \infty)$.

Given a state (Σ, c), a successor state can be found by:

1. If there are no excited gates, then there are no successors.
   Otherwise, find the minimum M of the upper bounds on the firing times of all excited gates. At least one gate must fire before this time. Pick any gate x which is excited and can fire before M, ie. $\sigma_{x0} \cap [0, M] \neq \emptyset$.

2. Create a copy $\dot{\Sigma}$ of Σ.

3. All gates fire from this point on in time, so for all i, $\dot{\sigma}_{i0} := \dot{\sigma}_{i0} \cap (0, \infty)$

4. Gate x fires first, so all other excited gates must fire later:
   For all $i > x$, $\dot{\sigma}_{ix} := \dot{\sigma}_{ix} \cap (0, \infty)$
   For all $i < x$, $\dot{\sigma}_{xi} := \dot{\sigma}_{xi} \cap (-\infty, 0)$

5. Check whether $(\dot{\Sigma}, \dot{c})$ is feasible, by finding $\ddot{\Sigma} = \theta(\dot{\Sigma}, c)$. If $\ddot{\Sigma}$ has any empty intervals, then x could not be fired. Abandon and return to step 1, picking another x if there is one.

6. Create a copy $\tilde{\Sigma}$ of $\ddot{\Sigma}$ and a copy $\tilde{c}$ of c.

7. Change the output of gate x in $\tilde{c}$ to its new state.

8. For all gates j that were (a) excited before x fired, (b) remain excited after x has fired, and (c) are not x, do $\tilde{\sigma}_{j0} := \ddot{\sigma}_{jx} \cap (0, \infty)$, with the convention that σ_{pq} is defined as $-\sigma_{qp}$ if $p < q$, as before. This step makes sure that gate j will fire at the right time relative to gate x.

9. For all gates j that were not excited before x fired, but are afterwards, including gate x if that is again excited, do $\tilde{\sigma}_{j0} := [d_j, D_j)$. This sets up excited gates to fire at the right time.

10. For all gates j that are not excited after x fires, set $\tilde{\sigma}_{j0} := (-\infty, \infty)$.

11. Set $\tilde{\sigma}_{ij}$ to be $\ddot{\sigma}_{ij}$ if i and j are both excited gates, and neither of them is x. Otherwise, set $\tilde{\sigma}_{ij}$ to $(-\infty, \infty)$.

12. Record $(\tilde{\Sigma}, \tilde{c})$ as a successor state to (Σ, c).

The algorithm looks formidable, but is actually quite simple. A quick example of its use will now be presented, using the example circuit shown back in Figure 7.22. The initial state of this circuit has the output of gates 1, 2 and 3 at logic 1, so the starting value of c is (1,1,1). The starting value of Σ is:

$$\Sigma = \begin{pmatrix} \omega & \omega & \omega & \omega \\ [1,2) & \omega & \omega & \omega \\ [3,4) & \omega & \omega & \omega \\ [3,4) & \omega & \omega & \omega \end{pmatrix}$$

where ω denotes $(-\infty, \infty)$. Running the Floyd-Warshall algorithm on Σ to find $\theta(\Sigma, c)$ gives the starting state for the algorithm:

$$(\Sigma_0, c_0) = (\theta(\Sigma, c), c) = (\begin{pmatrix} \omega & \omega & \omega & \omega \\ [1,2) & \omega & \omega & \omega \\ [3,4) & (1,3) & \omega & \omega \\ [3,4) & (1,3) & (-1,1) & \omega \end{pmatrix}, (1,1,1))$$

In this matrix, it can be seen that gates 2 and 3 must fire in some interval (1,3) after gate 1 fires, and the time difference between when gates 2 and 3 fire must lie in the interval (-1,1).

Step 1 of the algorithm finds that $M = 2$, and the only gate that can fire before time $t = 2$ is gate 1. Gate 1 must fire next.

Steps 2, 3 and 4 create the matrix $\dot{\Sigma}_0$, which in this case is the same as Σ_0 above. $\ddot{\Sigma}_0$ is therefore also the same as Σ_0.

Steps 6 onwards create $\tilde{\Sigma}_0$. In step 8, gates 2 and 3 were excited before gate 1 fired, are still excited after, and are not gate 1, so $(\tilde{\sigma}_0)_{20} = (\ddot{\sigma}_0)_{21} \cap (0, \infty) = (1, 3)$ and the same for $(\tilde{\sigma}_0)_{20}$. Steps 10 and 11 set various entries to ω, giving a successor state (Σ_1, c_1):

$$(\Sigma_1, c_1) = (\tilde{\Sigma}_0, \tilde{c}_0) = (\begin{pmatrix} \omega & \omega & \omega & \omega \\ \omega & \omega & \omega & \omega \\ (1,3) & \omega & \omega & \omega \\ (1,3) & \omega & (-1,1) & \omega \end{pmatrix}, (0,1,1))$$

Starting again with (Σ_1, c_1), step 1 gives that $M = 3$ and either gate 2 or gate 3 can fire. Assume gate 2 fires. Steps 2, 3 and 4 again create $\dot{\Sigma}_1 = \ddot{\Sigma}_1 = \Sigma_1$.

In step 8, the only gate excited both before and after gate 2 is gate 3, so $(\tilde{\sigma}_0)_{30} = (\ddot{\sigma}_0)_{32} \cap (0, \infty) = (0, 1)$, giving the successor state

$$(\Sigma_2, c_2) = (\tilde{\Sigma}_1, \tilde{c}_1) = (\begin{pmatrix} \omega & \omega & \omega & \omega \\ \omega & \omega & \omega & \omega \\ \omega & \omega & \omega & \omega \\ (0,1) & \omega & \omega & \omega \end{pmatrix}, (0,0,1))$$

It can be seen that the algorithm correctly determines that gate 3 can only fire upto one time unit after gate 2.

7.4.4 Additions to the algorithm

Two main alterations were made to the above algorithm. The first was to replace the constant delay (d_j, D_j) of gate j by a delay derived from the accurate timing models described earlier in this chapter. The output slope of all excited gates is stored with the current state c. When a gate i switches, the slope of its output waveform is then used to calculate accurate delays and output slopes for all the gates that become excited as a result of gate i switching. The range of delays used for a gate j is $[\lfloor \frac{d}{1+\alpha} \rfloor, \lceil (1+\alpha)d \rceil)$ where d is the calculated gate delay, expressed in hundredths of a constant typical gate delay, and α is a parameter that determines the permitted variations in gate delays. The default value of α is 20% or 0.2, but it can be changed as described below.

The second alteration was to allow circuits to cycle indefinitely, rather than only allowing circuits with a definite set of final states, as in the Brzozowski and Seger algorithm. This requires the ability to determine whether a state is very like one that has been seen before, even though it may not be identical. An example that was seen in the early stages of writing the verifier was a case when rising edges on a pair of wires a and b were about to happen, but three separate states were created:

State 1:	a will happen in [20,99),	b will happen in [30,99)
State 2:	a will happen in (21,99),	b will happen in [30,99)
State 3:	a will happen in [20,99),	b will happen in (31,99)

These states are obviously almost identical, so should really be merged and treated as the same state. Similar, but more subtle problems occur when a circuit enters a state it has been in before, but because the gate delays depend on waveform slopes and the slopes depend on previous gate delays, the gate delays in the new state are slightly shifted.

The solution used was to say that two states of the circuit (Σ, c) and (Σ', c') are *sufficiently close* if $c = c'$ and $\sigma_{ij} \cup \sigma'_{ij} \neq \emptyset$ for all i,j, which means that the same gates are excited in both and there is a pattern of gate delays that can occur in either state. The weaker condition of just requiring $c = c'$ was found to cause problems with circuits in which there is a race condition, but the race is always won by one side or the other.

During simulation, a list L of states (Σ, c) is built up, starting with the initial state. Every time a new state (Σ', c') is seen, it is checked against all known states. If it has not been seen before, (Σ', c') is added to L. If (Σ', c') is sufficiently close to another state (Σ'', c'') that has been seen before, a new state $(\hat{\Sigma}, \hat{c})$ is formed by taking the union of all intervals in Σ' and Σ''. The new state $(\hat{\Sigma}, \hat{c})$ encompasses the possible behaviours of both (Σ', c') and (Σ'', c''). If $\hat{\Sigma} = \Sigma''$, then the new state was contained in the old state (Σ'', c''); if not, the old state (Σ'', c'') is replaced with $(\hat{\Sigma}, \hat{c})$. The simulation continues until all successors of states in L are also in L. This process is guaranteed to converge by results in [21], and in practice, it does so within a second for most of the examples in this dissertation.

Consider the case when, in addition to the three states listed above, a fourth state occurred that had a occurring in [120,199) time units and b in [130,199), and no other gates were excited. This is clearly 100 time units before the three states above, so we can see that the set of behaviours reachable from this state is the same as the set reachable from the three above, but the intersection of the firing times of a and b is empty. To correct this problem, at Step 12 of the algorithm, states are always stepped on in time so that one of the signals fires in an interval (0,t) or [0,t). This makes the states more canonical, and reduces the number of states that the algorithm must find.

Minor additions were made to the algorithm to make sure that the circuit does not malfunction. The implementation to be checked needs some model of the full environment behaviour, so a single example of the circuit to be verified is composed with the blue diagram model of the environment. The BBD simulator was written to accept blue diagrams as well as gates, with delay ranges specified by the user or set at the default of two to twenty gate delays. During the simulation, the implementation is checked against the blue diagram for the circuit to see whether the implementation conforms to the specification.

Hazard checks are also implemented: any gates on the output of the circuit that are enabled and then disabled without firing might produce hazards, so this causes the circuit to fail the verification. Hazards inside the circuit do not cause the circuit to fail, but if these hazards get to an output or might possibly cause erroneous behaviour, then the circuit will be failed. The circuit will also be failed if there is a direct DC path from the supply to ground through a gate, or if a gate floats at a voltage which is not supply or ground.

The verification parameters can be changed by including a `verify`...`endverify` section in the `.timing` file for a circuit. The amount by which gate delays vary can be altered, and so can the environment delay on each input to the circuit. An example file for the latch controller is:

```
verify 20%
  rin delay 3gd - 30ns;
  aout delay 2gd - 30ns;
  ltin delay 3gd - 20gd;
endverify
```

7.4.5 Summary

Verification is necessary for the circuits produced in this dissertation, because the synthesis procedure is not provably correct. The binary bi-bounded analysis from Brzozowski and Seger [21] is particularly suitable for verifying the circuits produced. The algorithm is modified to use the accurate timing models presented earlier, and to work on circuits that do not have a final state but repeat a set of behaviours forever. Conformance to specification, hazard checks and other techniques are performed to make sure that the proposed implementation for a circuit does not malfunction. The verification algorithm is the most time-consuming part of the synthesis procedure, but perhaps the most important.

Results 8

Structure of this chapter

The results in this chapter are presented in four parts. The use of static and dynamic gates with keeper inverters was discussed in Section 6.4.2, and the pseudo-static gate introduced as an attempt to make faster and smaller circuits. Section 8.1 compares these different types of gate when they are used to synthesize the latch controller and DME examples. Section 8.2 compares the state assignment methods that were discussed in Section 6.3. Section 8.3 gives the results of running the tool described in this dissertation on the five example circuits that were described in Section 4.2, and compares these results with those obtained by other asynchronous circuit synthesis tools. Finally, Section 8.4 gives some limited results of running the tool on the other examples that were described in Chapter 4, taken from the SIS benchmark STGs.

All cycle time, response time and power measurements in this chapter were calculated using SPICE 3f4 patched to 3f5, with an example 1 micron technology file that came with this distribution of SPICE. Power measurements were obtained using a trapezium rule integration of the current taken from the power supply, and so reflect the total power taken by the circuit under test and the rest of the test wrapper. In particular, this means that most of the power taken by the latch controller example (Figure 8.1) is actually dissipated inside the delay on the forward request path, which does not depend on the controller implementation. The size of implementations is taken to be proportional to the total active area of gates in the implementation. P transistors have been assumed to be twice as wide as N transistors, although this ratio can be changed.

8.1 Comparison of static, pseudo-static and dynamic gates

Section 6.4.2 described four kinds of gate that can be used to implement an asynchronous circuit:

- **s**: Fully static gates,
- **s2**: Not quite static gates. Gates such that either the P tree or N tree will always conduct in all reachable states of the circuit, but the P tree and N tree are not necessarily duals of each other.

Type of gate used	Cycle time	Energy per cycle	Size (active area)	Number that passed verification
s	Reference case			711/847
s2	0.0 %	+0.2 %	+0.6 %	703/847
ps	−0.1 %	+0.4 %	+0.5 %	692/847
d	−0.4 %	−0.4 %	+25.3 %	813/847

Table 8.1: Comparing the four types of gate, for the latch controller example

- **ps**: Pseudo-static gates, where the output of the gate is defined only in states of the circuit that may persist indefinitely or states that require a change in the output value of the gate. This type of gate may rely on charge retention for periods that are a small number of gate delays.

- **d**: Dynamic gates with keeper inverters. The P and N trees are only used to change the current state of the gate; otherwise, a weak inverter will keep the current state indefinitely.

By removing some restrictions from static gates, it was hoped that pseudo-static gates would have fewer transistors on average than static gates. Pseudo-static implementations should not be larger than static implementations, because a static gate also satisfies the criteria for a pseudo-static gate; hence a fully static implementation is a lower bound in some sense for a pseudo-static implementation. Fewer transistors in a gate means faster operation, lower power and smaller chip area, so pseudo-static gates should show a clear advantage over static gates.

Table 8.1 gives a comparison between static, pseudo-static, s2 and dynamic gates, when used to synthesize the latch controller with the modified Tracey state assignment method. For each of the 847 pruned blue diagrams produced from the latch controller specification, a circuit was derived using each of the four types of gate. The circuits were verified, and the estimated cycle time, energy consumed per cycle and size were calculated for each circuit. Table 8.1 lists the average improvement in cycle time, energy and size when using each of the four kinds of gate, relative to using static gates. For these columns, a lower figure is better. Not all circuits produced passed the verification stage; the number of circuits that passed using each type of gate is also shown.

It can be seen that pseudo-static gates produce implementations that are a little faster than static implementations, but are larger and take more power. This is counter-intuitive; the average number of transistors per gate should decrease when using pseudo-static gates, which is precisely what is observed. The extra size and power must be coming from something other than the gates themselves.

This paradox is best explained by an example. When creating a static gate, any input that is connected to an N transistor is also connected to at least one P transistor, and vice versa. Consider a static gate with four transistors in the N tree. This will also have four transistors in the P tree, and there will be four inputs to this gate.

Type of gate used	Cycle time	Energy per cycle	Size (active area)	Number that passed verification
s	0.0 %	−0.1 %	−0.4 %	753/847
s2	0.0 %	+0.3 %	+0.6 %	746/847
ps	−0.1 %	+0.2 %	+0.3 %	733/847
d	−0.4 %	−0.3 %	+25.3 %	814/847

Table 8.2: As Table 8.1, but with a modified Quine-McCluskey cost function

Type of gate used	Cycle time	Energy per cycle	Size (active area)	Number that passed verification
s	Reference case			800/847
s2	+0.3 %	+0.5 %	+0.8 %	782/847
ps	+0.3 %	+0.6 %	+0.5 %	688/847
d	+1.2 %	+5.3 %	+36.0 %	813/847

Table 8.3: Effects of type of gate used for latch controller, MPP state assignment

When a pseudo-static gate is constructed, it is not the case that the P and N transistors must come in pairs. By removing the restriction that either one tree or the other must conduct in every state, it might be possible to reduce both the P tree and N tree to three transistors, but these transistors no longer come in pairs. Assume there is only one input shared between the P and N trees; so this gate requires five inputs. The number of inputs is more than in the static case, so the number of inverters that will be required to create negated values of input, state or output signals will be more than in the static case, on average. Saving a transistor or two per gate may require more inverters, and the extra circuitry and power consumed by these inverters outweighs any benefits in real circuits.

An attempt was made to reduce the number of inverters required in implementations by altering the Quine-McCluskey algorithm that was used. The cost of a negated literal was made to be twice the cost of an non-negated literal in the expression for the N-tree of a gate, and vice versa for the P tree. This produced the results shown in Table 8.2, where the figures given are again relative to the first row of Table 8.1. This modification to the Quine-McCluskey algorithm produces circuits that are insignificantly different in speed, power and size than the unmodified algorithm, but more importantly, several more of these circuits passed verification compared to the unmodified Quine-McCluskey algorithm. This unexpected but welcome modification was used while deriving the rest of the results in this chapter.

The four different kinds of gate were also compared for the MPP state assignment when applied to the latch controller, and both the MM and MPP methods for the DME example, giving the results in Tables 8.3, 8.4 and 8.5. The results for the parallel component were similar to the latch controller, so they are not listed. The loadable counter and nacking arbiter do not have enough different implementations produced from concurrency reduction to give reliable figures.

Type of gate used	Cycle time	Energy per cycle	Size (active area)	Number that passed verification
s	Reference case			444/848
s2	+0.3 %	+0.3 %	+0.7 %	467/848
ps	+0.3 %	+0.2 %	+0.2 %	455/848
d	−3.9 %	−27.2 %	+14.4 %	840/848

Table 8.4: Effects of type of gate used for DME example, MM state assignment

Type of gate used	Cycle time	Energy per cycle	Size (active area)	Number that passed verification
s	Reference case			569/848
s2	+2.3 %	+2.9 %	+0.8 %	312/848
ps	−0.2 %	+1.4 %	−6.5 %	243/848
d	−2.4 %	−3.6 %	+8.5 %	387/848

Table 8.5: Effects of type of gate used for DME example, MPP state assignment

The results in Tables 8.2–8.5 are largely inconclusive, because the estimated cycle time and energy per cycle are about 4–5% out from the actual values obtained using SPICE, but the differences between the different types of gate are typically much smaller than 4%. These results show that pseudo-static gates do not show any significant improvements over conventional static gates on average. In Tables 8.3 and 8.4, the pseudo-static implementations were larger, slower and consumed more power than the static implementations. In Table 8.2, the pseudo-static implementations were very slightly faster than the static ones, but only by 0.1%. The only significant gain was in Table 8.5, where pseudo-static implementations were 6.5% smaller than static implementations, but the number of circuits passing the verification algorithm was reduced by over a half. From these results, static and dynamic gates seem to be worth looking at, but pseudo-static and s2 gates do not appear to be worthwhile.

Tables 8.2–8.5 give the average changes when pseudo-static and dynamic gates are used relative to static gates, but the synthesis program always picks the best circuit according to a particular criterion. Table 8.6 gives the differences between the best static implementation of a circuit and the best pseudo-static and dynamic versions using the same state assignment method. It can be seen that pseudo-static gates only provide a significant advantage in the DME example with the MPP state assignment; the other results are too small to be of much significance. Results for dynamic gates are confused; dynamic implementations are always larger, but are sometimes faster and take less power.

Although the best pseudo-static gates seem to be a little better than the best static gates, the difference is small. To allow a more fair comparison with other asynchronous tools, only static gates or dynamic gates with keeper inverters will be considered from now on.

Example circuit	State assignment	Type of gate used	Best, compared to best static		
			Cycle time	Energy per cycle	Size (active area)
Latchc	MM	ps	−0.5 %	0.0 %	0.0 %
	MPP	ps	+1.2 %	−0.2%	−1.9 %
DME	MM	ps	−0.1 %	−1.3 %	−1.9 %
	MPP	ps	−6.5 %	+0.3%	−7.4 %
Latchc	MM	d	−1.3 %	+0.7 %	+37.3 %
	MPP	d	+0.1 %	+3.5 %	+37.0 %
DME	MM	d	−4.6 %	−4.1 %	+25.9 %
	MPP	d	+4.6 %	+3.9 %	+29.6 %

Table 8.6: How the best implementations produced are affected by the type of gate used

Type of state assignment	Cycle time	Energy per cycle	Size (active area)	Number that passed verification
TT	0.0 %	0.0 %	0.0 %	698/847
MM	0.0 %	−0.1 %	−0.1 %	753/847
MPP	+0.1 %	−14.2 %	−17.5 %	800/847
PP	+0.1 %	−14.1 %	−17.7 %	791/847

Table 8.7: Effects of the state assignment algorithm, on static latch controller circuits

8.2 Comparison of the state assignment algorithms

Section 6.3 described the four state assignment algorithms that were used, which were denoted TT, MM, PP and MPP respectively. TT was a standard Tracey [176] algorithm, slightly modified to allow some experiments to be carried out on flow tables with non-unique next-state entries. MM was Tracey's algorithm, modified using an extension of Unger's algorithm [178] to allow multiple concurrent inputs changes. It was expected that the MM algorithm will produce circuits that are more likely to work correctly and pass verification than the TT algorithm. PP and MPP were algorithms that attempted to reduce the number of state variables required and so to trade off speed for reduced area and power. Tables 8.7 and 8.8 give speed, power and size measurements for the four state assignment methods relative to the TT algorithm for static and dynamic versions of the latch controller; Tables 8.9 and 8.10 gives this information for the DME element.

For the latch controller, the MM state assignment algorithm produces more circuits that pass verification than the Tracey algorithm, as expected. Circuits produced using the MM algorithm are also slightly smaller and consume a little less power, which is a bonus. There is little to choose between the MPP and PP algorithms for the latch controller, but the MPP algorithm seems definitely better for the DME element; as expected, both produce circuits that are slower but much smaller and more power-efficient than the TT and MM algorithms.

For the static version of the DME element, the MM algorithm surprisingly pro-

Type of state assignment	Cycle time	Energy per cycle	Size (active area)	Number that passed verification
TT	0.0 %	0.0 %	0.0 %	764/847
MM	0.0 %	−0.2 %	−0.2 %	814/847
MPP	+1.7 %	−9.3 %	−12.7 %	813/847
PP	+1.8 %	−9.4 %	−12.8 %	807/847

Table 8.8: Effects of the state assignment algorithm, on dynamic latch controller circuits

Type of state assignment	Cycle time	Energy per cycle	Size (active area)	Number that passed verification
TT	0.0 %	0.0 %	0.0 %	478/848
MM	−4.4 %	−2.9 %	−4.6 %	444/848
MPP	+5.9 %	−21.7 %	−18.3 %	569/848
PP	+6.4 %	−18.4 %	−16.7 %	561/848

Table 8.9: Effects of the state assignment algorithm, on static DME element circuits

Type of state assignment	Cycle time	Energy per cycle	Size (active area)	Number that passed verification
TT	0.0 %	0.0 %	0.0 %	839/848
MM	−2.9 %	−2.0 %	0.0 %	840/848
MPP	+11.8 %	−30.7 %	−23.8 %	806/848
PP	+15.2 %	−19.9 %	−19.3 %	841/848

Table 8.10: Effects of the state assignment algorithm, on dynamic DME element circuits

duces less circuits that pass verification than the TT algorithm, but the circuits produced with the MM algorithm are significantly faster, smaller and more power-efficient than the TT algorithm. For dynamic DME elements, the MM algorithm produces one more correct circuit than the TT algorithm, but the circuits produced by the MM algorithm are still faster and more power efficient. There is again little to choose between the PP and MPP algorithms, but this time, both the PP and MPP algorithms are much worse than either the TT or MM algorithms when producing dynamic circuits.

These results suggest that, when attempting to find the fastest circuit, the best state assignment algorithm to use is the modified Tracey method, denoted MM. This algorithm is usually more reliable than the TT Tracey algorithm, and on average produces better circuits. When low power or small size are important, the MPP or PP algorithms should be used, with little to choose between them, although there are times when both of these algorithms are inferior to the MM algorithm. In the results in this chapter, both the MM and MPP algorithms were used and the best circuit for a particular task chosen.

8.3 Comparisons with other asynchronous tools

In this section, existing synthesis tools were used to create circuits for the latch controller, parallel component, DME element, nacking arbiter and loadable counter. These circuits were then SPICE simulated in a typical use of each circuit to determine their speed and power dissipation, and the circuits compared to ones produced by the work in this dissertation.

The tool described in this dissertation is directly comparable to the existing synthesis tools *petrify* [37], SIS [103, 164] and ASSASSIN [199]. FORCAGE [89] is only capable of synthesizing change diagrams, which cannot represent arbitration or choice behaviour such as that present in the nacking arbiter, the DME element or the loadable counter. 3D [203] uses an extended burst-mode specification style, a style in which it is not possible to describe a fully decoupled latch controller. The MEAT tool [33, 45] also uses burst-mode specifications, so that too cannot specify the latch controller example. These are the only tools that I was able to download.

SIS, ASSASSIN and *petrify* all take STGs as their input specification, so the five example circuits were all recast as STGs. It was found at this stage that the five examples were significantly harder to specify as STGs than they were to specify in the language described in Chapter 4. The synthesis tools each had options which can control the quality of their outputs:

- *Petrify*[1] can be used with the `-no -csc -cg -eqn out.eqn` option string to produce speed-independent circuits. The option `-redc` can be used to enable concurrency reduction, as described in [39]. The `-redc` option often serialises transitions in the environment, which results in slow circuits, so the option `-slowenv` can be specified to keep environment transitions concurrent. The form of the output is a network of static gates followed by inverters, similar to the static implementations described in this dissertation.

- ASSASSIN is a command-driven shell; commands are executed rather than passed in as options on the command line. The STG is read in to ASSASSIN by the `assa_read_stg` command. An implementation can then be derived by using the commands `assa_stg_to_sg` to create a state graph, `assa_opt_sa` and `assa_gen_sa` to assign state variables to remove CSC conflicts, and then `assa_haz_logic` to create a hazard-free implementation. This implementation was then converted to a network of static or dynamic complex gates by hand, to allow the circuits to be compared with ones produced by the work in this dissertation. Some circuits were too large to be converted by hand; these circuits were simply marked in the results as being too big.

  Two forms of concurrency reduction are supported in ASSASSIN: STG locking is performed by the the `assa_lock_stg` command, used before the STG is converted to a SG. The `assa_red_sg` command does state graph reduction, and is used before state assignment.

[1] *Petrify* version 3.5 was used in this dissertation; the `-redc` option does not appear in version 4.0.

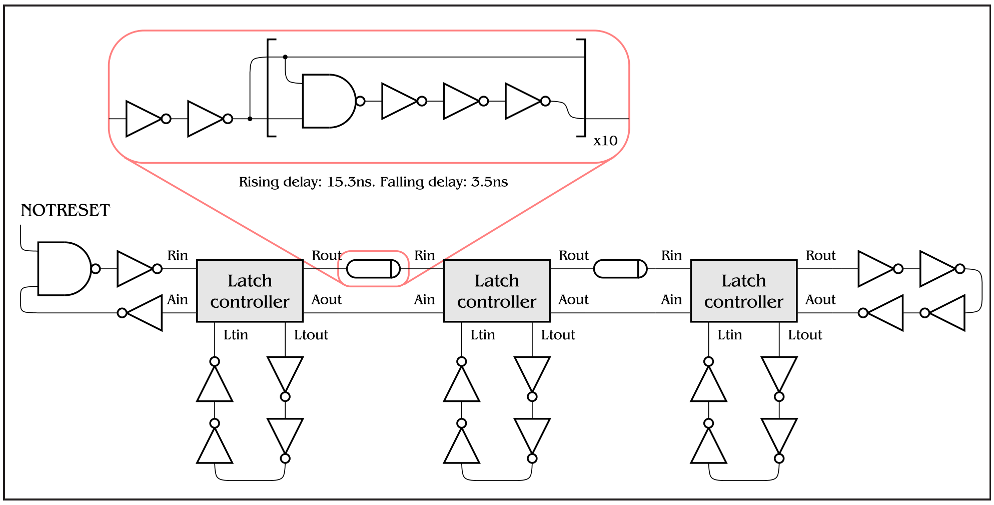


Figure 8.1: Circuit used to simulate a typical use of the latch controller

- SIS could not be made to synthesize any of the five example circuits with any set of commands.

8.3.1 The latch controller

The typical use of the latch controller was taken to be the circuit shown in Figure 8.1. This is a three-stage pipeline with a delay of 15.3 ns on rising edges of the forward-going request wire, corresponding to the processing delay of a datapath, and a delay of 3.5 ns on the falling edge, corresponding to the datapath precharge delay. Results from SPICE runs carried out with the implementations produced by ASSASSIN and *petrify*, both with and without concurrency reduction, are shown in Table 8.11. Also shown are the three latch controllers from Furber and Day [59].

Two circuits were produced using the tools described in this dissertation. The first was the fastest circuit found. The smallest and lowest power circuits are trivial, because it is possible to create a zero-size circuit by simply connecting `Rin` to `Ltout`, `Ltin` to `Rout` and `Aout` to `Ain`. Instead, the lowest power fully decoupled circuit was found, where all circuits that have a cycle time of less than twice the forward processing delay are taken to be fully decoupled.

It can be seen that the fastest latch controller produced by the new tool is a little slower, a little bigger and a little more power-hungry than Furber and Day's fully decoupled controller. This shows that the proposed synthesis method is not capable of producing circuits that are as good as hand-crafted speed-independent circuits. The new tool is however noticeably better than current automated SI design approaches.

The second circuit produced from the new tool in Table 8.11 is smaller than the semi-decoupled controller, but provides performance that is nearly as good as the fully-decoupled version. The small size is partially due to the fact that fully static gates were used; the two keeper inverters in Furber and Day's semi-decoupled controller are quite large. Logic equations for this circuit are:

Implementation	Cycle time	Energy/cycle	Size
Furber and Day:			
Simple	36.7 ns	69.8 pJ	30
Semi-decoupled	37.9 ns	78.7 pJ	60
Fully-decoupled	25.3 ns	92.2 pJ	117
petrify:			
No concurrency reduction	29.3 ns	112.5 pJ	129
With concurrency reduction (-redc)	63.0 ns	84.3 pJ	45
With -redc and -slowenv	27.5 ns	94.0 pJ	87
ASSASSIN:			
Without concurrency reduction	State assignment failed		
With assa_lock_stg	Fails; wrong kind of STG		
With assa_red_sg	49.2 ns	82.8 pJ	59
Proposed tool:			
Unreduced	Fails verification		
Highest speed	25.6 ns	94.4 pJ	127
Lowest power (cycle $<$ 30.6ns)	29.1 ns	82.3 pJ	54

Table 8.11: Latch controller implementations from various tools

Implementation	Cycle time	Energy/cycle	Size
petrify:			
No concurrency reduction	27.5 ns	141.5 pJ	141
With concurrency reduction (-redc)	85.4 ns	127.6 pJ	84
With -redc and -slowenv	25.1 ns	121.8 pJ	81
ASSASSIN:			
Without concurrency reduction	Too big		335
With assa_lock_stg	23.1 ns	118.8 pJ	92
With assa_red_sg	88.2 ns	145.5 pJ	109
Proposed tool:			
Unreduced	Fails verification		
Highest speed	23.8 ns	107.1 pJ	39
Lowest power and smallest	24.4 ns	100.8 pJ	21

Table 8.12: Parallel component implementations from various tools

$$
\begin{aligned}
\texttt{Ltout} &= \overline{\texttt{Aout}}\texttt{.(Rin + Ltout)} \\
\texttt{Rout} &= \texttt{Ain + Rout.Ltin} \\
\texttt{Ain} &= \texttt{Ain.Rin + }\overline{\texttt{Rout}}\texttt{.Ltin}
\end{aligned}
$$

8.3.2 Parallel component

Table 8.12 shows the results of running *petrify*, ASSASSIN and the new tool on the parallel component. SPICE timings were carried out using the circuit shown back

Implementation	Cycle time	Energy	Size
petrify:			
No concurrency reduction	21.2 ns	49.8 pJ	174
With concurrency reduction (`-redc`)	28.6 ns	50.0 pJ	168
With `-redc` and `-slowenv`	24.1 ns	57.2 pJ	174
ASSASSIN:			
Without concurrency reduction	24.6 ns	57.4 pJ	239
With `assa_lock_stg`	Fails, not right kind of STG		
With `assa_red_sg`	Fails, SIGSEGV		
Proposed tool:			
Highest speed and smallest	19.1 ns	53.8 pJ	156
Lowest power	21.5 ns	44.2 pJ	169

Table 8.13: Nacking arbiter implementations from various tools

in Figure 7.18, with each of the delays replaced by the same circuit that was used to create a delay in Figure 8.1. Most of the power that was dissipated in each cycle of the circuit comes from the delay lines that were used, rather than the control circuits themselves, which is why there is little variation in the power consumed by each of the different implementations. It can be seen that ASSASSIN produced the fastest circuit, shaving 3% off the best cycle time that can be produced with the new tool. The new tool did produce the smallest and most energy-efficient circuits, however.

8.3.3 Nacking arbiter

The circuit used for SPICE timings of the nacking arbiter was that shown in Figure 7.21, for which the results are shown in Table 8.13. Concurrency reduction was not possible in this case, so the only difference between the two circuits produced by the new tool is the state assignment method used—the MM method was used in the fastest circuit, which was also the smallest, and the MPP method used in the circuit which took least energy per cycle. Both circuits show some improvement over circuits produced by the other tools.

8.3.4 DME element

The circuit in Figure 7.20 was used in SPICE simulations of the DME element, and the results given in Table 8.14. ASSASSIN was unable to implement the DME element, due to internal errors and segmentation faults. Petrify was not able to use the `-redc` and `-slowenv` options, but did create circuits using no additional options and using the `-redc` option. The effects of concurrency reduction are clearly demonstrated in this example; the unreduced circuit produced by the new tool is much larger and more power-hungry than the reduced circuits, although not much slower.

The unreduced DME circuit can be seen to be faster than the circuit produced

Implementation	Response time	Energy	Size
petrify:			
No concurrency reduction	14.5 ns	34.4 pJ	138
With concurrency reduction (`-redc`)	21.7 ns	27.8 pJ	129
With `-redc` and `-slowenv`	Fails, suggests option `-timed`		
ASSASSIN:			
Without concurrency reduction	Fails, internal error		
With `assa_lock_stg`	Fails, not right kind of STG		
With `assa_red_sg`	Fails, SIGSEGV (stack growth)		
Proposed tool:			
Unreduced	11.8 ns	42.8 pJ	249
Highest speed	10.7 ns	26.2 pJ	195
Lowest power	11.2 ns	17.4 pJ	102
Smallest	16.9 ns	22.9 pJ	81

Table 8.14: DME implementations from various tools

Implementation	Time for count=5	Energy	Size
petrify:			
No concurrency reduction	78.9 ns	149.1 pJ	301
With concurrency reduction (`-redc`)	90.1 ns	135.3 pJ	285
With `-redc` and `-slowenv`	87.6 ns	143.0 pJ	273
ASSASSIN:			
Without concurrency reduction	Too big		650
With `assa_lock_stg`	Fails, not right kind of STG		
With `assa_red_sg`	Too big		531
Proposed tool:			
Unreduced	38.6 ns	75.4 pJ	173
Highest speed and lowest power	38.6 ns	74.0 pJ	175
Smallest	48.5 ns	79.4 pJ	144

Table 8.15: Loadable counter implementations from various tools

by *petrify*, but it is much larger and takes more power. This is probably typical of fundamental mode circuits, but the unreduced circuits often fail the verification stage so firm conclusions cannot be drawn.

8.3.5 Loadable counter

Figure 7.19 shows the circuit that was used to compare the loadable counter implementations, and Table 8.15 gives the results. The test circuit was simply using the loadable counter implementation to count up to five. It can be seen that the new tool shows a clear advantage over the other tools when synthesizing this circuit; circuits produced are about twice as fast, half the size and take half the power of circuit produced by *petrify*, and they are a quarter of the size of circuits produced

Example circuit	Optimized for	Percentage improvement on best of other tools
Latch controller	Speed	14% faster
Latch controller	Power (cycle < 30.6ns)	27% less power
Parallel	Speed	*3% slower*
Parallel	Power	15% less power
Parallel	Size	74% smaller
Nacking arbiter	Speed	11% faster
Nacking arbiter	Power	11% less power
Nacking arbiter	Size	7% smaller
DME	Speed	35% faster
DME	Power	37% less power
DME	Size	37% smaller
Loadable counter	Speed	104% faster
Loadable counter	Power	50% lower power
Loadable counter	Size	47% smaller

Table 8.16: Summary of results

Example circuit	Time to create static circuit	Time to create dynamic circuit
Latch controller	0 hr 52 min	0 hr 4 min
Parallel	0 hr 51 min	0 hr 5 min
Nacking arbiter	4 sec	0.6 sec
DME	4 hr 59 min	3 hr 51 min
Loadable counter	1 hr 5 min	0 hr 1 min

Table 8.17: Total run-time for each example

by ASSASSIN. The circuits from ASSASSIN were so large that they were not tested in SPICE.

8.3.6 Summary

Table 8.16 shows a summary of the results in Tables 8.11–8.15, which shows that the tool presented in this dissertation usually produces circuits that are significantly faster, smaller or dissipate less power than other approaches, depending on what factors are important to the designer. The gains are moderate for some examples, such as the nacking arbiter, but more pronounced for the DME element and loadable counter. There is one exception to this: ASSASSIN shows a surprising ability to synthesize an exceptional parallel controller, which is faster that the best circuit the new tool can produce.

Table 8.17 gives the run-times of synth on the five example circuits. L2b and prune take under five seconds on each example. It can be seen that synth is not interactive, but can synthesize circuits if left overnight. The synth program is easily parallelizable, so an alternative to an overnight run is to use a large number of machines concurrently.

8.3.7 Estimated timings

The estimated timings of the DME, nacking arbiter and loadable counter circuits were found to be 4.8% out on average from the SPICE timings. This is partly due to the fact that arbiters are analogue components, which are not handled particularly well by the gate level simulator. However, this is still a respectable accuracy. Estimated timings for the latch controller and parallel component cannot be compared with SPICE timings, because a model of a fixed pure delay was used to estimate the cycle time, but this model does not exist in SPICE. The average discrepancy between the estimated and actual power taken by the DME, nacking arbiter and loadable counter circuits was 6.4%, although again this could have been adversely affected by arbiters.

8.4 Results on other circuits

The tool described in this dissertation was also used to synthesize some of the standard SIS benchmark STGs. An exact environment was not specified for these circuits, so it is not possible to give reliable speed and power estimates. Instead, these circuits were optimized to be as small as possible. Table 8.18 gives the results of the new tool on each of the benchmarks used, and also the results of *petrify* with the option `-redc1`, which steers the concurrency reduction algorithm towards small circuits. There is no clear winner in this test in terms of the size of the circuits produced. It was not expected that the new tool would produce particularly good implementations from STG specifications, so this is not a surprise.

The run-times for the new tool are, unfortunately, worryingly high in some cases—*sbuf-ram-write* took just over 7.6 hours to synthesize. This is because some examples have a large number of pruned blue diagrams, and synth attempts to implement each diagram. Table 8.19 gives the number of blue diagrams produced by prune for each of the examples, which shows that *sbuf-ram-write* had over 200 pruned diagrams. Large blue diagrams also take longer to synthesize, as *pe-send-ifc* shows.

Benchmark	Proposed tool		*petrify*	
	size of circuit	time taken (s)	size of circuit	time taken (s)
alloc-outbound	96	6.2	87	1.1
atod	39	9.7	48	2.0
isend	151	5.5	150	21.8
master-read	Fails		141	480
mr1	60	25000	48†	6.9
mr2	66	4600	57†	30.0
total	126	30000	105†	36.9
mp-forward-pkt	81	0.3	81	1.6
nak-pa	54	1500	63	13.5
nowick	63	1.0	69	1.5
pe-send-ifc	140	40.2	123†	99.8
ram-read-sbuf	84	600	102	3.2
rcv-setup	45	0.1	45	0.3
rlm	45	0.2	45	0.1
sbuf-ram-write	84	27000	117	21.2
sbuf-read-ctl	78	0.5	72	0.5

†requires `-timed`

Table 8.18: Results on some of the SIS benchmarks

Benchmark	Pruned diagrams	Size of unpruned diagram
alloc-outbound	1	8
atod	34	11
isend	1	17
master-read	∼700 000	108
mr1	2 310	30
mr2	298	24
mp-forward-pkt	1	8
nak-pa	58	12
nowick	1	7
pe-send-ifc	1	33 or 35
ram-read-sbuf	15	17
rcv-setup	1	7
rlm	1	6
sbuf-ram-write	264	18
sbuf-read-ctl	1	7

Table 8.19: Recap of number of pruned blue diagrams

Summary and Conclusions 9

9.1 Summary

This dissertation has described a new synthesis tool, which augments existing algorithms for finite state machine synthesis with a number of new techniques. A front-end specification language provides an intuitive and easy-to-use interface to the synthesis tool. Concurrency reduction is carried out on an intermediate representation of the specification, to produce a large number of possible circuit behaviours. Finally, verification and timing analysis are performed to choose the best circuit for a particular application. The delay model used was fundamental mode, with the constraint that the circuits produced must have a delay-insensitive interface. In practice, this means that the environment has some minimum response time, and that concurrent inputs can occur with arbitrary delays between them.

Chapter 4 described the front-end specification language and the procedure that was used to translate this into an intermediate representation called a blue diagram. Blue diagrams are essentially compacted state graphs, in which output orderings are irrelevant and are not specified. The specification language is based on STG fragments, but also includes constructs that a circuit designer would find useful, such as `and` and `or` relations between transitions, handshake declarations, and `if` and `arbitrate` statements. STGs could not be used directly as a specification, because STGs use the speed-independent model whereas circuits produced in this dissertation are fundamental mode with a delay-insensitive interface. Specifications in the proposed language are easier to create and much easier to read than equivalent STG specifications that can be used as an input to other comparable tools, such as ASSASSIN or *petrify*. There is also more scope for giving helpful error messages when the designer inputs an invalid specification. The specification is first translated into a Petri net, and then simulated to form a blue diagram.

Chapter 5 explained the concurrency reduction operation on blue diagrams, and gave some examples of its use. This operation can be performed efficiently without requiring a full state graph to be held in memory, and is reasonably natural, although probably not as natural as the concurrency reduction work of Ykman-Couvreur et al. [197] or the forward reduction algorithm of Cortadella et al. [39]. However, it is slightly more powerful than either of those two algorithms, but the differences may be small in practice. I believe that it is as powerful as the backward reduction algorithm of Cortadella et al. [39], but that algorithm has not been implemented yet, and it is not clear whether it can be used effectively.

The concurrency reduction process produces a number of different blue diagrams, which are synthesized using the techniques presented in Chapter 6. The blue diagrams were first converted to flow tables, which is a trivial step. The flow table reduction algorithm of Puri and Gu [148] was used, because this appears to be a little faster and better than the standard reduction program, STAMINA [153]. State assignment was performed using an extension of the modification to Tracey's algorithm [176] suggested by Unger [178], which makes sure that concurrent input changes do not cause the circuit to malfunction. Although this modification adds constraints to the state assignment algorithm, which could possibly increase the number of state variables required, the circuits produced by the modified algorithm are usually a little smaller, faster and consume a little less power than Tracey's original algorithm. Another state assignment algorithm was also used, which reduces circuit size and power in exchange for decreasing the speed of the implementation. Scoring functions were provided to find the best flow table reduction if several were produced, and also to find the best way to shrink compatibles and the best state assignment. A new form of CMOS gate was explored, the pseudo-static gate, but this was found to be inferior to more conventional static gates. Dynamic gates with keeper inverters were found to offer lower power solutions at the expense of size. SOP/SOP expressions were used in the static gates, rather than the more conventional approach of making the P-tree be the dual of the N-tree, because of better hazard properties.

Timing and verification were described in Chapter 7. A new method of estimating waveform slopes in a circuit was used to provide reasonably accurate estimates of gate delays. The gate-level simulator was combined with the binary bi-bounded simulation algorithm described in Brzozowski and Seger [21] to produce a verification algorithm. The circuit under test was composed with a model of its environment, and then simulated while allowing all gate delays to vary by some fixed amount, by default 20%, from their nominal amount. The behaviour of the circuit was then checked against the specification. Hazards are allowed on internal circuit nodes as long as the hazard does not propagate to a primary output of the circuit. Timing was performed by allowing the designer to specify an example use of the circuit in a language similar to Verilog. This allows a better measure of the speed of a circuit, compared to other methods such as the path delay estimation used in ASSASSIN.

The results in Chapter 8 are promising, especially for the DME element and the loadable counter. Circuits produced are almost always better than those produced by existing synthesis tools. The program takes a long time to run, but this is not surprising, and probably not much of a handicap; when the specification has been accepted as correct, the synthesis program can be run overnight or in parallel on many machines. All the SIS STG benchmarks that were tried could be synthesized, although the *master-read* example had to be split into two parts.

9.2 Conclusions

In Chapter 1, the stated aims of the dissertation were:

1. *To create a front-end description that is powerful enough for almost all real-world circuits and is simple to use.*

   All the example circuits could be described using the proposed specification language, and the descriptions were much clearer than equivalent STG specifications in the standard file format that is used for *petrify* and ASSASSIN.

2. *To compile this specification into the intermediate form of a blue diagram.*

   All examples tried were compiled correctly into blue diagrams.

3. *To show that exhaustive enumeration of concurrency-reduced blue diagrams is possible within a reasonable time.*

   The execution times that were given in Chapter 8 show that the time requirements are reasonable, although the tool is certainly not interactive. The specification front end and concurrency reduction programs are fast, taking a few seconds each; the synthesis program can take anywhere between a second and about five hours on a moderate workstation. Running the tool overnight is not really a problem, especially if this produces circuits that are 10%, 20% or even 50% faster than comparable tools.

4. *To show that the concurrency-reduced blue diagrams can be synthesized into circuit modules and verified as correct given bounds on the environment response times.*

   Implementations for all the example STGs could be found that passed the verification algorithm. It is, however, very difficult to test the verification algorithm itself. Several bugs in the verifier were picked up by the simulator, because the simulator checks for strange behaviour that may be caused by an incorrect circuit, but the verifier cannot be inferred to be correct just because it has not passed an incorrect circuit for a while.

5. *To show that circuits produced tend to be superior to other asynchronous tools, in terms of the scoring function given by the designer.*

   With the exception of the parallel component, for which ASSASSIN created a very fast circuit, and the latch controller, for which Furber and Day created the best circuit by hand, this has been shown to be true. Circuits produced by the tool described in this dissertation have usually been better, sometimes by a factor of two, than circuits produced by current synthesis tools.

Several questions can be asked about the work in this dissertation. Firstly, is the use of fundamental mode justified? The results in Chapter 8 suggest that fundamental mode circuits are indeed smaller, faster and more power-efficient than speed-independent circuits, but they also have their drawbacks. Fundamental mode circuits are not guaranteed to work correctly under every distribution of gate delays, so they must be verified for a particular target technology. If the circuits are implemented using a different technology, the circuits must be re-verified. The best circuit for a particular task may actually change when moving to a different technology. Fundamental mode circuits are a two-edged sword; they appear to be better, but they are more difficult to produce and more likely to fail in the presence of process variations. The best approach to take when designing large circuits for high speed operation would be to use fundamental mode circuits for time-critical parts, and speed-independent circuits for everything else.

Another question that may be asked is whether exhaustive concurrency reduction is better than more conventional methods, which take small concurrency-reducing steps guided by a heuristic until a single solution is found. This is a difficult question to answer, because it is not possible to isolate the effects of the concurrency reduction process from the effects of the fundamental mode synthesis style. It can be seen that exhaustive concurrency reduction is not overly time-consuming; all the example circuits can be synthesized overnight. The most time-consuming part of the synthesis is verifying the circuits produced, so an algorithm based on exhaustive concurrency reduction in speed-independent circuits may run significantly faster. To answer this question fully, the fundamental mode synthesis routines would have to be replaced by a known speed-independent synthesis program, such as *petrify*. Then, a direct comparison between existing methods and the exhaustive approach can be made. Such an investigation is beyond the scope of this thesis.

A new specification language was presented in this dissertation, but the disadvantage of any new specification language is that it is an unknown, and needs to be learnt. It was found to be much easier to use than STGs, so its ease of use probably outweighs the drawback of having to learn how to use it. It would be possible to use the front-end as an interface to other synthesis tools, because the language is translated into a Petri net internally, and tools such as *petrify* and ASSASSIN accept Petri nets. Anything that increases the readability of specifications is likely to be welcomed by circuit designers. This also offers the possibility of integrating *petrify* into a Verilog framework, which may produce a unified synthesis environment.

The gate level simulator was found to have an average accuracy better than 5% on the examples tried, which would appear to be very good. Several published papers give accuracy results in the 5–10% region, but they often are limited to a few kinds of basic gate, such as inverters and 2-input and 3-input NAND and NOR gates. This simulation algorithm achieves 5% accuracy on circuits containing large complex gates, keeper inverters and analogue arbitration circuitry. It would be good to compare the proposed method with commercial gate-level simulators, but accuracy figures for these are not usually released.

It is difficult to draw conclusions about the verification algorithm without using the circuits produced in a real design; if all circuits produced are found to work in real designs, then the verifier is a success, if one or more fail, then it is faulty and should be abandoned. The combination of a fairly accurate timing simulator and the binary bi-bounded simulation algorithm is very powerful, and could have applications outside the synthesis tool described. A stand-alone version of the verifier could be used to check timing assumptions in published designs, and provide a higher degree of confidence than a SPICE simulation. The verifier could also produce a degree-of-confidence result, by raising the amount by which gate delays are permitted to vary until the circuit fails; a circuit for which the parameter must be 75% before the circuit fails would be more robust than a circuit for which only 50% causes failure.

It had been hoped that the tool would produced all three circuits from Furber and Day's paper [59], along with many other circuits, and then pick the one with the lowest cycle time. So why was the fully decoupled controller better than any produced with this tool? It turns out that the fully decoupled controller cannot be produced by the algorithms outlined in this dissertation—there are two states of the fully decoupled circuit that are compatible in the flow table sense, but have a different value of a state variable. Any flow table reduction algorithm will combine these two states, and so can not produce the fully decoupled circuit. It is unfortunate that the unofficial goal of this dissertation—to produce a better latch controller than the fully decoupled controller—is unattainable.

9.3 Further Work

This dissertation opens up many possibilities for further research:

- The *master-read* example had to be split into two halves, otherwise there were too many concurrency-reduced blue diagrams. When this was done, *petrify* was able to find a solution that was 25% smaller than when the original specification was used. This suggests that automatic cleaving of STGs, similar to Chu's seminal contraction algorithms [26], would be a useful first step for many synthesis algorithms.

- Currently, each blue diagram after concurrency reduction is synthesized, verified and timed. The gate-level simulator was written to allow modules to be specified by blue diagrams, rather than collections of gates. Relatively good speed estimates can be found by simulating the composition of a blue diagram with its environment before synthesis; then, only the most promising diagrams need be synthesized. Preliminary work shows that this is good for circuits that need to be fast, but it is difficult to determine the size or power of the resulting implementation just by looking at the blue diagram; more work is required to find good heuristics for power and size.

- It was said at the end of Chapter 6 that Schmitt triggers on the inputs to the circuit and buffers on all primary outputs would make the circuits more robust. This has not been implemented, but should be a fairly simple extension.

- The front end can be modified to act as an interface to a speed-independent synthesis tool, such as *petrify*.

- It would be good to rewrite the core of the synthesis routines to use a speed-independent approach, so that the effects of the fundamental mode synthesis style can be fully evaluated. This would also remove the need for a verification stage.

- The verifier could be expanded to handle standard cell or Xilinx designs as well as CMOS complex gates, and make into a stand-alone tool that could be used in other synthesis methodologies.

- It was envisaged at the start of the project that layout could also be tackled. This would allow the tool to produce a standard cell for a latch controller, for example. This cell could then be treated exactly as any other cell by place-and-route tools.

- Technology mapping issues have not been addressed: are CMOS complex gates a usable implementation medium? Will mapping them onto standard library gates reduce the number of circuits that pass verification?

- Observing arbiters could be used to increase the response time of some circuits that require arbitration. For example, if a DME circuit does not have the token when it receives an `lr+` or `ur+`, it does not need to make a decision between the two alternatives to know that it needs to request a token from the DME element on its right. The arbitration can be carried out in parallel with the request for the token.

Glossary

Alpha Digital's cutting edge processor. The Alpha 21264 is probably the fastest mainstream uniprocessor available.

ALU Arithmetic and Logic Unit. The part of a processor where calculations occur.

BiCMOS A combination of CMOS and bipolar technologies, designed to achieve the high speed of bipolar logic with the low power consumption and high density of CMOS logic.

CAD Computer-Aided Design. Specifically refers to the design of integrated circuits using specialized software packages.

CMOS Complementary MOS, an integrated circuit technology using both P and N channel MOSFETs. The self-aligning property of CMOS and its relatively low power consumption have made it the most popular fabrication technology for almost all mainstream digital circuits.

DCC Philips' Digital Compact Cassette. The decoding hardware of DCC players is a good example of where asynchronous circuits can provide superior solutions to conventional synchronous circuits.

DSP Digital Signal Processor.

FIFO First-In First-Out. A type of queue or pipeline where the order of items of data is preserved.

FPGA Field programmable gate array, essentially a programmable chip. FPGAs are composed of a number of CLBs (combinational logic blocks), which can generate a large number of logic functions and be connected together to build complex circuits. Most are synchronous, but asynchronous structures can just about be built.

GaAs Gallium Arsenide, one of the III-IV semiconductors. III-IV materials produce faster circuits and have other advantages, but cost significantly more for production.

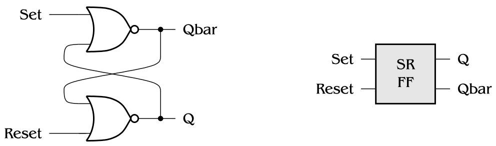
MOSFET Metal-Oxide-Semiconductor Field Effect Transistor. Name refers to the original method of fabrication of the device. MOSFETs act like small switches, and are the building blocks from which CMOS integrated circuits are made.

PLA Programmable Logic Array, an off-the-shelf chip that can implement sum-of-product expressions with feedback efficiently, and can be programmed cheaply.

Schmitt Trigger An analogue circuit with positive feedback, which can generate sharp edges from a slow ramp input even in the presence of noise.

SPICE Simulation Program with Integrated Circuit Emphasis. An analogue simulator which is widely regarded as the most accurate way, but also the slowest way, to find the behaviour of a ciruit.

SR flip-flop Set-reset flip-flop, a circuit element which can store a single bit of information, which can either be set or reset by giving one of two inputs.



Verilog A widely-used hardware description language, loosely based on C.

Bibliography

[1] Douglas B. Armstrong, Arthur D. Friedman, and Premachandran R. Menon. Design of asynchronous circuits assuming unbounded gate delays. *IEEE Transactions on Computers*, C-18(12):1110–1120, December 1969.

[2] Aaron Ashkinazy, Doug Edwards, Craig Farnsworth, Gary Gendel, and Shiv Sikand. Tools for validating asynchronous digital circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 12–21, November 1994.

[3] P. Beerel and T.H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, November 1992.

[4] P. A. Beerel, K. Y. Yun, and W. C. Chou. Optimizing average-case delay in technology mapping of burst-mode circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

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

[6] Peter A. Beerel, Jerry R. Burch, and Teresa H.-Y. Meng. Sufficient conditions for correct gate-level speed-independent circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 33–43, November 1994.

[7] Wendy Belluomini and Chris J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 88–100. IEEE Computer Society Press, April 1997.

[8] R. G. Bennetts. Improved method of prime C-class derivation in the state reduction of sequential networks. *IEEE Transactions on Computers*, 20:229–231, February 1971.

[9] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

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

[11] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. A fully-asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid-State Circuits*, 29(12):1429–1439, December 1994.

[12] Kees van Berkel, Ferry Huberts, and Ad Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Asynchronous Design Methodologies*, pages 99–106. IEEE Computer Society Press, May 1995.

[13] M. R. C. M. Berkelaar, P. H. W. Buurman, and J. A. G. Jess. Computing the entire active area / power consumption v. delay tradeoff curve for gate sizing with a piecewise linear simulator. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 15(11):1424–1434, November 1996.

[14] W. J. Bowhill et al. Circuit implementation of a 300-MHz 64-bit second-generation CMOS Alpha CPU. *Digital Technical Journal*, 7(1):100–115, 1995.

[15] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[16] W. F. Brinkman, D. E. Haggan, and W. T. Troutman. A history of the invention of the transistor and where it will lead us. *ieeejssc*, 32(12), dec 1997.

[17] L. M. Brocco, S. P. McCormick, and J. Allen. Macromodelling CMOS circuits for timing simulation. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 7(12), December 1988.

[18] R. B. Brown et al. Overview of complementary GaAs techonology for high-speed VLSI circuits. *IEEE Transactions on VLSI Systems*, 6(1):47–51, March 1998.

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

[20] J. A. Brzozowski and K. Raahemifar. Testing C-elements is not elementary. In *Asynchronous Design Methodologies*, pages 150–159. IEEE Computer Society Press, May 1995.

[21] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[22] Jerry R. Burch. Delay models for verifying speed-dependent asynchronous circuits. In *ACM Int. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, March 1992.

[23] B. S. Carlson and S. J. Lee. Delay optimisation of digital CMOS VLSI circuits by transistor reordering. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 14(10):1183–1192, 1995.

[24] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[25] J. F. Chappel and S. G. Zaky. A delay-controlled phase-locked loop to reduce timing errors in synchronous/asynchronous communications links. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1994.

[26] T.-A. Chu, C. K. C. Leung, and T. S. Wanuga. A design methodology for concurrent VLSI systems. In *Proc. International Conf. Computer Design (ICCD)*, pages 407–410. IEEE Computer Society Press, 1985.

[27] Tam-Anh Chu. On the models for designing VLSI asynchronous digital circuits. *Integration, the VLSI journal*, 4(2):99–113, June 1986.

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

[29] Tam-Anh Chu and Narayana S. Mani. CLASS: A CAD system for automatic synthesis and verification of asynchronous finite state machines. In *Proc. Hawaii International Conf. System Sciences*, volume I. IEEE Computer Society Press, January 1993.

[30] Tam-Anh Chu and Narayana S. Mani. CLASS: A CAD system for automatic synthesis and verification of asynchronous finite state machines. *Integration, the VLSI journal*, 15(3):263–289, October 1993.

[31] Edwin C.Y. Chung and Lindsay Kleeman. Metastable-robust self-timed circuit synthesis from live safe simple signal transition graphs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 97–105, November 1994.

[32] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336, Atlantic City, NJ, 1967. Academic Press.

[33] B. Coates, A. Davis, and K. S. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *Proc. VII Banff Workshop on Asynchronous Hardware Design*, 1993.

[34] Bill Coates, Al Davis, and Ken Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341–366, October 1993.

[35] W. S. Coates, J. K. Lexau, I. W. Jones, S. M. Fairbanks, and I. E. Sutherland. A fifo data switch design experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 4–17, 1998.

[36] Deep submicron seminar. Compass Design Automation Inc., 1995.

[37] J. Cortadella. The *petrify* home page at Univsitat Politècnia de Catalunya. http://www.ac.upc.es/˜vlsi/petrify/petrify.html.

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

[39] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Automatic handshake expansion and reshuffling using concurrency reduction. In *Workshop on Hardware Design and Petri Nets (HWPN'98)*, June 1998.

[40] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing petri nets from state-based models. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 164–171, 1995. ftp://ftp.ac.upc.es/archives/cad/petrify/UPC-DAC-95-09.ps.gz.

[41] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor, and Alexandre Yakovlev. Decomposition and technology mapping of speed-independent circuits using Boolean relations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, November 1997.

[42] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Methodology and tools for state encoding in asynchronous circuit synthesis. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[43] Jordi Cortadella, Luciano Lavagano, Peter Vanbekbergen, and Alexandre Yakovlev. A systematic approach to design asynchronous circuits with internal conflicts. In *Proc. ACiD-WG Workshop on Testing and Design for Testability, Aveiro, Portugal*, 1994.

[44] Jordi Cortadella, Alexandre Yakovlev, Luciano Lavagano, and Peter Vanbekbergen. Designing asynchronous circuits from behavioral specifications with internal conflicts. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 106–115, November 1994.

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

[46] A. Davis, B. Coates, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. In *Proc. Hawaii International Conf. System Sciences*, volume I, pages 409–418. IEEE Computer Society Press, January 1993.

[47] Paul Day and J. Viv Woods. Investigation into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[48] J. B. Dennis and S. S. Patil. Speed-independent asynchronous circuits. In *Proc. Hawaii International Conf. System Sciences*, pages 55–58, 1971.

[49] S. C. DeSarker, A. K. Basu, and A. K. Choudhury. Simplification of incompletely specified flow tables with the help of prime closed sets. *IEEE Transactions on Computers*, C-18:953–956, October 1969.

[50] A. Devgan. Transient simulation of integrated circuits in the charge-voltage plane. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 15(11):1379–1390, November 1996.

[51] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[52] D. L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuit. In J. Sifakis, editor, *Proceedings of International Workshop on Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science*, volume 407, pages 197–212. Springer-Verlag, 1990.

[53] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[54] B. Doyle, M. Bourcerie, J.-C. Marchelaux, and A. Boudou. Interface state creation and charge trapping . . . during hot-carrier stressing of n-MOS transistors. *IEEE Transactions on Electronic Devices*, 37(3):744–754, March 1990.

[55] Karl M. Fant and Scott A. Brandt. NULL conventional logic: A complete and consistent logic for asynchronous digital circuit synthesis. In *International Conference on Application-specific Systems, Architectures, and Processors*, pages 261–273, 1996.

[56] P. David Fisher and Sheng-Fu Wu. Race-free state assignments for synthesizing large-scale asynchronous sequential logic circuits. *IEEE Transactions on Computers*, 42(9):1025–1034, September 1993.

[57] S. Furber. Computing without clocks: Micropipelining the ARM processor. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 211–262. Springer-Verlag, 1995.

[58] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[59] Stephen B. Furber and Paul Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[60] M. R. Garey and D. S. Johnson. *Computers and Intractability - a Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[61] Jim D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 181–207. Elsevier Science Publishers, 1993.

[62] E. Grass and S. Jones. Improved current-sensing completion detection (CSCD) circuits. In *Proc. ACiD-WG Workshop on Testing and Design for Testability, Aveiro, Portugal*, 1994.

[63] E. Grass, R. C. S. Morling, and I. Kale. Activity monitoring completion detection (AMCD): A new single rail approach to achieve self-timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[64] A. Grasselli and F. Luccio. A method for minimising the number of internal states in incompletely specified sequential networks. *IEEE Transactions on Electronic Computing*, EC-14:350–359, June 1965.

[65] A. Grasselli and F. Luccio. Some covering problems in switching theory. In G. Biorci, editor, *Network and Switching Theory*, pages 536 – 557. Academic Press, New York and London, 1968.

[66] P. E. Gronowski, W. J. Bowhill, M. K. Gowan, and R. L. Allmon. High-performance microprocessor design. *IEEE Journal of Solid-State Circuits*, 33(5):676–686, May 1998.

[67] R. K. Gupta and S. Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, 14(2):72–80, April–June 1997.

[68] P. Hallam, P. J. Mather, and M. Brouwer. CMOS process independent propagation delay. *Electronics Letters*, 31(9):702–703, April 1995.

[69] Scott Hauck. Asynchronous design methodologies: An overview. Technical Report TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.

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

[71] Scott Hauck, Steven Burns, Geatano Borriello, and Carl Ebeling. An FPGA for implementing asynchronous circuits. *IEEE Design & Test of Computers*, 11(3):60–69, 1994.

[72] N. Hedenstierna and K. O. Jeppson. CMOS circuit speed and buffer optimisation. *IEEE Transactions on Computer Aided Design*, CAD-6(2):270–281, March 1987.

[73] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[74] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.

[75] B. Hoppe, G. Neuendorf, D. Schmitt-Landsiedel, and W. Specks. Optimisation of high-speed CMOS logic circuits with analytical models for signal delay, chip area and dynamic power dissipation. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 9(3):236–247, March 1990.

[76] Paul Horowitz and Winfield Hill. *The Art of Electronics*. Cambridge University Press, second edition, 1989.

[77] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964.

[78] O. A. Izosimov, I. I. Shagurin, and V. V. Tsylyov. Physical approach to CMOS module self-timing. *Electronics Letters*, 26(22):1835–1836, October 1990.

[79] Gordon M. Jacobs and Robert W. Brodersen. A fully asynchronous digital signal processor using self-timed circuits. *IEEE Journal of Solid-State Circuits*, 25(6):1526–1537, December 1990.

[80] Mark B. Josephs and Jelio T. Yantchev. CMOS design of the tree arbiter element. *IEEE Transactions on VLSI Systems*, 4(4):472–476, December 1996.

[81] Y.-H. Jun, K. Jun, and S.-B. Park. An accurate and efficient delay time modeling for MOS logic circuits using polynomial approximation. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 8(9):1027–1032, September 1989.

[82] Y. Kameda, S. Polonsky, M. Maezawa, and T. Nanya. Primitive-level pipelining method on delay-insensitive model for RSFQ pulse-driven logic. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 262–273, 1998.

[83] Vitit Kantabutra and Andreas G. Andreou. A state assignment approach to asynchronous CMOS circuit design. *IEEE Transactions on Computers*, 43(4):460–469, April 1994.

[84] David Kearney and Neil W. Bermann. Performance evaluation of asynchronous logic pipelines with data dependant processing delays. In *Asynchronous Design Methodologies*, pages 4–13. IEEE Computer Society Press, May 1995.

[85] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, January 1974.

[86] Joep Kessels. VLSI programming of a low-power asynchronous Reed-Solomon decoder for the DCC player. In *Asynchronous Design Methodologies*, pages 44–52. IEEE Computer Society Press, May 1995.

[87] D. J. Kinniment. An evaluation of asynchronous addition. *IEEE Transactions on VLSI Systems*, 4(1):137–140, March 1996.

[88] Michael Kishinevsky, Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Synthesis of general Petri-nets. Technical Report TR96-2-004, University of Aizu, Japan, November 1996.
ftp://ftp.u-aizu.ac.jp/u-aizu/async/TR96-2-004.ps.gz.

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

[90] Michael Kishinevsky and Jørgen Staunstrup. Checking speed-independence of high-level designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 44–53, November 1994.

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

[92] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Taubin. The use of Petri nets for the design and verification of asynchronous circuits and systems. *Journal of Circuits, Systems and Computers*, 8(1), 1998.
ftp://ftp.u-aizu.ac.jp/u-aizu/async/pn-review98.ps.gz.

[93] Alex Kondratyev, Michael Kishinevsky, Jordi Cortadella, Luciano Lavagno, and Alex Yakovlev. Technology mapping for speed-independent circuits: decomposition and resynthesis. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 240–253. IEEE Computer Society Press, April 1997.

[94] Alex Kondratyev, Michael Kishinevsky, Bill Lin, Peter Vanbekbergen, and Alex Yakovlev. Basic gate implementation of speed-independent circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 56–62, June 1994.
ftp://ftp.u-aizu.ac.jp/u-aizu/async/230.ps.Z.

[95] Alex Kondratyev, Michael Kishinevsky, Alexander Taubin, and Sergei Ten. Analysis of Petri nets by ordering relations in reduced unfoldings. Technical Report TR 95-2-003, University of Aizu, Japan, June 1995.
ftp://ftp.u-aizu.ac.jp/u-aizu/async/95-2-0003.ps.gz.

[96] Prabhakar Kudva and Venkatesh Akella. A technique for estimating power in self-timed asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 166–175, November 1994.

[97] Prabhakar Kudva, Ganesh Gopalakrishnan, and Hans Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[98] Prabhakar Kudva, Ganesh Gopalakrishnan, Hans Jacobson, and Steven M. Nowick. Synthesis of hazard-free customized CMOS complex-gate networks under multiple-input changes. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[99] J. G. Kuhl and S. M. Reddy. A multicode single transition-time state assignment for asynchronous sequential machines. *IEEE Transactions on Computers*, 27:927–934, October 1978.

[100] Masashi Kuwako and Takashi Nanya. Timing-reliability evaluation of asynchronous circuits based on different delay models. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 22–31, November 1994.

[101] L. Lavagno. The SIS benchmark STGs.
ftp://ftp.cs.man.ac.uk/pub/amulet/www_support/async.tar.gz.

[102] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vincentelli. A novel framework for solving the state assignment problem for event-based specifications. Technical Report UCB/ERL M92/19, University of California, Berkeley, 1992.
http://www-cad.eecs.berkeley.edu/˜luciano/publications/tr/UCB-ERL-92-19.ps.gz.

[103] Luciano Lavagno, Kurt Keutzer, and Alberto Sangiovanni-Vincentelli. Synthesis of verifiably hazard-free asynchronous control circuits. Technical Report UCB/ERL M90/99, University of California, Berkeley, 1990.
http://www-cad.eecs.berkeley.edu/˜luciano/publications/tr/UCB-ERL-90-99.ps.gz.

[104] Luciano Lavagno, Kurt Keutzer, and Alberto Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 302–308. IEEE Computer Society Press, 1991.

[105] Luciano Lavagno, Kurt Keutzer, and Alberto Sangiovanni-Vincentelli. Synthesis of hazard-free asynchronous circuits with bounded wire delays. *IEEE Transactions on Computer-Aided Design*, 14(1):61–86, January 1995.

[106] Trevor W. S. Lee, Mark R. Greenstreet, and Carl-Johan Seger. Automatic verification of asynchronous circuits. *IEEE Design & Test of Computers*, 12(1):24–31, Spring 1995.

[107] D. L. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical Report TR-15-89, Harvard University, Cambridge, Massachusetts, USA, 1989.

[108] M. Lewis, J. Garside, and L. Brackenbury. Latch controller operating mode in asynchronous circuits. In *4th UK Asynchronous Forum*, July 1998.

[109] Bill Lin and Srinivas Devadas. Synthesis of hazard-free multilevel logic under multi-input changes from binary decision diagrams. *IEEE Transactions on Computer-Aided Design*, 14(8):974–985, August 1995.

[110] C. N. Liu. A state variable assignment method for asynchronous sequential switching circuits. *Journal of the ACM*, 10:209–216, 1963.

[111] Loadable counter and interrupt controller, design problems presented at the 1996 ACiD workshop at Groningen.
http://www.cs.man.ac.uk/amulet/async/problems/twoprob.ps.

[112] R. Madhavan. *Quick Reference for Verilog HDL*. Automata Publishing Company, San Jose, CA 95129, 1993.

[113] G. K. Maki and J. H. Tracy. A state assignment procedure for asynchronous sequential circuits. *IEEE Transactions on Computers*, 20:666–668, June 1971.

[114] A. J. Martin. Synthesis of asynchronous VLSI circuits. In *Proc. VII Banff Workshop on Asynchronous Hardware Design*, 1993.

[115] A. J. Martin. Tomorrow's digital hardware will be asynchronous and verified. In *Proc. VII Banff Workshop on Asynchronous Hardware Design*, 1993.

[116] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 245–260. Computer Science Press, 1985.

[117] Alain J. Martin. The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *Lecture Notes in Computer Science*, pages 244–259. Springer-Verlag, 1989.

[118] Alain J. Martin. Formal program transformations for VLSI circuit synthesis. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, UT Year of Programming Series, pages 59–80. Addison-Wesley, 1989.

[119] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[120] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The first asynchronous microprocessor: the test results. *Computer Architecture News*, 17(4):95–110, June 1989.

[121] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[122] M. D. Matson and L. A. Glasser. Macromodelling and optimization of digital MOS circuits. *IEEE Transactions on Computer Aided Design*, CAD-5(4):659–678, October 1986.

[123] E. J. McCluskey, Jr. Minimisation of boolean functions. *Bell Systems Technical Journal*, 35(6):1417–1444, November 1956.

[124] R. E. Miller. *Sequential Circuits and Machines*, volume 2 of *Switching Theory*. John Wiley & Sons, 1965.

[125] T. Miyamoto and S. Kumagai. An efficient algorithm for deriving logic functions of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[126] Charles E. Molnar, Ian W. Jones, Bill Coates, and Jon Lexau. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 279–289. IEEE Computer Society Press, April 1997.

[127] Cho W. Moon, Paul R. Stephan, and Robert K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 322–325. IEEE Computer Society Press, November 1991.

[128] S. Moore, P. Robinson, and S. Wilcox. Rotary pipeline processors. *IEE Proceedings, Computers and Digital Techniques*, 143(5):259–265, September 1996.

[129] Simon W. Moore and Peter Robinson. Rapid prototyping of self-timed circuits. In *Proc. International Conf. Computer Design (ICCD)*, October 1998.

[130] Shannon V. Morton, Sam S. Appleton, and Michael J. Liebelt. An event controlled reconfigurable multi-chip FFT. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 144–153, November 1994.

[131] Shannon V. Morton, Sam S. Appleton, and Michael J. Liebelt. ECSTAC: A fast asynchronous microprocessor. In *Asynchronous Design Methodologies*, pages 180–189. IEEE Computer Society Press, May 1995.

[132] D. E. Muller. Theory of asynchronous circuits. Technical report, University of Illinois Digital Computer Laboratory, December 1955.

[133] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits I. Technical report, University of Illinois Digital Computer Laboratory, November 1956.

[134] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits II. Technical report, University of Illinois Digital Computer Laboratory, March 1957.

[135] T. Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[136] Chris J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Elec. Eng., Stanford University, October 1995.

[137] Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[138] L. W. Nagel. SPICE2: A computer program to simulate semiconductor circuits. Technical Report ERL-M520, UC-Berkeley, May 1975.

[139] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994.

[140] S. M. Nowick and B. Coates. Automated design of high-performance asynchronous state machines. In *Proc. VII Banff Workshop on Asynchronous Hardware Design*, 1993.

[141] Steven M. Nowick, Mark E. Dean, David L. Dill, and Mark Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. In *Proc. Hawaii International Conf. System Sciences*, volume I, pages 419–427. IEEE Computer Society Press, January 1993.

[142] R. Panwar and D. Rennels. Input ordering for low power in CMOS logic gates. *International Journal of Electronics*, 78(5):925–943, May 1995.

[143] E. Pastor, J. Cortadella, A. Kondratyev, and O. Roig. Cover approximations for the synthesis of speed-independent circuits. In *Proc. IFIP Workshop on Logic and Architecture Synthesis*, December 1995. ftp://ftp.ac.upc.es/pub/archives/Papers/PCKR95.ps.gz.

[144] Priyadarsan Patra and Donald Fussel. Efficient building blocks for delay insensitive circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 196–205, November 1994.

[145] Priyadarsan Patra, Donald S. Fussell, and Stanislav Polonsky. Delay insensitive logic for RSFQ superconductor technology. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 42–53. IEEE Computer Society Press, April 1997.

[146] M. A. Peña and J. Cortadella. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[147] S. L. Peyton Jones. A practical technique for designing asynchronous finite-state machines. Technical Report CSC 91/R2, Department of Computing Science, University of Glasgow, April 1991.

[148] R. Puri and J. Gu. An efficient algorithm to search for minimal closed covers in sequential machines. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 12(6):737–745, June 1993.

[149] Ruchir Puri and Jun Gu. Asynchronous circuit synthesis; persistency and complete state coding constraints in signal transition graphs. *Int. Journal Electronics*, 75(5):933–940, 1993.

[150] W. V. Quine. The problem of simplifying truth functions. *American Math. Monthly*, pages 521–531, Fall 1952.

[151] W. Reisig. *Petri Nets: an Introduction*. Springer-Verlag, Berlin, 1985.

[152] C. A. Rey and J. Vaucher. Self-synchronized asynchronous sequential machines. *IEEE Transactions on Computers*, 23(12):1306–1311, December 1974.

[153] J.-K. Rho, G. D. Hachtel, F. Somenzi, and R. M. Jacoby. Exact and heuristic algorithms for the minimisation of incompletely specified state machines. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 13(2):167–177, February 1994.

[154] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, September 1988.

[155] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.

[156] G. Ruan, J. Vlach, and J. A. Barry. Current-limited switch-level timing simulator for MOS logic networks. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 7(6):659–667, June 1988.

[157] G. V. Russo and G. Palamà. Minimization of incompletely specified sequential machines. *Digital Processes*, 6(2–3):199–206, Summer-Winter 1980.

[158] J. W. J. M. Rutten and M. R. C. M. Berkelaar. Improved state assignments for burst mode finite state machines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 228–239. IEEE Computer Society Press, April 1997.

[159] T. Sakurai and A. R. Newton. A simple mosfet model for circuit analysis and its application to CMOS gate delay analysis and series-connected MOSFET structure. Technical Report ERL Memo No. ERL M90/19, Electronics Research Laboratory, University of California, Berkeley, March 1990.

[160] Charles L. Seitz. Ideas about arbiters. *Lambda*, 1(1, First Quarter):10–14, 1980.

[161] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[162] Alex Semenov, Alexandre Yakovlev, Enric Pastor, Marco Pe na, Jordi Cortadella, and Luciano Lavagno. Partial order based approach to synthesis of speed-independent circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 254–265. IEEE Computer Society Press, April 1997.

[163] Alexei Semenov and Alex Yakovlev. Verification of asynchronous circuits using time Petri-net unfolding. In *Proc. ACM/IEEE Design Automation Conference*, pages 59–63, 1996.

[164] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis*. Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992. Memorandum no. UCB/ERL M92/41.

[165] K. L. Shepard and V. Narayanan. Conquering noise in deep-submicron digital ICs. *IEEE Design and Test of Computers*, 15(1):51–62, January–March 1998.

[166] Y.-H. Shih, Y. Leblebici, and S.-M. Kang. ILLIADS: a fast timing and reliability simulator for digital MOS circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 12(9):1387–1402, September 1993.

[167] R. Smith, K. Fant, D. Parker, R. Stephani, and C. Y. Wang. An asynchronous 2-D discrete cosine transform chip. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 224–233, 1998.

[168] R. J. Smith. Generation of internal state assignments for large asynchronous sequential machines. *IEEE Transactions on Computers*, 23:924–932, September 1974.

[169] Robert F. Sproull and Ivan E. Sutherland. *Asynchronous Systems*. Sutherland, Sproull and Associates, Palo Alto, 1986. Vol. I: Introduction, Vol. II: Logical effort and asynchronous modules, Vol. III: Case studies.

[170] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, Fall 1994.

[171] I. Sutherland, R. Sproull, D. Roberts, C. Molnar, I. Jones, B. Coates, R. Yung, and J. Lexau. The counterflow pipeline processor project. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1994. Special invited session.

[172] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[173] C. J. Tan. State assignments for asynchronous sequential machines. *IEEE Transactions on Computers*, 20(4):382–391, April 1971.

[174] M. Theobald and S. M. Nowick. Fast heuristic can exact algorithms for two-level hazard-free logic minimization. Technical Report CUCS-001-98, Dept. of Computer Science, Columbia University, 1998. http://www.cs.columbia.edu/˜library/1998.html.

[175] José A. Tierno, Alain J. Martin, Drazen Borkovic, and Tak Kwan Lee. A 100-MIPS GaAs asynchronous microprocessor. *IEEE Design & Test of Computers*, 11(2):43–49, 1994.

[176] J. H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15:551–560, August 1966.

[177] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[178] Stephen H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, 20(12):1437–1444, December 1971.

[179] Stephen H. Unger. Self-synchronizing circuits and nonfundamental mode operation. *IEEE Transactions on Computers*, 26(3):278–281, March 1977.

[180] Stephen H. Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, June 1995.

[181] K. Usami, M. Igarashi, F. Minami, T. Ishikawa, M. Ichid, and K. Nogami. Automated low-power technique exploiting multiple supply voltages applied to a media processor. *IEEE Journal of Solid-State Circuits*, 33(3):463+, March 1998.

[182] P. Vanbekbergen, B. Lin, G. Goossens, and H. de Man. A generalized state assignment theory for transformations on signal transition graphs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 112–117. IEEE Computer Society Press, November 1992.

[183] Peter Vanbekbergen, Gert Goossens, Francky Catthoor, and Hugo J. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic

specifications. *IEEE Transactions on Computer-Aided Design*, 11(11):1426–1438, November 1992.

[184] Victor I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[185] S. R. Vemuru and E. D. Smith. Accurate delay estimation model for lumped CMOS logic gates. *IEEE Proceedings - G, Electronic Circuits and Systems*, 138(5):627–628, October 1991.

[186] Eric Verlind, Gjalt de Jong, and Bill Lin. Efficient partial enumeration for timing analysis of asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[187] W. T. Weeks, A. J. Jiminez, G. W. Mahoney, D. Mehta, H. Qasemzadah, and T. R. Scott. Algorithms for ASTAP – a network analysis program. *IEEE Transactions on Circuit Theory*, CT-20(6):628–634, November 1973.

[188] U. Weiser. Future directions in microprocessor design. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996. Invited lecture.

[189] N. Weste and K. Eshragian. *Principles of CMOS VLSI Design, A Systems Perspective*. Addison-Wesley, 1988.

[190] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.

[191] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 104–111. IEEE Computer Society Press, November 1992. http://www-cad.eecs.berkeley.edu/˜luciano/publications/tr/UCB-ERL-92-78.ps.gz.

[192] A. V. Yakovlev, M. Kishinevsky, A. Kondratyev, and L. Lavagno. On the models for asynchronous circuit behaviour with OR casality. Technical Report TR #463, University of Newcastle upon Tyne, November 1993.

[193] A. V. Yakovlev and A. I. Petrov. Symbolic signal transition graphs and asynchronous design. Technical Report TR #395, University of Newcastle upon Tyne, September 1993.

[194] Alexandre Yakovlev, Alexei Petrov, and Luciano Lavagno. A low latency asynchronous arbitration circuit. *IEEE Transactions on VLSI Systems*, 2(3):372–377, September 1994.

[195] Alexandre V. Yakovlev. On limitations and extensions of STG model for designing asynchronous control circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 396–400. IEEE Computer Society Press, October 1992.

[196] H. G. Yang and D. M. Holburn. Switch-level timing verification for CMOS circuits: a semianalytic approach. *IEEE Proceedings - G, Electronic Circuits and Systems*, 137(6):405–412, December 1990.

[197] C. Ykman-Couvreur, P. Vanbekbergen, and B. Lin. Concurrency reduction transformations on state graphs for asynchronous circuit synthesis. In *Proc. Int'l Workshop on Logic Synthesis*, May 1993. (In the ASSASSIN `ftp` distribution).

[198] Chantal Ykman-Couvreur and Bill Lin. Optimised state assignment for asynchronous circuit synthesis. In *Asynchronous Design Methodologies*, pages 118–127. IEEE Computer Society Press, May 1995.

[199] Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.

[200] J. Yuan and C. Svensson. High-speed CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, 24(1):62–70, February 1989.

[201] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[202] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 346–350. IEEE Computer Society Press, October 1992.

[203] Kenneth Yi Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems*. PhD thesis, Stanford University, August 1994.

Index