School of Electrical, Electronic & Computer Engineering

UNIVERSITY OF
NEWCASTLE UPON TYNE

Asynchronous Communication Circuits: Design, Test and Synthesis

Delong Shang

Technical Report Series

NCL-EECE-MSD-TR-2003-100

April 2003

Contact:
Delong.Shang@ncl.ac.uk

NCL-EECE-MSD-TR-2003-100
Copyright © 2003 University of Newcastle upon Tyne

School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

University of Newcastle upon Tyne

School of Computing Science
School of Electrical, Electronic and Computer Engineering

Asynchronous Communication Circuits: Design, Test, and Synthesis

A thesis submitted in partial fulfilment
of the requirements for the degree of

Doctor of Philosophy

Delong Shang

March 2003

NCL-EECE-MSD-TR-2003-100

Contents

List of Figures

List of Tables

Acknowledgements

This thesis would not have been possible without the generous and invaluable help I received from many people and organisations during the course of study.

Firstly, thanks to Alex Yakovlev and Albert Koelmans, my supervisors, for introducing me in the world of asynchronous circuits. Their enthusiasm in the asynchronous research field is the main motivation that has encouraged this work since the early beginning. Alex and Albert have patiently taught me during these years everything I know about research. They deserve my sincere gratitude.

Also my thanks go to Fei Xia. Words cannot express my gratitude to my friend and colleague Fei Xia, whose endless enthusiasm and brilliant mind benefited much of the work reported here. In addition, many thanks go to him for his careful reading of the draft of this thesis and his constructive suggestions for improvement.

I also address a big thanks to Alex Bystrov and Frank Burns for many useful discussions related to this thesis and their valuable help in English language in these years.

The same thanks go to all other colleagues of the VLSI group in the Department of Computing Science and my officemates for their enthusiasm and encouragement during these years.

I appreciate help and useful feedback received from the members of the COMFORT project, as well as members of the UK and international community working in asynchronous designs.

I am also grateful to the UK asynchronous Forum, AINT 2000 international workshop, ASYNC 2000 and ASYNC 2001 international conferences, DDECS 2001 *international workshop, ISCAS 2002 international conference and so on, which give me chances to present my work.*

Here, I would also like to thank the EPSRC and the Department of Computing Science in the University of Newcastle upon Tyne for vital financial support which kept me alive during the many years of research work, and the Shenyang Institute of Computing Technology, the Chinese Academy of Sciences, for letting me have a chance to study for this degree.

Finally, thanks to my parents Shiyuan and Yelan for their love and support. My special gratitude and love goes to my wife Junying and my son Ersong. They sometimes waited patiently at home and sometimes joined in the fun. They have been the indispensable persons to complete this research. Their combined efforts have made possible these lines. The work is dedicated to them.

September 2002 *Newcastle upon Tyne*

Abstract

This thesis presents the design and testing of asynchronous communication mechanism (ACM) circuits, and the development of an asynchronous circuit synthesis method which not only supports the ACM work but also has much wider application potential.

ACMs are a unique approach to data transmission between subsystems not synchronized with one another. The successful systematic implementation of ACM hardware circuits presented here demonstrates the potential of ACM applications in hardware systems and establishes a number of techniques well suited for ACM hardware design and synthesis. Novel testing procedures are developed specially for ACM circuits, and testing carried out on fabricated ACM circuits complement knowledge on the ACM implementations gained from analyses and simulations.

The asynchronous circuit synthesis method proposed in this thesis and its useful library proved to be very helpful in bringing an element of automation to the design and implementation process. Not limited to ACM circuits, this method can be further developed to help designers of general asynchronous circuits.

Chapter 1: Introduction

Asynchronous (or self-timed) circuits and systems have attracted increasing attention from the research community in recent years. The inherent concurrency in their operation and the absence of the requirement for a pre-determined settling period, the clock cycle, means that these systems reflect more naturally the processes happening in real life.

1.1 Motivation

1.1.1 General purpose motivation

Most digital circuits designed and fabricated today are "synchronous". In essence, they are based on two fundamental assumptions that greatly simplify their design:

1. all signals are binary and;
2. all components share a common and discrete notion of time, as defined by a clock signal distributed throughout the circuit.

By making the assumption of a synchronous mode of operation, designers can abstract from the problem of tracking of all intermediate states of the system. It can be safely assumed that the clock period is chosen to be long enough for the signals to settle to their new values. Any feedback is cut off to prevent the changing outputs from affecting the inputs. The arrival of a new clock pulse triggers a transition to the next state of the system.

Asynchronous circuits are fundamentally different. In an asynchronous circuit there is no such a “start-stop” mechanism. They also assume binary signals, but there is no common and discrete time. Any change of signals may cause a transition of the system into the next state. Instead the circuits use handshake protocols between their components in order to perform the necessary synchronization, communication and sequencing of operations. Expressed in ‘synchronous terms’ this results in a behaviour that is similar to systematic fine-grain clock gating and local clocks that are not in phase, and whose period is determined by actual circuit delays – registers are only clocked where and when needed [Sparsø 2001].

This difference gives asynchronous circuits inherent properties that can be (and have been) exploited [Amulet] in the following areas:

1. Low power consumption;

   During the operation of a synchronous circuit the clock signal is propagated to every operational block of the circuit even if this block is not used in a particular computation. Thus the power is spent on driving the clocked inputs of gates which do not perform any useful actions.

   Each part of an asynchronous circuit operates only when signalled to commence the operation, after the data has been prepared on the inputs of this part. Therefore, until such a request is produced, this part of the circuit does not consume any power at all (apart from very small leakage currents). In other words, asynchronous circuits work under fine grain clock gating and zero standby power consumption.

2. Average case performance;

   The clock cycle of every synchronous circuit is determined by the longest propagation delay of the circuit. The rate of the clock signal must accommodate the settling times for the longest possible operation. Therefore, during a faster operation some of the part of the circuit will stay idle while the clock signal is due to switch. To overcome this problem, designers need to come up with elaborate scheduling and re-timing schemes.

In an asynchronous circuit, every part works at its own pace. As soon as the data has been processed by one part, the next part is informed and may start working with the data. Thus the overall cycle time, i.e. the average time between the completions of two sequential operations, will be the average of the execution times of all operations.

3. Less emission of electro-magnetic noise;

   Electro-magnetic emission generated by synchronous circuits causes interference with other equipment. Much of this interference is attributed to the clock signal which produces a steady peak in the spectrum on the frequency at which the transistors are switched.

   The transistor switching frequency in an asynchronous circuit depends on the data which is being processed by the circuit. Thus the spectrum is smoother and the peak values are lower.

4. Robustness towards variations in supply voltage, temperature and fabrication process parameters;

   Timing is based on matched delays (and can even be insensitive to circuit and wire delays) [Sparsø 2001].

5. Better composability and modularity;

   Synchronous circuits are subject to precise synchronisation between the modules comprising them. Redesigning of any module requires meeting heavy restrictions on the execution times to ensure the correct synchronisation.

   Because of the simple handshake interfaces and the local timing, any part of an asynchronous circuit can be redesigned at will. The new module must, of course, conform to the same interface protocol as the module that is being replaced. However, the speed at which the new module operates is irrelevant allowing easy upgrading of asynchronous circuits.

6. No clock distribution and clock skew problems.

   The existence of a propagation delay in the wires of a chip means that the signal change may arrive at two ends of a forking wire at different times. This phenomenon is known as the clock skew problem [Berkel 1992]. To guarantee that all operational blocks work synchronously, the designer needs to make sure that the block signal is received by each block at exactly the same time. However, with growing clock rates it becomes increasingly difficult to guarantee the absence of clock skew. In addition, clock wiring has been reported to take up to 60% of all wiring in the chip.

   By choosing an asynchronous implementation the designer escapes the clock skew problem and the associated routing problem. That means there is no global signal that needs to be distributed with minimal phase skew across the circuit.

The above theoretical advantages have inspired a large number of researchers into the asynchronous circuit area. The asynchronous community has demonstrated that it is possible to design fully functional circuits beyond trivial examples. Several microprocessors have been designed to date. Examples of microprocessor designs can be found in works reported by the Caltech [Martin 1989a, Martin 1990c], Titech [Nayna 1995], Philips [Gageldonk 1998] and Manchester [Furber 1999] research groups. In Manchester, the AMULET group designed an instruction-level compatible asynchronous version of the ARM6 microprocessor whose performance characteristics are comparable to those of the synchronous one. In addition, Philips reported a design of an asynchronous error correction chip [Berkel 1994a, Berkel 1994b] which demonstrated 80% saving in the power consumption. Kol has claimed “Future processors will be asynchronous” [Kol 1997].

In addition, with the advent of sub-micro VLSI technology, which will soon enable a billion of transistors to be placed on a single chip, hardware design becomes a big challenge. Future VLSI circuits will often be system-on-chip, even multiple systems on a same chip, whose subsystems include processors, memory banks and input/output controllers. Problems with distributing the global clock between these subsystems, treated as Intellectual Property (IP) cores, are unavoidable. Systems-on-chip will effectively lose the global notion of physical time and permit actions in different parts of the systems to be executed in parallel or independently of one another. Such hardware systems will inevitably become more asynchronous and concurrent [ITRS 2001, Yakovlev 2000].

1.1.2 Special topics of concern in this thesis

1.1.2.1 Asynchronous communications

As mentioned above, many researchers focus on the asynchronous circuit design area. However, the area of communication between two globally unsynchronised processes (systems) has not been widely investigated and is worth being studied.

Simpson proposed a simple and elegant classification for this kind of communication, which is named asynchronous communication mechanisms (ACMs) [Simpson 1990, Simpson 1994]. The simplest model of Simpson's ACMs assumes that two independent processes, one the writer which supplies data and the other the reader which consumes the data, communicate with each other independently. Normally ACM has four types. They are *Channel*, *Pool*, *Signal* and *Constant*. The definitions and more details will be given in Chapter 2.

So far, the *Channel* type is very popular (this is also named as FIFO). It has been studied for long time and implemented in both software and hardware. Especially it has been implemented by using asynchronous circuits, such as [Sutherland 1989, Hoke 1999, Liljeberg 2001]. The important feature is that the above implementations use asynchronous circuits, which show a large number of benefits [Sutherland 1989]. As for the other types of ACMs, most research on implementations of these mechanisms is focused on using software, such as [Chen 1998b]. So far only

Simpson himself described a hardware implementation for the *Pool* type ACM. However this implementation is FM (Fundamental Mode) but not self-timed [Simpson 1994]. Some timing assumptions are used, and in order to guarantee that two processes -- the writer and reader -- run independently, four slot shared memory is used to pass data. From the hardware point of view, it is not safe. It should work under the worst case assumptions.

As an alternative to synchronous techniques, asynchronous techniques do not have the above problems. Because of this, asynchronous techniques should give us a better implementation.

The classification is based on the properties of the writer and reader, such as “destructive” and “non-destructive” on the writer and/or reader. However, the classification based on “destructive” and “non-destructive” properties has limitations. For example, the *Constant* type ACM has not any communications between the writer and reader.

All of the above are worth being studied.

1.1.2.2 Testing asynchronous circuits

Testing is a hot topic in hardware design. A number of testing techniques have been developed for synchronous circuits [Wang 1991], such as Design For Testability (DFT), Built-in Self Test (BST), Boundary Scan Techniques (BST) and so on. A summary of one technique, DFT, is introduced in [Williams 1982]. However, in asynchronous areas, designing asynchronous circuits has been challenging because hazards and races must be carefully considered. Therefore the focus of research in the area has been primarily directed to synthesis and verification techniques, while little attention has been paid to techniques that efficiently verify whether a fabricated asynchronous circuit has any physical faults.

As asynchronous circuits become larger and start to be used in commercial products, testing concerns become critical [Hulgaard 1994]. However, synchronous circuit testing techniques cannot be used on asynchronous circuits directly, especially self-timed circuits [Hulgaard 1994]. An example of asynchronous circuit testing using synchronous testing techniques can be found in [Kondratyev 2002].

Some techniques for testing asynchronous circuits have been introduced in [Hulgaard 1994] and [Petlin 1994]. These include DFT, Path Delay Fault Testing (PDFT), and self-timed circuit testing techniques. However they are not adequate for testing our self-timed ACM. Self-timed circuits are delay-independent. We cannot exactly know when an event happens and when the next event starts. This property is very difficult to test [Petlin 1994]. Our self-timed ACM is a fully asynchronous system. Apart from the self-timed implementation, it has three important properties: asynchrony, data coherence and data freshness. These properties will be introduced in Chapter 2 and 3. So far no testing techniques for this kind of circuit have been developed.

With the advent of sub-micron VLSI technology, multiple independent systems in one chip should be possible. Communication between them cannot be avoided. Obviously this topic is worth being investigated.

1.1.3 Automated asynchronous circuit design (synthesis)

Although asynchronous circuits have a large number of advantages, and many researchers are focussing on this area, asynchronous circuits are harder to design. This is because mature asynchronous CAD tools are not available [Sutherland 2002]. Although some asynchronous CAD tools such as the Petrify tool [Petrify] and the Tangram tool [Kessels 2001] have been presented, they have some limitations. One important limitation is that they cannot deal with big designs. We will discuss this in more detail in Chapter 5.

From our requirement point of view, the other main problem is that existing tools cannot guarantee the obtained circuits are speed independent (SI). More details will be discussed in Chapter 5.

As a result, in order to design asynchronous products and meet time-to-market demands, much more research in this area is needed.

1.2 Contributions of this thesis

1.2.1 Self-timed ACMs

The main objectives of this work are to investigate design techniques for implementing communications between multiple independent systems (non-synchronized), using self-timed circuits.

The work in this thesis is motivated by Simpson's work [Simpson 1990, Simpson 1994]. As introduced in the motivation section, the work is focused on ACM mechanisms and self-timed implementations. In addition, a new classification (*Channel*, *Pool*, *Signal* and *Message*) is presented which is based on timing properties, blocking and waiting. More details will be shown in Chapter 3.

As discussed above, in this thesis we focus on *Pool*, *Signal* and *Message* type ACM mechanisms and self-timed implementations. In addition, the ACM is a slot mechanism. We wish to use as few as possible slots to implement ACMs, to reduce hardware size.

1.2.2 Testing ACMs

As mentioned in the above section, testing is very important. Generally, only after testing can a circuit be expected to be correct.

After implementing and fabricating some of the ACMs using self-timed circuits, we have the chance to study the asynchronous testing problem. The mechanism is a special one. It has two global non-synchronized processes. As mentioned in the above motivation section, it is very difficult to test this kind of circuit. No methods are available. In this thesis, a method is proposed for testing such circuits.

1.2.3 Synthesis

After implementing self-timed ACMs, we appreciate the difficulty of designing asynchronous circuit systems. One reason is the lack of mature asynchronous CAD tool support [Sutherland 2002]. We wish to develop an asynchronous methodology to support design of self-timed circuits.

The idea is based upon Varshavsky's direct translation method [Varshavsky 1996]. However, Varshavsky's method is not automated and thus not good enough to support asynchronous designs. The reason is discussed in Chapter 5. In this thesis, we will extend the method and are trying to automate it.

Based on this new synthesis method, some optimization methods are also proposed to improve the performance of asynchronous circuits.

Summary

In summary, the main contributions of the work described in this thesis are:

1. Study ACMs and implement them using self-timed circuits;
2. Test self-timed ACMs;
3. Extend Varshavsky's direct translation method and automate it.

1.3 Organization of this thesis

The remainder of the thesis is organised as follows:

Chapter 2 presents the background of my contributions. This consists of the fundamental of asynchronous VLSI design, Simpson's asynchronous communication mechanisms and related work, the current situation of asynchronous CAD tools.

Finally, in order to develop CAD tools, a useful specification language, Petri nets/STG is introduced.

Chapter 3 presents a new classification of asynchronous communication mechanisms. In addition, four-slot and three-slot *Pool* type of mechanism are studied and implemented using self-timed circuits. Furthermore, a two-slot *Signal* type of mechanism is investigated and implemented using self-timed circuits.

Chapter 4 presents a testing method for asynchronous communication mechanisms. We use the four-slot *Pool* as an example to demonstrate the testing method. Testing results show the method is as expected.

Chapter 5 presents a purpose of an asynchronous synthesis tool which is used to directly translate Petri nets specifications to logic circuits that are guaranteed speed independent.

Chapter 6 presents some case studies based on the synthesis method introduced in Chapter 5. In this chapter we illustrate how the application of the direct translation may assist in implementing speed independent circuits.

Chapter 7 concludes the thesis, summarises the results presented in this work and outlines the areas for future research.

1.4 Publications on the thesis

The following papers, based on the work presented in this thesis, have been published or submitted for publication:

- An Asynchronous Communication Mechanism using self-timed circuits [Xia 1999a]

  (6th UK Asynchronous Forum);

- Self-timed and speed independent latch circuits [Bystrov 1999]

  (6th UK Asynchronous Forum);

- Asynchronous Communication Mechanisms Using Self-timed Circuits [Xia 2000b]

  (ASYNC 2000 Conference);

- A self-timed asynchronous data communication mechanism [Shang 2000a]

  (1st PGNET 2000 Conference);

- An implementation of a three-slot asynchronous communication mechanism using self-timed circuits [Shang 2000b]

  (AINT 2000 Workshop);

- Testing a self-timed asynchronous communication mechanism (ACM) VLSI chip [Shang 2000c]

  (9th Asynchronous UK Forum);

- Testing a self-timed asynchronous communication mechanism (ACM) VLSI chip [Shang 2001a]

  (DDECS 2001 Workshop);

- Synthesis and implementation of a *Signal*-type asynchronous data communication mechanism [Yakovlev 2001]

  (ASYNC 2001 Conference);

- Asynchronous Circuit Synthesis via Direct Translation [Shang 2001b]

  (11th UK Asynchronous Forum);

- Asynchronous Circuit Synthesis via Direct Translation [Shang 2002a]

  (ISCAS 2002 Conference);

- Behavioural synthesis of asynchronous controllers: a case study with a self-timed communication channel [Yakovlev 2002a]

(ACiD-WG 2002 Workshop);

- Asynchronous communication mechanisms: classification and hardware implementations [Xia 2002]

  (MPCS 2002 Conference);

- Data communication in system with heterogeneous timing [Xia 2002b]

  (IEEE Micro journal).

Chapter 2: Background

2.1 Asynchronous circuit design

Computer pioneers designed digital computers based on thermionic valves [Williams 1948, Williams 1951] fifty years ago. Computer systems such as the Ferranti Mark I [Ferranti 1952] were constructed based on digital signals, which have the following properties:

- Two values, 0 and 1, represent information and they are represented as distinct signal values (most commonly voltages);
- Signals must only be sampled or observed when in one of these two distinct states.

The digital signal systems have three significant advantages in the design of digital equipment [Hayes 1993]:

1. Most information-processing systems are constructed from switches, which are binary devices.
2. The basic decision-making processes required of digital systems are binary.
3. Binary signals are more reliable than those formed by more than two quantization levels.

Until now, the principles of binary digital design are the same as they are before.

Generally, circuit design styles can be classified into two major categories, namely synchronous and asynchronous. They are informally defined as follows:

- The synchronous (clocked) circuit is one with a global timing signal (clock signal) which is distributed to all parts of the circuit. Transitions (rising and/or falling depending on the design) on this clock line indicate moments at which the data signals become stable;
- The asynchronous (self-timed) circuit is one which utilises time delays, as indicated by local matched delay lines, to indicate when the data signals are stable, or encodes the timing information in the data line activity itself.

In the early days of digital circuit design, little distinction was made between synchronous and asynchronous circuits. However, since the 1960's, the mainstream of the digital circuit design enterprise has been primarily concerned with synchronous circuits [Davis 1997].

These kinds of circuits make use of centralized control and are based on two major assumptions: (1) all signals are binary, and (2) time is discrete. By assuming binary values on signals, simple Boolean logic can be used to describe and manipulate logic constructs. By assuming time is discrete, hazards and feedback can largely be ignored. So synchronous circuits are easy to understand and design, even with increasing complexity of design. As a result, most modern digital systems are synchronous. They are organized around a global clock, and system events are synchronized to the clock. On each clock tick, data is latched into storage elements and then a new computation begins between two clock ticks. Computation must be completed before the next clock tick [Hauck 1995, Nowick 1993].

However, as devices become smaller and faster, especially when the SoC (systems on a chip) era arrives and hardware systems become much more complex and concurrent, synchronous approaches will become increasingly unwieldy.

There are a number of difficulties with synchronous design as follows [Nowick 1993, ITRS 2001]:

- **Clock skew**: in a synchronous system, if the clock is not distributed evenly, clock skew results and the system may malfunction. Clock skew is an inherent problem in most synchronous systems. However, in practice, the effects of clock skew can be eliminated in two ways. First, the clock can be slowed down to ensure correct operation. That is, a safety margin is added to each clock cycle to ensure that the clock has been broadcast throughout the system and all components are stable before a new cycle begins. However, the cost of this approach is a performance loss. Alternatively, clock skew can be minimized by using carefully balanced clock trees. The cost of this approach is an increase in system area.
- **Asynchronous external inputs**: in a synchronous system, there is a reliability problem when attempting to synchronize inputs which can arrive at arbitrary times. Such inputs may cause synchronous storage elements to enter into undefined states. This problem is called metastability [Chaney 1973]. No known method can eliminate metastability. However, the probability of entering a metastable state is significantly reduced by using a pair of storage elements to "resynchronize" an asynchronous input to the clock [McCluskey 1986]. However, such resynchronization results in a performance loss.
- **Worst-case design**: synchronous designs have difficulty taking advantage of data dependent processing delays. If a component can process particular inputs or data quickly, its performance is still bound by the global clock speed. In fact, the speed of the clock is usually set assuming worst-case conditions for process, temperature, voltage and data. As a result, even when the system operates under nominal conditions, performance is limited by worst-case design assumptions. In practice, the cumulative "derating" of system performance based on these factors can be significant [Dean 1992, Williams 1991]. Dean [Dean 1992] indicates that, if such design-for-worst-case could be avoided, many systems would actually run almost twice as fast on average.

- **Power consumption**: at a time when designers are increasingly interested in low power applications, the distribution of the clock throughout the system is a large source of power consumption. The problem of power consumption will only grow worse as clock frequency increases and feature size decreases.
- **Modularity (reuse)**: in a synchronous system, a component cannot be replaced without global implications. If the new component is slow, the system may malfunction unless the global clock speed is reduced. If the new component is fast, system performance will not change unless the clock speed can safely be increased. The contrast to modern object-oriented software systems is illuminating. In an object-oriented system, a software module can be replaced without global implications. Such modularity increases the lifetime of a system, allows rapid development, and simplifies system organization. Modularity is an important feature in system design; however it does not fit well with a synchronous paradigm.
- **Composability**: finally, at a time when designers are interested in constructing large multi-chip systems, synchronous designs have limited composability. It is difficult to combine synchronous subsystems operating at different clock speeds.

To solve the above problems, an alternative approach is to build asynchronous systems. Dean has proved that systems operated without the above assumptions have the potential to generate better results [Dean 1992].

Asynchronous circuits have been studied in one form or another since the early 1950’s [Keister 1951] when the focus was primarily on mechanical relay circuits. A number of theoretical issues were studied in detail by Muller and Bartky as early as 1956 [Muller 1956].

Asynchronous circuits are systems which do not have a global clock; instead, they operate under distributed control. The key is that asynchronous systems avoid many of the above problems by eliminating the global clock (see Chapter 1).

Over the past two decades, research into asynchronous design has concentrated on finding more disciplined approaches which can challenge clock-based design in offering a reliable basis for VLSI design. Often unbounded delay assumptions are used, which guarantee that a circuit will always operate correctly under any distribution of delay amongst the gates and wires within the circuit [Bainbridge 2000].

Currently, asynchronous design methods are well developed, and whole computer systems can be constructed as a single integrated circuit using either synchronous or asynchronous methods.

However, in spite of much research over the last 20 years, asynchronous designs are notoriously difficult to build [Davis 1997, Hauck 1995]. Fundamental to an understanding of asynchronous design is a familiarity with the assumptions commonly made regarding the delays in the gates and wires within a circuit and the mode in which the circuit operates. There are two common delay models, bounded delay and unbounded delay. The bounded delay model was commonly used in the early days of asynchronous design, and is still used in some backplane level interconnection schemes such as the SCSI bus where part of the protocol is based on known, fixed delays. Current asynchronous VLSI designs and research efforts use the unbounded delay model for the implementation of state machines and controllers since it leads to circuits that will always operate correctly whatever the distribution of delays. It separates delay management from the correctness issue, allowing the functionality of the circuit to be more easily verified. The bounded delay model is still commonly used for datapath components, however, since in this area it can lead to more efficient implementations.

The following sub-sections discuss other aspects of asynchronous circuits.

2.1.1 Delay models and hazards

2.1.1.1 Delay models, circuits and the environment

There is a wide spectrum of asynchronous designs. One way to distinguish among them is to understand the different underlying models of delay and operation. Every physical circuit has an inherent delay. However, since synchronous circuits process inputs between fixed clock ticks, they can often be regarded as instantaneous operations, computing a new result in each clock cycle. On the other hand, since asynchronous circuits have no clock, they are best regarded as computing dynamically through time. Therefore, a delay model is critical in defining the dynamic behaviour of an asynchronous circuit.

There are two fundamental models of delay: (1) the pure delay model and (2) the inertial delay model [Unger 1969]. A pure delay can delay the propagation of a waveform, but does not otherwise alter it. An inertial delay can alter the shape of a waveform by attenuating short glitches. More formally, an inertial delay has a threshold period, say δ. Pulses of duration which are less than δ are filtered out.

Delays are also characterized by their timing models. In a fixed delay model, a delay is assumed to have a fixed value. In a bounded delay model, a delay may have any value in a given time interval. In an unbounded delay model, a delay may take on any finite value.

An entire circuit's behaviour can be modelled on the basis of its component behaviour. In a simple gate, or gate-level, model, each gate and primitive component in the circuit has a corresponding delay. In a complex-gate model, an entire sub-network of gates is modelled by a single delay; that is, the network is assumed to behave as a single operator, with no internal delays. Wires between gates are also modelled by delays. A circuit model is thus defined in terms of the delay models for the individual wires and components. Typically, the functionality of a gate is modelled by an operator with an attached delay.

Given a circuit model, it is also important to characterize the interaction of the circuit with its environment. The circuit and environment together form a closed system, called a complete circuit [Miller 1965]. If the environment is allowed to respond to a circuit's outputs without any timing constraints, the two interact in input/output mode [Brzozowski 1989]. Otherwise, environment timing constraints are assumed. The most common example is fundamental mode [Unger 1969] where the environment must wait for a circuit to stabilize before responding to circuit outputs. Such a requirement can be seen as the hold time for a simple latch or flip-flop [Davis 1998].

2.1.1.2 Hazards

A hazard is an unspecified change of the signal, e.g. a spike. It can be classified into two types.

2.1.1.2.1 Function hazards

A function f which does not change monotonically during an input transition is said to have a function hazard in the transition. The following definitions are from [Bredeson 1972].

Definition 2.1 A Boolean function f contains a static function hazard for the input transition from A to C iff:

1. $f(A) = f(C)$, and

2. there exist some input states $B \in [A, C]$ such that $f(A) \neq f(B)$.

Definition 2.2 A Boolean function f contains a dynamic function hazard for the input transition from A to D iff:

1. $f(A) \neq f(D)$,

2. there exist a pair of input states B and C ($A \neq B$, $C \neq D$) such that

(a) $B \in [A, D]$ and $C \in [B, D]$ and

(b) $f(B) = f(D)$ and $f(A) = f(C)$.

It is well known that, if a transition has a function hazard, no implementation of the function is guaranteed to avoid glitches during the transition, assuming arbitrary gate and wire delays [Eichelberger 1965]. Therefore, in the remainder of this thesis, transitions are assumed to be free of function hazards except where otherwise indicated.

2.1.1.2.2 Logic hazards

If *f* is free of function hazards for a transition from input state A to B, it may still have hazards due to delays in the actual logic realization [Unger 1969].

Definition 2.3 A combinational circuit for a function *f* contains a static logic hazard for the input transition from min-term *A* to min-term *B* iff:

1. $f(A) = f(B)$,

2. no static function hazard exists in the transition from *A* to *B*,

3. for some delay assignment, the circuit's output is not monotonic during the transition interval.

Definition 2.4 A combinational circuit for a function *f* contains a dynamic logic hazard for the input transition from mini-term *A* to mini-term *B* iff:

1. $f(A) \neq f(B)$,

2. no dynamic function hazard exists in the transition from *A* to *B*,

3. for some delay assignment, the circuit's output is not monotonic during the transition interval.

These definitions formalize the notion that a logic hazard occurs if, for some particular gate and wire delays, the combinational circuit output glitches during the transition (a pure delay model is of course assumed).

A fundamental difference between synchronous and asynchronous circuits is in their treatment of hazards. In a synchronous system, computation occurs between clock ticks. Glitches on wires during a clock cycle are usually not a problem. The system operates correctly as long as a stable and valid result is produced before the next clock tick, when the result is sampled. In contrast, in an asynchronous system, there is no global clock; computation is no longer sampled at discrete intervals. As a result, any glitch may be treated by the system as a real change in value, and may cause the system to malfunction [Davis 1997].

Hazards were first studied in the context of asynchronous state machines, and much of the original work focused on combinational logic. Sequential hazards are also possible in asynchronous state machines; these are called critical races or essential hazards.

In an asynchronous circuit design, two traditional classes of combinational hazards are SIC (single input change) and MIC (multiple input change) hazards.

The original theory of SIC was developed by Huffman, Unger and McClusky [Unger 1969]. Generally for a SIC, a static hazard can be eliminated by inserting redundant items. In terms of function, it is less efficient, but is necessary to eliminate the hazard.

The case of a MIC is much more complex. A MIC transition has a start input value M and a destination input value N where several inputs change monotonically between M and N. Multiple input change can lead to function-hazards, static hazards and dynamic hazards.

The static hazards and the dynamic hazards are informally defined as follows:

Definition 2.5 A circuit is said to be a static hazard at an output for the input transition form *A* to *B* iff:

1. $f(A) = f(B)$,

2. there exists a temporary output value at the output when the inputs changing from *A* to *B*, say $V_{A \to B}$, and $V_{A \to B} \neq f(A)$.

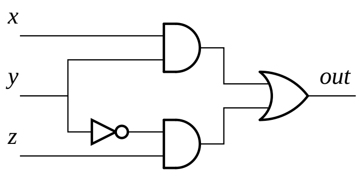


Figure 2.1 A circuit with hazards.

Figure 2.1 is used to show a static hazard. The circuit is assumed that all gates have a gate delay of 1 unit, and the current state is (x, y, z) = (1, 1, 1). In this state, the output is 1. If we change the inputs from (1, 1, 1) to (1, 0, 1), the output should still remain at 1. However, because of the delay in the inverter, the top AND gate will become false before the lower AND becomes true, and a 0 will propagate to the output. This momentary glitch on the output is known as a static-1 hazard. A static-0 hazard is similar, with a value meant to remain stable at 0 instead momentarily becoming 1.

Definition 2.6 A dynamic hazard is the case where a signal that is meant to make a single transition ($0 \rightarrow 1$ or $1 \rightarrow 0$) instead makes three or more transitions (such as $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$, $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$).

Generally, function hazards cannot be avoided. Therefore, classic synthesis methods focus only on MIC transitions which are already function hazard free. Static hazards can be eliminated by using the same methods as those used in a SIC. A more difficult problem is to eliminate MIC dynamic logic hazards.

2.1.2 Circuit classification

Asynchronous circuits can most easily be categorized by the timing models they assume. The followings will introduce some most popular types of asynchronous circuits.

2.1.2.1 Fundamental mode circuits (Huffman circuits)

The most obvious model to use for asynchronous circuits is the same model used for synchronous combinational circuits which are generally referred to as Huffman circuits. Specifically, in this kind of circuit it is assumed that the delay in all circuit elements and wires is known, or at least bounded. Circuits designed with this model usually are associated with the fundamental mode (FM) assumption.

However, since there is no clock to synchronize input arrivals, the system must behave properly in any intermediate states caused by multiple input changes. In terms of this point, asynchronous sequential circuits are different to synchronous ones.

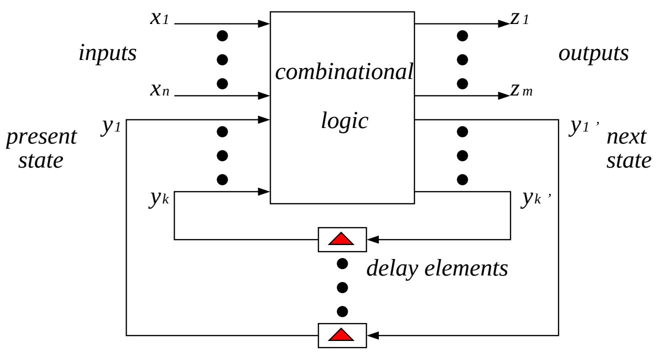


Figure 2.2 Huffman sequential circuit structure.

In asynchronous sequential circuits, we use a model similar to that used for synchronous circuits as shown in Figure 2.2. Since the restriction that only one input to the combinational logic can change at a time is made, this forces several requirements on our sequential circuit. First, we must make sure that the combinational logic settles in response to a new input before the present-state entries change. This is done by placing delay elements on the feedback lines as shown in Figure 2.2.

2.1.2.2 Non FM circuits (extended Huffman circuits)

The FM assumption, while making logic design easy, greatly increases cycle time. Therefore there could be considerable gains from removing this restriction.

One method is quite simple, and can be seen by referring back to the original argument for the FM. The issue was that when multiple inputs change, and no single cube covers the starting and ending point of a transition, there is the possibility of a hazard. However, if a single cube covers an entire transition, then there is no need for the FM restriction, since that cube will ensure the output stays at 1 at all times. However, obviously this method cannot completely eliminate the FM assumption [Hauck 1995].

Another method, proposed by Hollaar [Hollaar 1982], uses detailed knowledge of the implementation strategy to allow new transitions to arrive earlier than FM assumption allows. Essentially, Hollaar builds a 1-hot (introduced below) encoded asynchronous state machine with set-reset flip-flop for each state bit. The set input is driven when the previous state's bit and the transition function are true. This basic scheme is expanded beyond simple straight-line state machines, and allows parallel execution in asynchronous state machines.

2.1.2.3 Burst-mode (BM) circuits

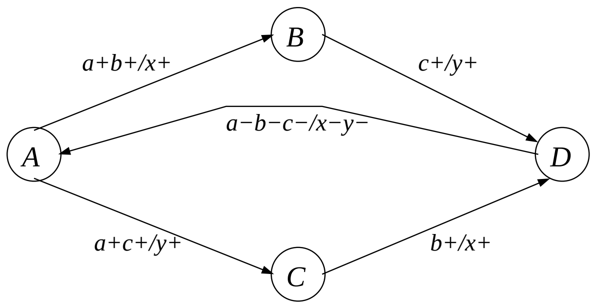


Figure 2.3 Burst-mode specification.

Circuits designed with a bounded delay model do not necessarily have to use the structures described previously. A different design methodology, referred to as burst mode (BM), attempts to move even closer to synchronous design styles than Huffman circuits. The BM design style was developed by Nowick, Yun and Dill [Nowick 1991, Yun 1992a, and Yun 1992b] based on earlier work at HP laboratories by Davis, Stevens and Coates [Davis 1993]. As shown in Figure 2.3, circuits are specified via a standard state machine, where each arc is labelled by a non-empty set of inputs (an input burst) and a set of outputs (an output burst). Similar to its use in synchronous circuits, the assumption is that when in a given state, only the inputs specified on one of the input bursts leaving this state can occur. These are allowed to occur in any order, and the machine does not react until the entire input burst has occurred. The machine then fires the specified output burst, and enters the specified next state. New inputs are allowed only after the system has completely reacted to the previous input burst. Thus, BM systems still require the FM assumption. Also, no input burst can be a subset of another input burst leaving the same state.

The above three kinds of circuits are all based on a timing assumption that is the delay on all elements and wires in a system is bounded. Although the bounded delay design methodologies have been successfully applied to complex asynchronous systems, there are some common problems that restrict these asynchronous methodologies. They are generally due to the fact that circuits are often not simply single small state machines, but instead are complex systems with multiple control state machines and datapath elements combined to implement the desired functionality. Unfortunately, none of the methodologies discussed above address the issue of system decomposition. Also, these methodologies cannot design datapath elements. This is because datapath elements tend to have multiple input signals changing in parallel, and the FM assumption would unreasonably restrict the parallelism in datapath elements [Hauck 1995].

Even for circuits that the previous systems can handle, there can be performance problems with these design styles. Most obviously, the FM and BM circuits explicitly add delays to avoid certain hazard cases, decreasing performance. Also, the modules must assume the worst-case in both input data and physical properties when inserting delays, thus leading to worst-case behaviour. Finally, these circuits exhibit what can be called additive skew.

2.1.2.4 Timed circuits

Generally the delays on gates and wires on a technology can be estimated reasonably accurately. Based on this, timed circuits were proposed [Rosenblum 1985].

Timed circuits are a class of asynchronous circuits that incorporate explicit timing information during some portion of synthesis. This timing information is typically given as bounds on gate, wire and environment delays. Many of the asynchronous designs done in industry today are timed. That is, their correctness is dependent on meeting certain timing constraints. However, the techniques used for the design of these circuits are typically ad hoc, and can result in unreliable designs [Myers 1995b].

Some systematic techniques exist for the design of timed circuits. Borriello describes in [Borriello 1987] a method which uses timing information in the design of transducers, interfaces between synchronous and asynchronous circuits. Lavagno in [Lavagno 1995] develops a synthesis technique which uses methods similar to Chu [Chu 1987] and Meng [Meng 1989] to get a complex gate implementation which is then mapped to a gate library using synchronous technology mapping techniques. In both of these approaches, timing analysis is applied only after synthesis to verify that hazards do not exist. If hazards are detected, delay elements are added to avoid them, degrading the reliability and performance of the implementation. Beerel et. al. has shown in [Beerel 1994] that the more conservative SI model while resulting in somewhat large circuits actually produces faster circuits compared with the timed circuits described in [Lavagno 1995]. This surprising result can be attributed to the fact that these timed circuits often need to have delay elements added to the critical path to remove hazards [Myers 1995b].

However, the above four kinds of asynchronous circuits do not show off the important advantage of asynchronous circuits, i.e. average performance. They are still working under the worst-case performance. Unbounded asynchronous circuits, as alternatives, can solve this problem.

Within the unbounded delay model, there are various different design styles in use, each with its own merits and problems. Here we will introduce two kinds of unbounded asynchronous circuits in order of increasing number of timing assumptions. They are:

2.1.2.5 Delay insensitive (DI) circuits

Delay insensitive circuits use a delay model completely opposite to the bounded delay model: they assume that delays in both elements and wires are unbounded. In other words, a circuit whose operation is independent of the delays in both circuits (gates) and wires is said to be delay insensitive.

With a DI model, unlike the bounded delay model, no matter how long a circuit waits there is no guarantee that the input will be properly received. This forces the recipient of a signal to inform the sender when it has received the information. This function is performed by completion detection circuitry in the receiver. The sender in this protocol is required to wait until it gets the completion signal before sending the next data item.

This model also requires a new way of passing data. In synchronous circuits, the value of a wire is assumed to be correct by a given time, and can be acted on at that time. In DI circuits, there is no guarantee that a wire will reach its proper value at any specific time, since some prior element may be delaying the output. However, if a transition is sent on a wire, the receiver of that signal will eventually see that transition, and will know that a new value has been sent.

This kind of arbitrary assumption that element and wire delays are unbounded leads to significant complications in the signalling protocols. In fact, Martin has shown that the range of true delay insensitive circuits that can be implemented in CMOS are very restricted [Martin 1990a].

2.1.2.6 Speed independent (SI) and Quasi delay insensitive (QDI) circuits

SI circuits, associated with Muller's pioneering work, make the assumption that gate delays are unbounded and all wire delays are negligible (less than the minimum gate delay). QDI circuits adopt the DI assumptions that both gate and wire delays are unbounded, but augment this with isochronic forks [Martin 1989b], which are forking wires where the difference in delays between destinations of an isochronic fork is negligible. In practical, they are not much more different. A QDI model can be easily changed to a SI model. The method is shown in Figure 2.4. So we only discuss SI circuits instead of both of SI and QDI circuits in this thesis.

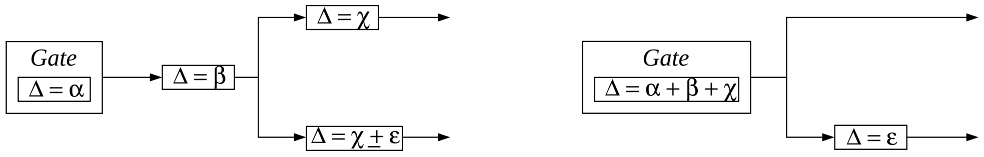


Figure 2.4 An isochronic fork (left) and an equivalent SI circuit (right).

Speed independent (SI) circuits are one of the most broadly used asynchronous circuit design styles. The main advantage of an SI circuit is its robustness to parameter variations. Additionally, an SI circuit does not need any modifications to guarantee its correctness after a technology migration.

Several kinds of SI definitions are defined based on the difference purpose, which can be found in [Yakovlev 1996a]. Theoretically, SI circuits are defined as follows [Beerel 1991]:

Definition 2.7 A circuit is SI if the state diagram representation of the circuit satisfies the following three conditions:

1) The state diagram is strongly connected;

2) Each state in the diagram can be assigned a unique bit-vector;

3) If a signal transition is enabled in one state but not fired when the circuit changes into a second state, that signal transition must still be enabled in the second state.

Because of relaxing the timing limitation, a SI model allows more implementation alternatives than DI circuits.

However, while the SI wire delay assumption may be valid in some technologies, it is obviously unrealistic in many others [Hauck 1995]. In fact, in these unbounded kinds of asynchronous circuits, it costs area and time to implement the requirement of unbounded delay. When the datapath is large, for example, completion detection will take a large amount of area and time.

2.1.2.7 Relative timing (RT) circuits

In order to get a better asynchronous circuit, RT circuits, introduced as an informal asynchronous design method for aggressive asynchronous design, are an alternative to metric timing which allows the designer to specify the effect of delays in a circuit in terms of assertions on relative ordering of events.

Metric timing requires the specification of propagation times or ranges [Myers 1995a, Young 1999]. Unfortunately metric timing analysis can explode in complexity to the extent that the synthesis and verification of even moderately sized timed circuits can become intractable [Alur 1994]. Metric timing typically needs complete characterization of all device and environment delays to achieve improvements over unbounded delay models. Complete characterization of environment delays as well as estimation of the latencies of the circuits to be synthesized is awkward.

RT circuits are designed to meet the relative orderings, or check that the restrictions are already part of the delays in the system. There exists potential possibility to improve performance, area, power and testability using RT circuits [Kondratyev 1998, Stevens 1999].

2.1.3 Signalling conventions

In traditional asynchronous circuits, the transfer of information across a channel is negotiated between the sender and receiver using a signalling protocol. Every transfer features a request (*req*) action where the initiator starts a transfer, and an acknowledgement (*ack*) action allowing the target to respond. These may occur on dedicated signalling wires, or may be implicit in the data encoding used (as described below), but in either case, one event indicates data validity, and the other signals its acceptance and the readiness of the receiver to accept further data.

There are two kinds of channels. One is the "push channel", where information flows in the same direction as the request signal. The other one is the "pull channel", where information flows in the same direction as the acknowledgement signal. These two types of channel are illustrated in Figure 2.5, in which (a) is a push channel and (b) is a pull channel.

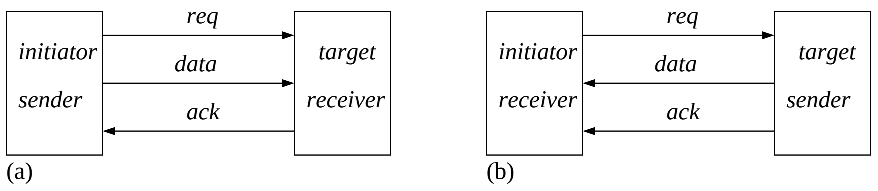


Figure 2.5 Channel signalling protocols.

The request and acknowledge may be passed using one of the two protocols described below; either a 2-phase event signalling protocol (a non return to zero scheme) or a 4-phase level signalling protocol (a return to zero scheme).

2.1.3.1 2-phase (transition) signalling

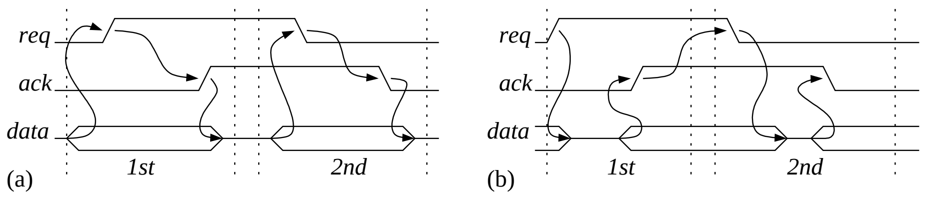


Figure 2.6 (a) 2-phase push protocol and (b) 2-phase pull protocol.

In the 2-phase signalling scheme, the level of the signal is unimportant. A transition carries information with rising edges equivalent to falling edges. Each is interpreted as a signalling event. A push channel using a 2-phase signalling protocol thus passes data using a request signal transition, and acknowledges its receipt with an acknowledgement signal transition. Figure 2.6 illustrates the push and pull data validity schemes for the 2-phase signalling protocol respectively.

Proponents of the 2-phase design style try to use the lack of a return to zero phase to achieve higher performance and lower power circuits.

2.1.3.2 4-phase (level) signalling

The 4-phase signalling protocol uses the level of the signalling wires to indicate the validity of data and its acceptance by the receiver. When this signalling scheme is used to pass the request and acknowledge timing information on a channel, a return to zero phase is necessary so that the channel signalling system ends up in the same state after a transfer as it was before the transfer. This scheme thus uses twice as many signalling edges per transfer than its 2-phase counterpart. Push and pull variants of the 4-phase signalling protocol are shown in Figure 2.7.

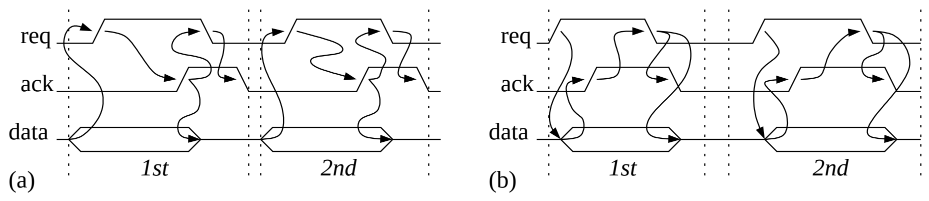


Figure 2.7 a. 4-phase push protocol and b. 4-phase pull protocol.

4-phase control circuits are often simpler than those of the equivalent 2-phase system because the signalling lines can be used to drive level-controlled latches and the like directly.

Note: conversion between the different protocols has been well documented in [Liu 1997], with many of the latch controllers documented for converting between the different 2-phase and 4-phase signalling protocols.

2.1.4 Data representation

A further dimension in asynchronous design is the choice of encoding scheme used for data representation where the designer must choose between a single-rail, dual-rail, 1-hot or other more complex n-of-m scheme. They are discussed below.

2.1.4.1 Single-rail encoding

Single-rail encoding [Peeters 1996] uses one wire for each bit of information. The voltage level of the signal represents either a logic 1 or a logic 0 (typically Vdd and Vss respectively for CMOS technology). This encoding is the same as that conventionally used in synchronous designs. Timing information is passed on separate request and acknowledgement lines which allow the sender to indicate the availability of data and the receiver to indicate its readiness to accept new data. This scheme is also known as the bundled-data approach. All single-rail encoding schemes contain inherent timing assumptions in that the delay in the single line indicating data readiness must be no less than the delay in the corresponding datapath.

Single-rail design is popular, mainly because its area requirements are similar to those of synchronous design, as is the construction of any arithmetic components using this scheme.

2.1.4.2 Dual-rail encoding

Table 2.1 Dual-rail encoding of ternary values

Logic value	*d.t*	*d.f*
Spacer “E”	0	0
Valid “0”	0	1
Valid “1”	1	0
Not used	1	1

Dual-rail circuits [Verhoeff 1995] use two wires (*d.t* and *d.f*) to represent each bit of information. Table 2.1 shows the dual-rail encoding. Here the value (*d.t*=1 and *d.f* =1) is not used. Each transfer will involve activity on only one of the two wires for each bit i.e. only one of *d.t* or *d.f* can be 1, and a dual-rail circuit thus uses 2n signals to represent n bits of information. Timing information is also implicit in the code, in that it is possible to determine when the entire data word is valid or withdrawn by detecting a level (for 4-phase signalling) or an event (for 2-phase signalling) on one of the two rails for every bit in the word. A separate signalling wire to convey data readiness is thus not necessary.

4-phase dual-rail data encoding is popular for the DI/SI design style but, as with all dual-rail techniques, it carries a significant area overhead in both the excess wiring and the large fan-in networks that it requires to detect an event on each pair of wires to determine when the word is complete and the next stage of processing can begin. As an illustration of this point, Figure 2.8 shows a circuit fragment suitable for detecting the presence of a valid word on a 4-bit datapath. In this circuit, a Muller C element is used.

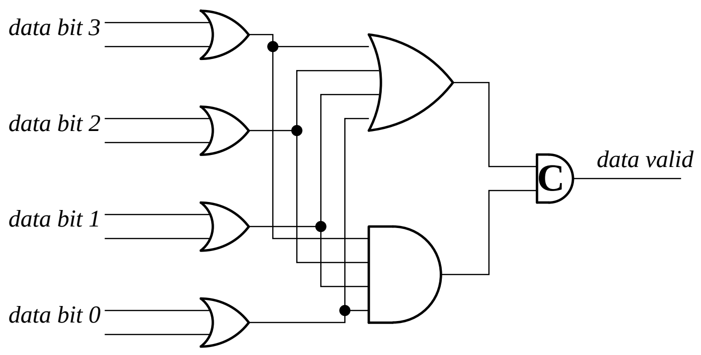


Figure 2.8 4-bit dual-rail completion detection logic.

2.1.4.3 1-hot (one-hot) encoding

In 1-hot encodings, each state q_i is represented by a vector y with y_i = 1 and with y_j = 0 for $i \neq j$. A state transition from q_i to q_j is accomplished by first setting y_j and then resetting y_i. Although the process requires two transitions, it simplifies the associated logic. The final requirement is that the next external input transition cannot occur until the entire circuit settles in a stable state. For a 1-hot encoding, this means that a new input must be delayed long enough for three event propagations through the combinational logic and two through the delay elements. With a 1-hot encoding one can implement each state variable with the same type of module; state transitions are then realized by appropriate connections between modules.

2.1.4.4 *n*-of-*m* encoding

Dual-rail encoding and 1-hot encoding are examples of an *n*-of-*m* encoding scheme where *n*=1. Coded data systems using an *n*-of-*m* code where *m* > *n* operate correctly regardless of the distribution of delay in the wires or gates, and are thus said to be delay insensitive [Bainbridge 2000].

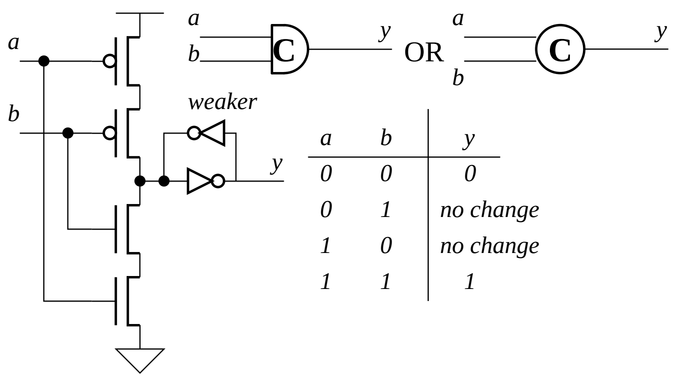
More complex codes exist (where *n* > 1) which use actions on more than one wire in a group to indicate one of a set of possible codes. These offer better utilisation of the available wires (for example a 2-of-7 code can transmit 4-bits of information over 7 wires in a delay insensitive manner), but result in large arithmetic circuits and conversion between the coded form and a single-rail scheme is more expensive than for the 1-of-*m* codes. An example can be found in [Yakovlev 1995], in which a 3-of-6 circuit is used to design an asynchronous pipeline token ring interface.

2.1.5 Basic asynchronous components

Asynchronous circuits have been studied for more than fifty years. Especially in the last two decades, because of the problems existing in synchronous circuits, this area has been focused on by a large number of researchers. So far, many very useful components have been proposed, such as the C-element, D-element, Mutex and so on. In this sub-section, we will give a brief introduction to these useful components.

2.1.5.1 Muller C-element

The Muller C-element [Muller 1959] which is shown in Figure 2.9 (often known as C-element or a C-gate) is commonly encountered in asynchronous VLSI design.



a	*b*	*y*
0	*0*	*0*
0	*1*	*no change*
1	*0*	*no change*
1	*1*	*1*

Figure 2.9 The Muller C-element: possible implementation, symbol and function definition.

The Muller C-element is a state-holding element much like an asynchronous set-reset latch. When both inputs are 0 the output is set to 0, and when both inputs are 1 the output is set to 1. For other input combinations the output does not change. Consequently, an observer seeing the output change from 0 to 1 may conclude that both inputs are now at 1; and similarly, an observer seeing the output change from 1 to 0 may conclude that both inputs are now at 0.

2.1.5.2 D-element

In a synchronous circuit the role of the clock is to define points in time where signals are stable and valid. In between clock-ticks, signals may exhibits hazards and may make multiple transitions as the combinational circuits stabilize. This does not matter from a functional point of view. In asynchronous circuits the situation is different. The absence of a clock means that, in many circumstances, signals are required to be valid all the time. This is because every signal transition has a meaning and, consequently, hazards and races must be avoided.

Handshake circuits are an alternative which can be used to implement hazard-free and race-free asynchronous circuits.

The D-element is a very popular handshake interface circuit, which has been independently proposed in [Varshavsky 1990] and [Martin 1990c]. There are two handshake protocols in a D-element. One is a master handshake and the other is a slave handshake. When the master one receives a request signal from environment, the slave one should issue a request signal. After the slave handshake receives an acknowledgement signal and the slave handshake is finished, the master one gives an acknowledgement signal to the environment. A possible implementation and the STG specification of a D-element are shown in Figure 2.10.

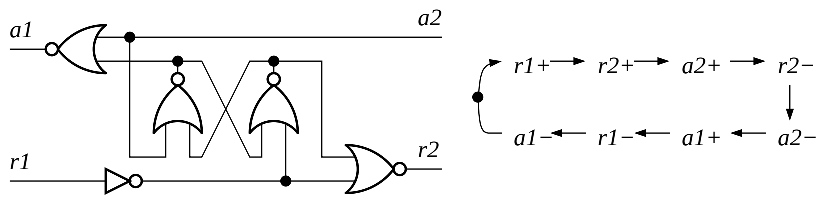


Figure 2.10 D-element possible implementation and its STG specification.

This implementation consists of four two-input NOR gates and one inverter.

2.1.5.3 Mutual exclusion elements, Mutex/Arbiter

Implementing fixed delays on the metastability characteristics of the hardware technology is not an efficient way of doing things as the delays must be present whether metastability (which is a very low probability event) has happened or not. In comparison, it is possible to make use of self-timed, speed independent circuits so that waiting/delay is only invoked when metastability occurs. This would provide, on the average, a much faster performance [Kinniment 1998].

The capability of finding out whether metastability, if should occur, has settled, is of utmost importance in such a self-timed design. This is easy to obtain by using Mutexes with metastability detectors (sometimes known as metastability resolvers) [Seitz 1980]. One simple circuit which may serve this purpose is shown in Figure 2.11.

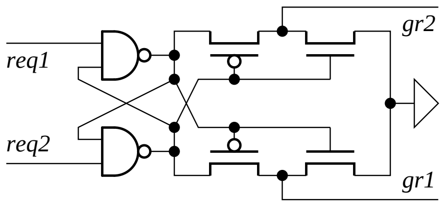


Figure 2.11 The possible implementation of Mutexes.

For the circuit shown in Figure 2.11, the intermediate outputs of the flip-flop can indeed become metastable if the inputs arrive at approximately the same time. The output of the metastability resolver, however, will not change until this metastability has settled. If properly reset and arranged with "spacer" states an arbiter constructed from this or based on a similar concept can indeed prevent metastability from passing on to subsequent circuits.

2.1.5.4 David Cells (DCs)

The simplest DC, which was proposed in [David 1977], form one kind of distributed circuit. Some extension work has been proposed [Varshavsky 1996, Yakovlev 1998]. However, it is not a popular type of asynchronous circuit and they are not enough to be used to construct self-timed circuits. In this thesis, we further extend DCs and give a formal definition as they are central to our synthesis method and asynchronous implementations.

DCs are defined as follows:

Definition 2.8 David Cells (DCs) are a non empty finite set, where $\forall\ dc_i \in$ DCs, dc_i is a tuple dc_i = <DCname, Function of SET, Function of RESET, Initial State> where

—— DCname is the name of the dc_i;

—— Function of SET is the "set" function of the dc_i. When the function is true, the dc_i will be set. That means a Token is present;

—— Function of RESET is the "reset" function of the dc_i. When the function is true, the dc_i will be reset. That means a Token is absent;

—— Initial State represents the presence and absence of a Token in the dc_i initially. If the initial value is logic high, that means the Token is held in the dc_i. Otherwise no Token is in the dc_i.

In a DC, there are a group of set signals (*s1*, *s2*, ⋯⋯, *sn*), a group of reset signals (*r1*, *r2*, ⋯⋯, *rm*), a forward signal (*fw*) and a backward signal (*bk*). A schematic representation of DCs is shown in Figure 2.12 (a).

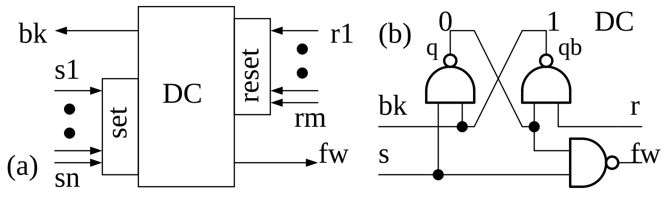


Figure 2.12 (a) Schematic of a DC and (b) The simplest DC.

In each DC, there exists a pair of complementary stable states, which can be used to represent the presence and absence of a token in a corresponding 1-safe PN place. In order to explain a DC's function clearly, the simplest DC shown in Figure 2.12 (b) is employed as an example. One of these states, *q*=0 and *qb*=1, represents the state of no token in the place. The opposite state, *q*=1 and *qb*=0, represents a token in the place. This simplest DC is a negative active component. So *s*, *r*, *bk* and *fw* signals are normally at 1. The operation of a DC in this context is as follows: starting from no token in the DC, when the set signal arrives (*s*=0), *q* will be set to 1 and owing to *r* remaining at 1, *qb* will be 0. This means that the token has been transferred to this DC. However, any events (transitions) controlled by this DC cannot be fired owing to tokens having not been removed from preceding DCs according to PN rules. After the *qb* becomes 0, any preceding DCs will be reset and then withdraw the set signal (*s*=1). Since *q*=1 and the set signal is withdrawn, the forward signal *fw* will become active (*fw*=0) to cause the firing of events controlled by this DC, thereby setting subsequent DCs to start propagating the token. This DC waits for a reset signal from subsequent DCs to reset itself back into the zero token state, which will allow events controlled by subsequent DCs to start firing. This way a DC simulates a 1-safe PN place perfectly.

For an event control system specified with a 1-safe PN, it is possible to build a circuit satisfying the specification entirely out of DCs in this manner. The input and output signals of the DCs can then be used to connect to the events being controlled (i.e. the datapath).

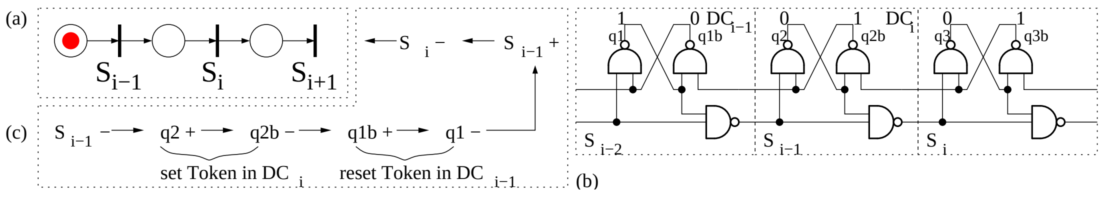


Figure 2.13 (a) Linear PN fragment (b) Its implementation using DCs and (c) Signals in two adjacent DCs.

A circuit built using simple DCs that implements the propagation of tokens in the linear PN fragment shown in Figure 2.13 (a) is depicted in Figure 2.13 (b). Figure 2.13 (c) shows the signal transitions of how two adjacent DCs interact. More details of this example can be found in [Varshavsky 1996].

2.2 Metastability

The term metastable operation refers to the prolonged transition time of a bistable device that may result if the input that causes the bistable to change state is marginal. The marginal input leaves the bistable in a “metastable state” or “metastable region” that is between the two stable states, where theoretically it may remain for an indefinite time before resolving to one of the stable states [Marino 1981]

Metastable operation is a fundamental phenomenon of sequential networks that process asynchronous inputs [Kinniment 1972, Kinniment 1976].

Metastable operation has shown itself to be very important for both synchronous and asynchronous circuit designs recently, especially asynchronous circuit designs. With continuing advances in digital technology and increasing complexity of systems with large-scale parallelism, especially those which integrate one billion transistors on a chip, there are likely to be numerous high frequency asynchronous interactions, which may result in frequent failures. As a result, some state register flip-flops may respond to an old input value and others to a new value, producing a state transition that is correct for neither value.

Theoretically, metastable operation can last an unlimited time. In practice, however, it should be settled down in a limited amount of time [Kinniment 1999].

2.3 Asynchronous circuit design methodologies

Much more difference exists between synchronous and asynchronous circuit designs than we introduced above. Because of the absence of a global clock, asynchronous circuits must operate without hazards. This is because in an asynchronous circuit, there is no way to distinguish a spurious spike from a sequence of signal changes. Any spike may be registered by a gate and cause the circuit to malfunction.

In order to ensure that an implementation is correct, some formal methods are proposed.

2.3.1 Formal methods

2.3.1.1 State graph based models

In the state graph based methods the specification is given in term of a finite automaton describing all possible states of the system [Cortadella 2002]. If the system has many events that can happen concurrently, then the total number of states in the system may be prohibitively large.

The problem with the size of the specification comes from the fact that any set of concurrent events produces an exponential number of intermediate states, although the state reached at the end is always the same. The use of the burst mode FSM specifications allows a reduction in the size of the specification. In effect, a burst of input and/or output signals captures all interleaving which would be possible had these signals been allowed to change freely. The penalty paid for such a reduction is the requirement for the difference between the moments of signal changes in one burst to be negligible.

The state based models offer a direct route for obtaining the circuit implementations. The states are encoded using binary codes and the truth tables are obtained in a straightforward manner [Semenov 1997a].

2.3.1.2 Trace based models

The trace theory was originally suggested for the verification of SI circuits by Dill [Dill 1988]. Ebergen [Ebergen 1989] suggested an approach for the synthesis of DI circuits based on the trace theory. This approach uses a top-down design methodology. A future circuit is specified using the trace theory description of its input/output behaviour. The specification is verified for delay insensitivity. Alternatively the specification can be constructed using a restricted automatically using syntax-directed translation and a predefined table of the implementation primitives.

Josephs [Josephs 1990] takes a similar approach suggesting an algebraic solution to the synthesis of asynchronous circuits. Using a special DI algebra the specification is transformed to the level of the implementation primitives.

The trace based model provides a powerful formal semantical foundation to the automated synthesis of asynchronous circuits. The circuits are hazard free by construction. This method, however, is more applicable to the verification of the already designed circuits, i.e. the designer must take a trial and error approach if he wishes to implement a particular specification. But the problem is that this model does not have provision for the verification of such important properties as a deadlock, i.e. a state from which no further advancement of the system can be made. In addition, implementations produced by a syntax-driven synthesis process are often far from optimal.

2.3.1.3 Event based models

The use of event-based models in asynchronous circuit design was prompted by the difficulties with the state space size for complex behaviours. Instead of the complete enumeration of all states of the system, an event-based formal model specifies events and relations between them.

A suitable formal model for this was found in the form of Petri nets (PNs) [Peterson 1981]. In addition, Signal Transition Graphs (STGs) were suggested independently in [Rosenblum 1985, Chu 1987] for the specification, verification and synthesis of self-timed circuits. An STG is a PN where each transition is labelled with a directed signal transition (up or down). We will introduce them in the following sub-sections. Apart from them, a model closely related to the STG model, called Change Diagrams (CDs), was suggested in [Kishinevsky 1993]. CDs have two distinctive features. Firstly, they have provision for non-repeatable events using disengagable arcs.

Secondly, they allow OR-causality, i.e. they are able to model an event whose happening is induced by any of its causes. A set of algorithms for the verification and automated synthesis of SI circuits was suggested in [Kishinevsky 1993].

Based on the above formal methods, hazard-free asynchronous circuits can be obtained, although all of them exists some problems, such as explored states and so on.

From the design point of view, specifications are an important aspect in formal design methods. So far, a large number of languages are proposed for asynchronous circuit designs from low level (signal level) to high level.

In this thesis we only introduce some specification languages concerned for modelling and synthesis of asynchronous circuits.

2.3.2 Specification languages

2.3.2.1 Petri Nets (PNs) [Peterson 1981]

PNs provide a simple graphical description of the system with an easy representation of concurrent events or a choice between alternative events. In addition, the set of researchable states can be obtained from a PN using a straightforward algorithm.

PNs do not make any assumptions about the time at which an event occurs. This makes them attractive for asynchronous circuit design. Patil [Patil 1974] was among the first who suggested to use no syntax-directed approach for the translation of PN specifications of asynchronous systems into implementations.

The following PN definitions are useful to understand PNs:

Definition 2.9 A Petri net (PN) is a tuple $N = <P, T, F>$ where

P is a set of places, and

T is a set of transitions such that $P \bigcap T = \phi$, and

F is a flow relation between places and transitions, $F \subseteq P \times T \bigcup T \times P$.

Both P and T are assumed to be finite unless stated otherwise.

A PN is represented in the form of a graph with two types of vertices: circles, which correspond to places, and bars (or boxes), which correspond to transitions. The flow relation *F* is represented by directed edges (arcs) of the graph. A bi-directional arc is used sometimes as shorthand for a pair of arcs going in the opposite directions between a place and a transition.

Each element $x \in P \bigcup T$ of a PN *N* has a set of input elements (which are connected with *x* by the arcs going to *x*) and a set of output elements (which are connected with *x* by the arcs originating from *x*). These sets of PN are called the pre-set and post-set of *x* respectively and are defined as follows:

Definition 2.10 The set $\bullet x$ and $x \bullet$ are called the pre-set and post-set of $x \in P \bigcup T$ respectively iff:

$$\bullet x = \{ y \in P \bigcup T \mid (y, x) \in F \}$$

$$x \bullet = \{ y \in P \bigcup T \mid (x, y) \in F \}$$

Structural properties of PNs define structural classes of PNs. Some useful classes are identified below:

Definition 2.11 A free choice PN (FCPN) is a PN *N* such that for any $p_i \in$ P with $| p_i \bullet | \geq 2$ the following is true: $\forall\, t_i \in p_i \bullet : | \bullet t_i | = 1$.

Definition 2.12 An extended free choice PN (EFCPN) is a PN *N* such that for any $p_i \in P$ the following is true: $\forall\, t_i, t_j \in p_i \bullet : \bullet t_i = \bullet t_j$.

A dynamic system is usually described by its structure and some initial state from which the system progresses. In terms of PNs this is defined as a marked PN.

Definition 2.13 A marked PN is a tuple *N* = <*P*, *T*, *F*, M_0> where M_0 is an initial marking of the PN *N*.

In this thesis, any PN will be treated as a marked PN unless stated otherwise.

Transitions of a PN fire from its initial marking M_0 and firing may continue while there exists at least one enabled transition. A sequence of transitions such that: σ =

$M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} M_3 \ldots$ is called a firing sequence from M_1. Obviously a transition t_i may be included several times in one firing sequence. Given a marking M_1 and a sequence σ, it is easy to know (restore) all visited markings by firing the transitions in the order of transitions in σ.

Definition 2.14 A marking M_m is said to be reachable in a PN *N* from M_1 iff there exists at least one firing sequence σ = $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} M_3 \ldots \xrightarrow{t_{m-1}} M_m$. This is also denoted as $M_1 \xrightarrow{\sigma} M_m$.

Definition 2.15 Two transitions t_i and t_j of a PN *N* are said to be concurrent iff there exists a reachable marking *M* at which both transitions are enabled and *M* contains the multiset $\bullet t_i$ and $\bullet t_j$; i.e. $\bullet t_i + \bullet t_j \subseteq M$.

Definition 2.16 Transition t_i of a PN *N* is said to be in dynamic conflict with another transition t_j at a marking *M* iff both transitions are enable at *M* and the firing of t_i disables t_j.

Definition 2.17 A labelled PN (LPN) is a tuple N^L = <*N*, *A*, *L*> where

N is a marked PN,

A is a set of actions, and

L: $T \rightarrow A$ is a labelling function which associates each transition of the PN *N* with some action from *A*.

Definition 2.18 A deadlock is a marking at which no transition is enabled.

Obviously a deadlock represents a state of the system from which no further progress can be made. Presence of deadlocks is regarded as an error in a system which operates in cycles. A deadlock can be found while traversing the RG (researchability graph) as a node with no outgoing arcs.

Another notion, closely related to the correct functioning of the system is boundedness.

Definition 2.19 A PN is said to be k-bounded iff the number of tokens in any place at any reachable marking does not exceed *k*.

When $k = 1$ (1-bounded), the net is called a 1-safe PNs. In this thesis, all PNs are 1-safe unless otherwise stated.

Definition 2.20 A transition $t \in T$ of a PN *N* is called live at a marking *M* if there exists a firing sequence $\sigma : M \xrightarrow{\sigma}$ … such that $t \in \sigma$.

Definition 2.21 A PN is called strongly live if every transition from *T* is live at every reachable marking.

A PN is called weakly live if every transition from *T* is live at M_0.

A transition which is never live usually indicates that some operation of the designed system can never be performed. A live action of an LPN is defined in a similar way.

Another important notion is persistency, i.e. the ability of transitions and actions to stay enabled while other transitions are firing.

Definition 2.22 A transition t_i of a PN *N* is said to be persistent with respect to another transition t_j if t_j is not in dynamic conflict with t_i at any reachable marking *M* enabling both t_i and t_j.

Definition 2.23 An action a_i of an LPN *N* is said to be persistent with respect to another action a_j if a_j is not in dynamic conflict with a_i at any reachable marking *M* enabling both actions.

2.3.2.2 Signal Transition Graphs (STGs) [Rosenblum 1985, Chu 1985, and Chu 1987]

An STG specification serves as a low-level description of the future circuit's behaviour. They became popular because of their close relationship to PNs, which provides a powerful theoretical background for the specification and verification of asynchronous circuits. In addition, once the binary code assignment is completed the implementation is generated by deriving the truth tables. So an asynchronous circuit can be synthesised from its STG specification if it satisfies certain criteria of implementability.

In order to understand STG very well, the following definitions are introduced:

Definition 2.24 A Signal Transition Graph (STG) is a tuple $G = <N, A, v_0, \Lambda>$ where

N is a marked PN,

A is a set of signals,

v_0 is an initial state of the STG, which is a binary vector of dimension $|A|$: $v_0 \in \{0, 1\}^{|A|}$,

Λ is a labelling function which labels every transition of N with a signal transition a+ or a- where a $\in$ A.

It can be observed from the definition that STGs are a particular case of LPNs where the set of actions is restricted to signal transitions, i.e. a+ (a-) represents the change of the value of the signal a from logical 0 to logic 1 (from 1 to 0). The set signal transitions on A is defined as $*A = A \times \{+, -\}$ so that $*a \in \{a+, a-\}$ and |*a| denotes the signal itself, i.e. |*a| = a. There also exists a less strict definition of the STG which implies that some of the transitions of the STG can be dummy transitions, i.e. they do not change the values of any signal in the STG.

STGs were introduced as a formal model for the specification of asynchronous circuits. Each transition is associated with some signal transition of the circuit or its environment. Therefore, the set of signals of an STG is usually divided into two subsets: a set of input signals and a set of output signals. Obviously, not every behaviour can be regarded as a correct one for circuit implementation. The notion of correctness of an STG is defined on the firing sequences that it can generate. First, a valid firing sequence is defined.

Definition 2.25 A firing sequence $\sigma : M_0 \xrightarrow{\sigma} M$ is valid iff for every signal a: $\exists t \in \sigma : \Lambda(t) = *a$ the following is true:

the next possible change of signal a after a+ (a-) can only be a- (a+),

the first change of signal *a* is consistent with the initial state of the STG, i.e. if the value of a is 0 (1) in the initial state, then a+ (a-) is the first change of a in any firing sequence.

The first condition is known as *switchover correctness* [Kishinevsky 1993]. The second condition is known as *stability of the initial marking*, also due to [Kishinevsky 1993].

Definition 2.26 An STG is called valid (correct) iff the underlying PN is finite, bounded and every feasible sequence in it is valid.

Definition 2.27 An STG is invalid if there exist two concurrent transitions labelled with signal transitions of one signal *a*.

The above property proves to be very important later in the analysis of STGs using the PN unfolding method. Its correctness follows from the fact that concurrent transitions are enabled together in at least one reachable marking. Hence, further advancement of the system from this marking will violate at least one of the conditions of a valid sequence.

STGs are a particular subclass of LPNs. Thus they have the same properties as LPNs defined as before. An additional important property of STGs, related to the correctness of the circuit functioning, is output signal persistency which is defined below.

Definition 2.28 A signal a_i of an STG *G* is said to be persistent with respect to another signal a_j if a_j is not in dynamic conflict with a_i at any reachable marking *M* which enables transitions labelled with both actions.

Since the set of signals is divided into sets of input and output signals, the signal persistency can be defined with respect to a set of signals.

Definition 2.29 An STG *G* is called persistent with respect to a set of signals $A' \subseteq A$ if every signal $a_i \in A'$ is persistent with respect to any other signal

$a_j \in A$ at any reachable marking *M* which enables a transition labelled with a_i.

An STG where *A'* = *A* is simply called a persistent STG. The output signal persistency has an important practical meaning.

Definition 2.30 An STG is called output signal persistent if it is persistent with respect to its output signals.

Output signal persistency is closely related to the correct operation of the circuit. It guarantees that the outputs of the circuit cannot change non-deterministically. Thus, for the observer in the environment, the circuit always reacts deterministically to any input stimuli.

Special note: in all figures of this thesis, input signals (events) at low level (signal level) are identified by underscored letters in STG and PN format specifications.

2.3.2.3 Finite state machine (FSM)

A finite state machine (FSM) is the most popular design model in computer science and engineering. The model consists of a set of states, a set of transitions between states, and a set of actions associated with either states, transitions or both states and transitions. More formally, an FSM is defined as follows:

Definition 2.31 An FSM is a tuple $<S, I, O, f: S \times I \rightarrow S, h: S \times I \rightarrow O>$. where

$S = \{ s_i \}$ is a set of states,

$I = \{ i_j \}$ is a set of input values, and

$O = \{ o_k \}$ is a set of output values;

f and *h* are the next state and output function that map a cross product of *S* and *I* into *S* and *O* respectively.

The basic FSM model can be limited or extended to represent different target architectures. For example, this model provides formal underpinning for the Burst-Mode synthesis methods [Nowick 1993].

2.3.2.4 High-level specification languages

High-level description languages specify the system in a similar way to the conventional programming languages. Among the most well-known are Martin’s [Martin 1990b] and Brunvand’s [Brunvand 1989, Brunvand 1991] compilation systems and van Berkel’s Tangram language [Berkel 1991, Berkel 1992]. Most of these methods are based on the theory of Communicating Sequential Processes [Hoare 1985] using a channel as the primary communication mechanism between subsystems.

Martin’s compilation system used a CSP-like hardware description language whereas van Berkel suggested a completely new language. The approach is, however, similar. The system is specified as a composition of the communicating processes. Once the system is specified, each process is decomposed into simpler processes. At the low level, the communication and synchronisation commands are expanded into a four-phase handshake protocol. The final circuit is obtained after re-shuffling of the transitions and the insertion of state signals to eliminate any ambiguities.

Brunvand’s approach is based on a subset of Occam. Similar to the techniques described above, the system is specified as a program. Each statement has a corresponding hardware primitive. The program is directly translated into a set of interconnected primitives. The resulting circuit is often very poor with respect to size and performance. Similar to programming language compilers, this approach uses optimisation to increase the performance and the area results. The optimisation, called peephole optimisation, is based on detecting those parts of the circuit which can be safely substituted by an already optimised fragment with an equivalent behaviour.

Recently, most researchers have focussed on commercial design languages, such as VHDL and Verilog. Because they are very popular, we introduce them only briefly.

2.3.2.4.1 Commercial design languages (VHDL/Verilog)

VHDL (VHSIC Hardware Description Language) and Verilog are two very popular commercial hardware design languages, especially in the synchronous circuit design area.

They have the following characteristics:

- Structural description of the design (netlist, schematic and HDL);
- Behavioural model for each device in the design;
- Stimulus for the design (test vectors);
- Design configuration information (specify which version of each device model to use during simulation).

Generally, they provide a wide range of modelling methods from the architecture level to the gate level. In a typical design environment, they are used in two distinct fashions:

- High level behaviour specification. Use of abstract data types for signals.
- Low level structure specification. Use of standard components supplied by a library, modelling logic level data type for signals.

VHDL/Verilog applications on asynchronous circuit designs can be found in [Blunno 2000b] and [Eles 1998].

2.3.2.4.2 Others

Apart from the above high-level languages being used in asynchronous circuit designs, there exist others, such as Tangram, developed by Philips Semiconductors [Berkel 1991], Balsa, developed by Manchester [Sparsø 2001], and so on. They are beyond the scope of this thesis.

2.4 Asynchronous communication mechanism (ACM)

The term “Real Time Network” is used for a form of real time system description in which active processing elements (activities) are interconnected through passive inter-communication data areas. The original form of real time network was proposed in [Phillips 1967]. Since then, it has been investigated by many researchers, such as Simpson [Simpson 1979, Simpson 1986] and Jackson [Jackson 1977].

An activity in a real time network is potentially continuously active and can be regarded as running in an endless loop to perform its dedicated system function. This model of activity interaction is also used to express the relationship between a network system and the environment in which it operates. Within the system boundary it is usual for the activities to be regarded as software processes scheduled by multi-tasking executives.

Although the real time network is primarily a software concept, it can also be extended to cater for processes and data which are to be directly implemented in hardware. The only qualification here is that the shared data ideas for interaction between processes should be preserved. The ability to include specialised hardware within the general scheme is an important possibility, allowing mixed hardware/software systems to be developed within a common methodological and notational framework.

These principles are particularly relevant to the field of real time, embedded, distributed and multi-processor systems. Generally such systems will react simultaneously to many uncorrelated and unsynchronised inputs. Furthermore they will generate many outputs in parallel. Implementation will often involve the use of multi processor configurations. Furthermore, design in terms of multiple processes may constitute the most natural form of software partitioning. Real time networks are ideally suited to this purpose.

Real time networks are characterised by the explicit recognition of shared data for communication and interaction between processes. The accurate and timely transfer of data between concurrent processes is of crucial importance in the exploitation of parallel architectures within distributed real time data processing systems. Techniques for solving this problem generally rely on mutual exclusion principles [Raynal 1986] to control access to shared communication resources.

The implementation of shared data objects is a very important issue in multiprocessor real time systems because the commonly used lock-based exclusion mechanism not only reduces parallelism but also introduces priority inversion in which a task has to wait for the lock on a shared data object to be released by another task with lower priority. When data sharing is between processors, a task may also be blocked waiting for tasks running on different processors. Such waiting and blocking cause problems in both the scheduler and the schedulability of the task set [Chen 1998b].

Instead of locking shared data, loops of “read-and-check” operations have been introduced to the reader to achieve non-interfered reading [Kopetz 1993, Lamport 1977]. Alternatively, the writer is made to write multiple copies of the shared data so that the reader can obtain an uncorrupted version of the shared data [Peterson 1983, Vidyasankar 1990]. These approaches are lock-free or non-blocking because no locking mechanism is involved and the shared data is accessible at any time. However, repeated operations of “read-and-check” imposes an increase in reader’s response time while writing multiple buffer slots increases writer’s execution time and space requirement. A trade-off between such time and space overheads in order to satisfy application requirements is thus necessary. For real time applications, lock-free data sharing mechanisms are suitable and beneficial only if the timing behaviour of the tasks is still predictable and analyzable. The timing overheads of “read-and-check” loops therefore should be avoided or at least restricted to a minimum.

To solve the above problems, Simpson has systematically studied these kinds of mechanisms. He proposed a loop-free mechanism in the sense that no loop of read-and-check operations is involved and presented his approach using a four-slot fully asynchronous data sharing mechanism with which a single writer communicates with a single reader [Simpson 1990]. The mechanisms can completely eliminate timing interference. A typical example of the mechanisms is Simpson's "four-slot fully asynchronous communication mechanism (ACM)" [Simpson 1990].

The important properties of this kind of mechanism are:

- **Asynchrony:** neither process may affect the timing of the other as a direct result of its communications;
- **Coherence:** data must always be passed as a coherent set, i.e. interleaved access to any data record by any process is not permitted;
- **Freshness:** the latest complete data record produced by a process must be made available for use by the other processes.

An ACM is a scheme which manages the transfer of data between two processes not necessarily synchronised for the purpose of data transfer. It is assumed that the data being passed consists of a stream of individual items of a given type. It is also assumed that the processes in question are single thread cycles, one providing and the other making use of a single item of data during each cycle.

In order to study the mechanisms clearly, one simple and elegant classification system for ACMs, developed by Simpson, was based on the number of items of data in the ACM and its modification by the reader and writer accesses [Simpson 1994]. In this classification, the basic data state is the item of data in the ACM, which the writer and reader accesses modify by a system of "destructive" and "non-destructive" reading and writing. The scheme of this classification is shown in Table 2.2.

Table 2.2 Classification of ACMs

	Destructive Reading	Non Destructive Reading
Destructive Writing	*Signal*	*Pool*
Non Destructive Writing	*Channel*	*Constant*

In Table 2.2, “*Pool*”, “*Signal*” etc. are the names given by Simpson for the protocols that demonstrate the corresponding writing and reading rules. They are defined as follows [Simpson 1994]:

Definition 2.32 A ***Pool*** is characterised by non destructive reading and destructive writing. It allows reference data to be passed from one process to another. The reference data (a single coherent record in conventional programming terms) is retained within the *Pool* where it can be consulted at any time by the reader and refreshed at any time by the writer. Special techniques can be used to maintain the coherence of the data whilst ensuring that there is no temporal interference between writer and reader when the *Pool* is implemented in private or shared memory. An initial (legal) value should be loaded in a *Pool* at build time to cater for the situation where the *Pool* is first accessed by the reader before any value has been inserted by the writer.

Definition 2.33 A ***Signal*** is characterised by destructive reading and destructive writing. It allows event data to be passed from one process to another. Event data (a single coherent record) can be overwritten at any time by the writer, but the data can only be actioned once by the reader. It follows that some data may not be actioned at all if the reader is too slow or if the writer “changes its mind” before the event data has been read. A S*ignal* should be initialised to empty at build time. The *Signal* is an important communication mechanism in real time systems, as it avoids back propagation of temporal interaction effects (i.e. the actions (or inaction) of the reader have no direct effect on the timing of the writer).

Definition 2.34 A ***Channel*** is characterised by destructive reading and non destructive writing. It allows message data to be passed from one process to another. Whereas the *Pool* and the *Signal* notionally hold a single coherent record value (from the functional viewpoint), the *Channel* has a capacity and can be used to retain a number of values between processes. Thus complete characterisation must include the capacity of the *Channel*. It is now possible that, at the time of a destructive read, several items will be available for removal. Thus an additional constraint is needed which defines that items are removed from a *Channel* in the order in which they are inserted. A *Channel* should be initialised to empty at build time.

Definition 2.35 A ***Constant*** can be regarded as configuration data. It essentially provides a "write once" capability. Generally the value of a *Constant* is established (written) at build time, and we would not expect to see any real time networks which show a process writing to a *Constant*; hence the use of a restricted form of symbol which indicates no means of connection for a writer.

In the above definitions, the destructive and non destructive properties of reading and writing are mentioned. The meanings of them are as follows:

- **Destructive Writing:** this means that the writing process for a *Pool* can never be held up. Note here that although the data in a *Pool* is always destroyed by a writer, this is not generally the case for a *Signal* where the data will usually have already been destroyed by a read. The point is that the writer behaves as if it were destroying the data, and therefore cannot be held up.
- **Non Destructive Writing:** this means that the writing process for a *Channel* or *Constant* may be held up if there is no space to put the data. Note here that an attempt to write (notionally) to a *Constant* will always stop the writer (forever) as there is no mechanism for creating space in the (notional) route.
- **Destructive Reading:** this means that the reading process for a *Signal* or *Channel* may be held up if there is no data in the route waiting to be read.
- **Non Destructive Reading:** this means that the reading process for a *Pool* or *Constant* can never be held up.

One of the most important real time properties for ACMs is the amount of blocking/waiting the data state of the ACM demands of either accessing process. The data state cannot hold up the writer in a destructive writing scheme and the reader in a non-destructive reading scheme. The writer must wait until the ACM is empty if writing is non-destructive, and the reader must wait until the ACM is full if reading is destructive.

2.5 Synthesis

Synthesis is usually defined as the translation of a behaviour representation of a design into a structural design description, similar to the compilation of programming languages into an assembly language. Each component in the structural description is in turn defined by its own behavioural description. The component structure can be obtained through synthesis at a lower abstraction level. Synthesis, sometimes called design refinement, adds an additional level of detail that provides information needed for the next level of synthesis or for the manufacturing of the design. This more detailed design must satisfy design constraints supplied with the original description or generated by a previous synthesis step [Eles 1998].

In some cases, however, the translation process does not necessarily mean the creation of a purely structural representation.

The whole synthesis process consists of several consecutive steps performed at different abstraction levels. The different steps make use of different basic implementation primitives and employ synthesis methods at the different levels. The following synthesis steps can usually be identified:

- System level: accepts an input specification in the form of communicating concurrent processes. The synthesis task is to generate the general system structure defined by processors, ASICs, buses, etc. System level synthesis operates at the highest level of abstraction where fundamental decisions are taken which have great influence on the structure, cost and performance of the final product.

- High level: the input specification is given as a behavioural description which captures the functionality of the system to be designed. High level synthesis is, therefore, also called behavioural synthesis. Using functional units and memory elements as basic primitives, a high level synthesis tool generates an implementation at the RTL level. The basic high level synthesis steps are scheduling, allocation and binding. RTL level synthesis is typically considered to be a subset of high level synthesis where allocation and binding are done automatically while scheduling is carried out by the designer.
- Logic level: this can be divided into combinational and sequential logic synthesis. The combinational logic synthesis accepts as input Boolean equations, while the sequential logic synthesis accepts some sort of finite state machine description. Logic level synthesis produces a gate level netlist as output.
- Physical level: this accepts a gate level netlist and produces the final implementation of the design in a given technology. This synthesis step depends on the implementation technology. However, the common main tasks are placement and routing.

Generally, synthesis is of key importance in digital circuit design, and even more so in asynchronous circuit design.

In this thesis, we study the synthesis process which synthesizes PN or STG specifications to generate self-timed circuits through self-timed implementations of asynchronous communication circuits. In addition, we also investigate their automatic synthesis.

2.6 Conclusions

This chapter briefly introduced a number of fundamental concepts and references on asynchronous circuit designs, metastability, asynchronous circuit design methodology, asynchronous communication mechanisms, and automate asynchronous circuit designs (synthesis). They are the background of the work in this thesis. The reader will probably want to revisit some of the material in this chapter again while reading the following chapters.

Chapter 3: New Classification and Self-Timed Implementation of Asynchronous Communication Mechanisms (ACMs)

3.1 Introduction

ACMs have been briefly introduced in Section 2.4. However, as noted in Chapter 1, classifying ACMs based on whether data accesses destroy data in the ACM has limitations. One of the most significant shortcomings of this system lies with the role played by the type *Constant* which in effect does not allow any writing. That means no communication between processes. This hardly qualifies as an ACM. In order to make ACMs more meaningful, a modified classification is proposed in this chapter.

In addition, apart from *Channel* having been implemented in both synchronous circuits and asynchronous circuits, we are unaware of any asynchronous circuit implementations of the other kinds of ACMs such as *Pool* and *Signal*. As mentioned in Chapter 1, asynchronous circuits have many advantages such as low power, fast and safety. Especially asynchronous *Channel*, Micropipeline, shows a large number of benefits [Sutherland 1989]. In this chapter we also introduce new designs for *Pool*, *Signal* and *Message*.

So in this chapter, we will present a new classification of ACMs, the relevant topics such as current situations and self-timed four-slot *Pool,* three-slot *Pool* and two-slot *Signal* mechanisms and implementations.

3.2 New classification for ACMs

An ACM has a capacity, a non-negative constant, which is the number of data items it contains. Each data item an ACM contains is either read or unread, at any time. The basic data state of an ACM consists of the number of unread data items it contains.

In order to study it clearly, the simplest model with one writing process, called writer which provides data, and one reading process, called reader is employed, which is shown in Figure 3.1.

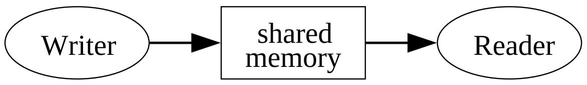


Figure 3.1 A simple ACM model.

In this model, write data accesses are divided into writing and overwriting. Read data accesses are divided into reading and re-reading. Writing increases the data state by 1 (one more unread item in the ACM) and reading decreases it by 1 (one less unread item in the ACM) while overwriting and re-reading do not modify the data state. Overwriting may occur, if permitted by the ACM protocol, only when the ACM's data state is equal to its capacity, for example, when all items of data in it are unread. Re-reading may occur, if permitted by the ACM protocol, only when the ACM's data state is 0, i.e. when none of the items of data in it is unread.

ACMs are classified according to whether overwriting and re-reading are permitted as shown in Table 3.1.

Table 3.1 New Classification of ACMs

	Non Re-Reading	Re-Reading
Non Overwriting	*Channel*	*Message*
Overwriting	*Signal*	*Pool*

In Table 3.1, the *Channel*, *Pool* and *Signal* protocol names are inherited from the classification introduced in Section 2.4 and are unchanged in effective specification. A new ACM type, *Message*, is the dual of *Signal*. That means that overwriting is not allowed and re-reading is permitted in *Message* which is defined as follows:

Definition 3.1 A ***Message*** is characterised by re-reading and non-overwriting. It allows event data to be passed from one process to another. Event data can be re-read at any time by the reader, but the data is not allowed overwriting before it is read. However, rather than “write once” capability in *Constant*, re-write is allowed in *Message* when the data has been read by the reader.

In terms of the blocking of data accessing by the data state, if re-reading is permitted there will be no holding up of the reader and if overwriting is permitted there is no holding up of the writer. If re-reading is not permitted, the reader must wait when the data state is 0. If overwriting is not permitted, writer must wait when the data state equals the ACM’s capacity.

Compared to *Constant***,** the new *Message* type is more general. In fact, *Constant* is a special case of *Message* where writing is disallowed entirely and reading is re-reading all the time.

This classification is deliberately non-specific for data item arrangements within ACMs with non-1 capacities in order to be as general as possible. Overwriting and re-reading in ACMs with capacities greater than 1 are treated as implementation issues. More details about this new classification please refer to [Yakovlev 2001] and [Xia 2002].

3.3 Current situation with ACMs

Although the protocols developed by Simpson implied software implementations and most of the detailed ACM designs seen in [Simpson 1990, Tromp 1989, Kirosis 1987, Chen 1998a] indeed assume implementations in software, this section will concentrate only on hardware implementations.

The *Channel* type of ACM, which is the common no-loss buffer and most often connected with a FIFO arrangement, is used widely in data communications. Many kinds of *Channel* have been reported, such as Sutherland's Micropipeline [Sutherland 1989]. Because the *Channel* type of ACM is very popular, it will not be discussed in this thesis.

The *Pool* type of ACM can be used to implement truly atomic data transfer with full asynchrony for both the reader and writer. This makes the *Pool* very suitable for transmitting reference data, as a kind of replacement for an analogue wire holding a variable value.

The Simpson's four-slot fully ACM [Simpson 1990] is a typical *Pool* implementation. This *Pool* ACM was studied systematically by Simpson. The dynamic properties of several possible designs are briefly discussed before concentrating on a fully asynchronous form called a four-slot mechanism.

Table 3.2 Simpson's 4-slot mechanism.

Writer	Reader
wr: d[n, s[n]'] := input;	r0: r := l;
w0: s[n] := s[n]';	r1: v := s;
w1: l := n || n := r'.	rd: output := d[r, v[r]].

One way to implement the four-slot mechanism presented in [Simpson 1990] is shown in Table 3.2 in algorithm form and in Figure 3.2 as a schematic hardware diagram. For more details about this schematic hardware diagram please refer to [Simpson 1990].

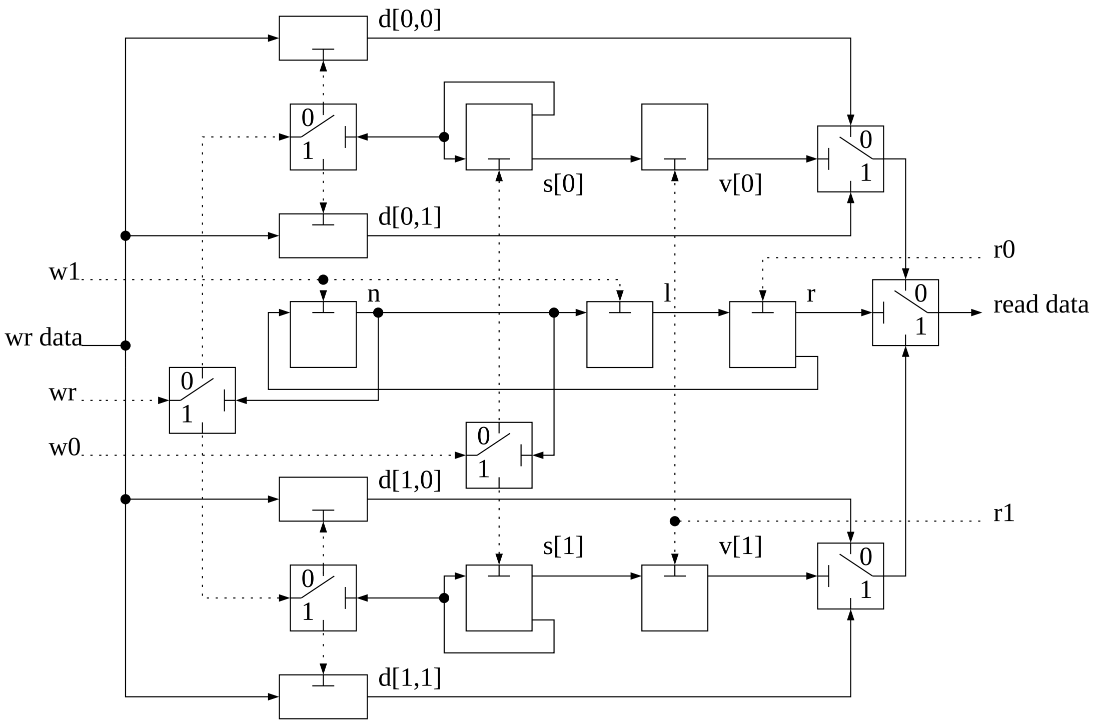


Figure 3.2 Hardware diagram for the mechanism in Table 3.2.

Simpson’s four slot mechanism shifts the problem of synchronization from the data memory (coarse granularity) to the control variables (fine granularity). This model is shown in Figure 3.3.

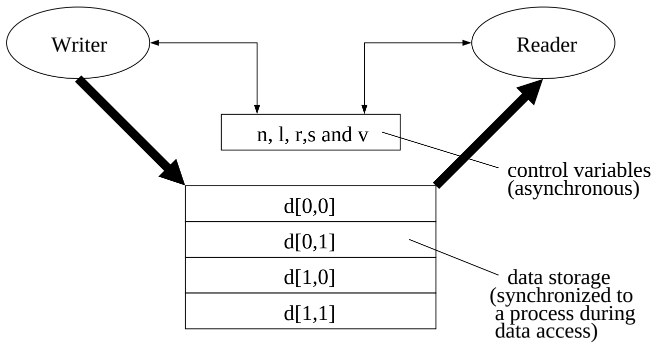

Writer
Reader
n, l, r,s and v
control variables (asynchronous)
d[0,0]
d[0,1]
d[1,0]
d[1,1]
data storage (synchronized to a process during data access)

Figure 3.3 Schematic of the 4-slot mechanism.

Here the writer and reader processes are single thread loops with three statements each. The mechanism maintains the storage of four data slots *d*[0,0] to *d*[1,1] and the control variables *n*, *l*, *r*, *s*[0..1], and *v*[0..1], which are either single bits or vectors of two single bits. The statements *wr* and *rd* are the data accesses and the other statements are used by the writer and reader to choose slots and indicate such choices to the other side.

The four-slot mechanism is designed assuming that, at the time of use, the value of a control variable is stable, rather than metastable. If at the time of use, the value of a control variable is metastable, an element of uncertainty is introduced into the system. If the control variable value is used to determine that of another control variable, the worst case is that the metastability is passed on to the other control variable. If the control variable is used directly in the selection of a data slot, the worst case is that two data slots may be accessed during one read/write action. As all control variables eventually affect data slot selection, it can be said that metastability of a used control variable may cause data coherence failures [Xia 2000a].

For this Pool ACM, however, the writer and reader are two independent processes without any timing relationship assumptions. This implies that they may be allowed to operate entirely independently in time. So, as mentioned in Chapter 2, it is impossible to avoid metastability at the hardware level, although the mechanism is very ingenious. For example, in Figure 3.2, because signals *w1* and *r0* are not synchronized, it is possible that *r0* latches an unsettled *l* into the r register. It will affect the value of *r*.

Fortunately, although the metastability can theoretically last forever, it will, in practice, settle down within a finite time [Kinniment 1998]. In the case of the value of a metastable control variable being used to determine that of another control variable, the probability of passing on this metastability is very small, making it less likely that it will eventually affect data coherence.

Based on this, the conventional way to implement this mechanism is to make fundamental mode assumptions by specifying that both the writer and reader processes must have enough delay between the acquisition of the value of a control variable, where metastability is possible, and the value is used. This delay ensures that any metastability would have settled down with a reasonably high probability by the time it is used. That means each statement in this mechanism works under the worst case timing assumption illustrated as in Figure 3.4.

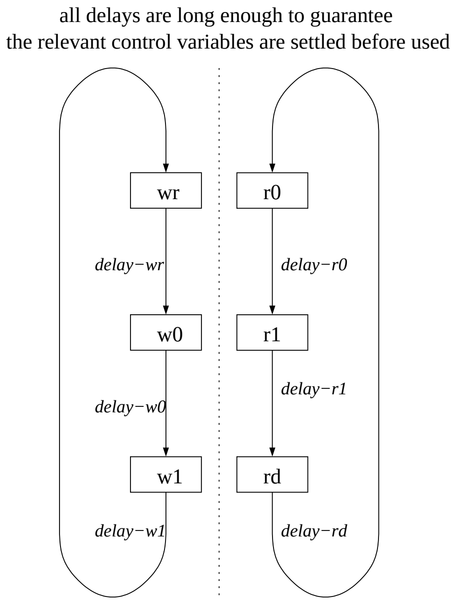

all delays are long enough to guarantee
the relevant control variables are settled before used
wr
r0
delay−wr
delay−r0
w0
r1
delay−w0
delay−r1
w1
rd
delay−w1
delay−rd

Figure 3.4 The model of 4-slot with FM assumptions.

Apart from the above study and hardware implementation of this *Pool* type of ACM, some formal methods of modelling with PNs were used to systematically analyse the properties of this kind of ACM in [Xia 2000a].

In [Xia 2000a], a PN model of the mechanism is presented. The *wr, w0*, *w1*, *r0*, *r1* and *rd* statements of this model are shown in Figure 3.5, Figure 3.6, Figure 3.7, Figure 3.8, Figure 3.9, and Figure 3.10 respectively in PN format.

The PN specification of the *w1* statement is very complex. In order to understand it easily, the truth table of this statement is given in Table 3.3. In order to explain this truth table, the transition *t15* is used as an example. In the model, if each of *p2*, *p17,* *p18* and *p20*, all holds a token, *t15* will be fired. This means that *n*=0 (*p2*), *l*=0 (*p18*), *w1 ready* (*p17*) and *r*=0 (*p20*) are the necessary conditions to fire *t15*. After that, each of *p1*, *p3, p18* and *p20* will get a new token. This means that *wr ready* (*p1*), *n*=1 (*p3*), *l*=0 (*p18*) and *r*=0 (*p20*) are prepared for the next step. This realizes the function of the *w1* statement.

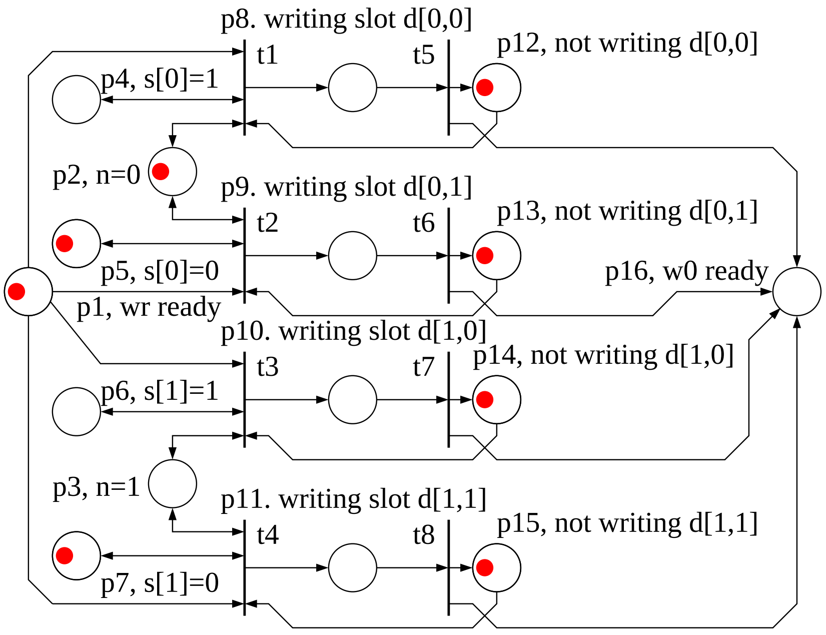


Figure 3.5The wr statement in 4-slot mechanism (wr: d[n,s[n]']:=input).

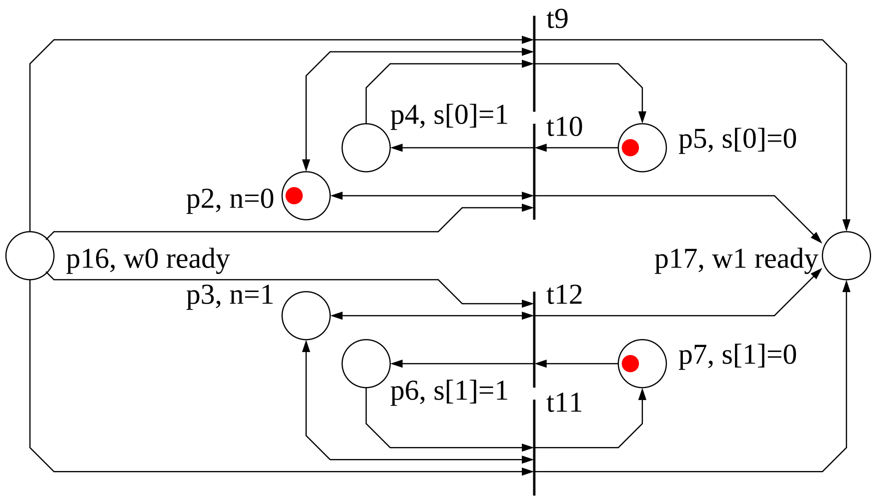


Figure 3.6 The w0 statement in 4-slot mechanism (w0: s[n]:=s[n]').

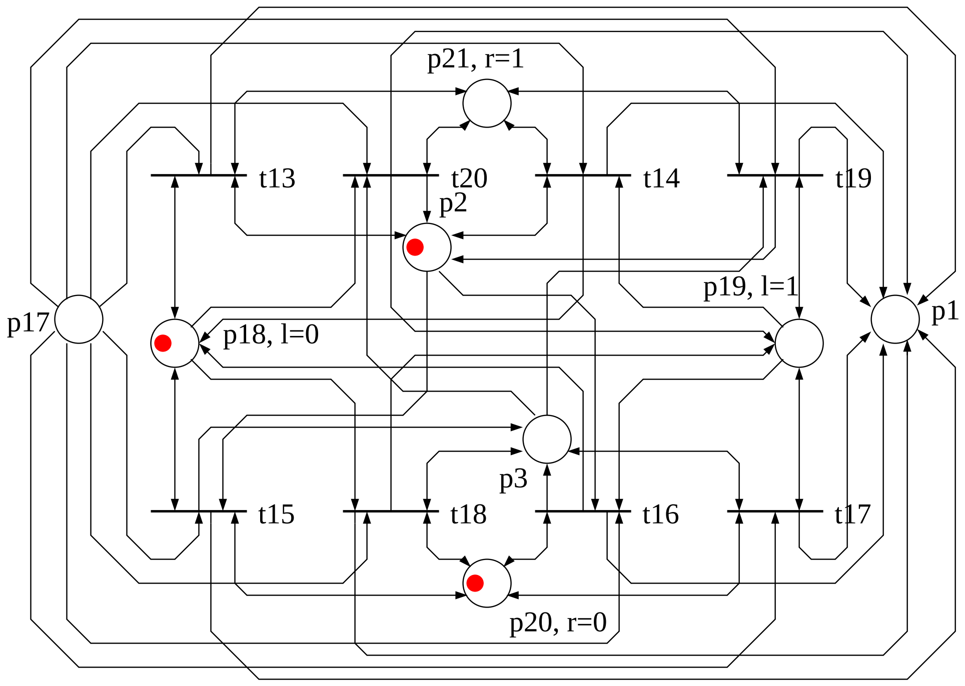


Figure 3.7 The w1 statement in 4-slot mechanism (w1: l:=n || n:=r').

Table 3.3 Truth table for the w1 statement in 4-slot ACM (w1: l:=n || n:=r').

before statement			after statement			transition firing
l	n	r	l	n	r	
0	0	0	0	1	0	t15
0	0	1	0	0	1	t13
0	1	0	1	1	0	t18
0	1	1	1	0	1	t20
1	0	0	0	1	0	t16
1	0	1	0	0	1	t14
1	1	0	1	1	0	t17
1	1	1	1	0	1	t19

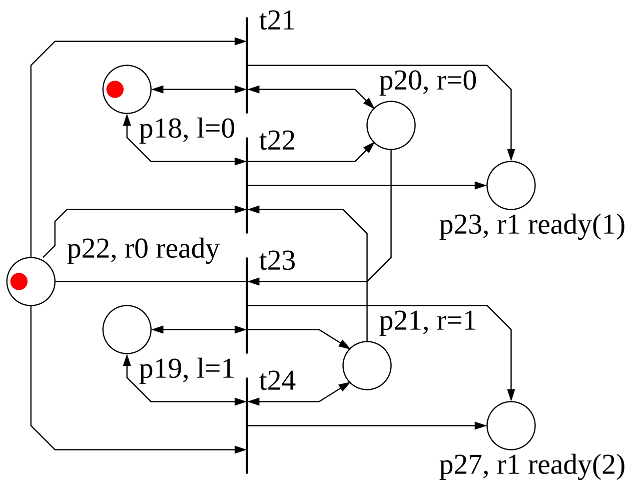


Figure 3.8 The r0 statement in 4-slot mechanism (r0: r:=l).

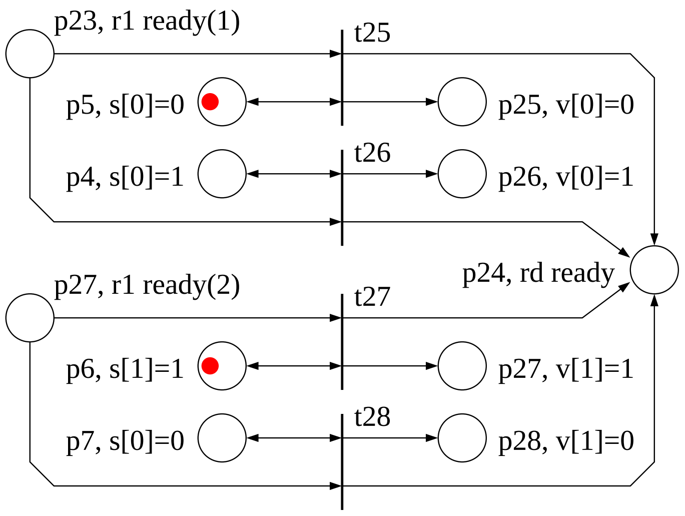


Figure 3.9 The r1 statement in 4-slot mechanism (r1: v:=s).

The results from [Xia 2000a] are that the four-slot mechanism under FM assumptions has been finally proved to maintain data coherence under normal operations, even with control variable statements taking arbitrarily long time to complete; this mechanism maintains data freshness, even with control variable statements taking an arbitrarily long time to complete; this mechanism does run fully asynchronously without hidden problems in the communications.

In addition, the Simpson four-slot mechanism has been proved to be very robust when a single control variable becomes metastable but settles to a logic level by the time it is referred to. It has shown to be robust even when a single control variable assumes an incorrect value in a random manner.

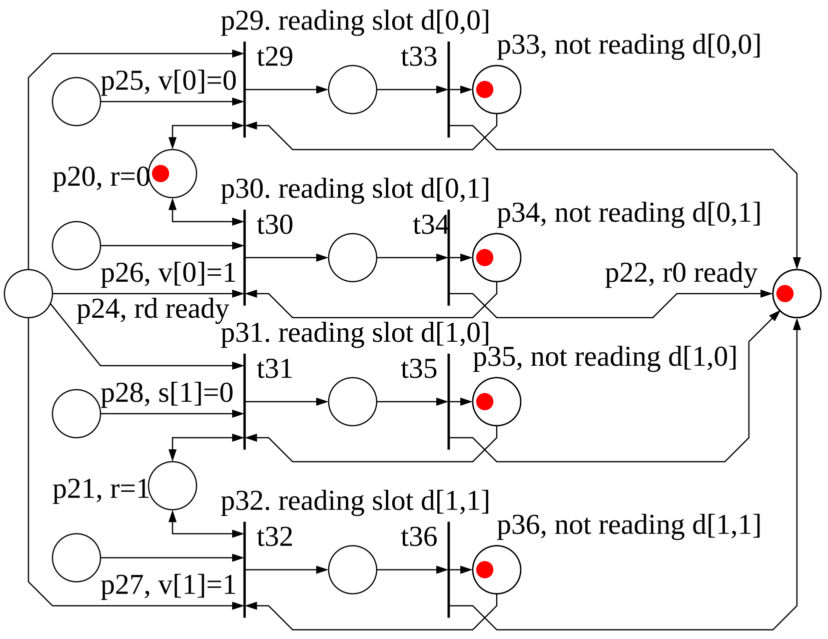


Figure 3.10 The rd statement in 4-slot mechanism (rd: output:= d[r,v[r]]).

Furthermore, the results show that it is both possible and desirable to use PNs as a modelling and analysis tool on the lowest discrete level systems.

Apart from the *Channel* and *Pool* ACMs, the other two kinds of ACMs, *Signal* and *Message*, we have not seen any hardware implementations of these two kinds of ACMs so far.

3.4 Self-timed *Pool* specification and implementation

As mentioned above (Section 3.3), the conventional way to implement ACMs is to make FM assumptions by inserting long enough delay time for each statement in the writer and reader. In order to meet the requirements, the delay should be made based on the worst case timing. Generally it will degrade performance.

In fact, metastability seldom happens in logic circuits. However, delays are always inserted regardless whether metastability happens or not. As discussed in the above section, FM assumptions are only used when metastability happens to guarantee that the control variables are settled down before processes entering the next step because synchronous circuits do not know when the actions finish. If after one action finishes the next one can happen immediately. This means that actions know how much time they take for each action. This will improve the performance. Asynchronous circuits, especially self-timed circuits, have this ability. They are running at average case speed, rather than worst case speed, because they are event-driven.

3.4.1 Self-timed four-slot *Pool*

3.4.1.1 PN modelling

This mechanism is based on Simpson’s four-slot ACM. Three statements in each process, the writer and reader, work in sequence in this mechanism. After one statement finishes, the next one can start immediately. This naturally fits the traditional start-done handshake protocol [Berkel 1992]. Based on this we refine the original mechanism of the writer process to the one shown in Figure 3.11.

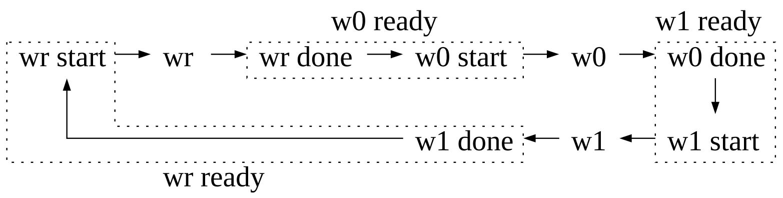


Figure 3.11 The modified writer mechanism.

In this refined version, the *wr* statement is in the “*wr start*/*wr done*” handshake protocol. This means that after the request signal *wr start* arrives, the statement *wr* can fire and then the acknowledge signal *wr done* will be given when *wr* finishes. The *w0* statement is in “*w0 start*/*w0 done*” and *w1* in “*w1start/w1 done*”.

In Figure 3.11, some handshake signal pairs may be treated as single signals because a *done* signal of one handshake protocol, in fact, is the same as the *start* signal of the next handshake protocol. For example, *wr done* and *w0 start* may be treated as a single signal, called *w0 ready* (in order to be consistent with [Xia 2000a]).

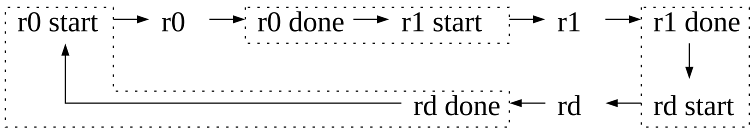


Figure 3.12 The modified reader mechanism.

As we have done for the writer, the original mechanism of the reader is also changed to a new one as shown in Figure 3.12.

There are three handshake protocols, "*r0 start*/*r0 done*", "*r1 start*/*r1 done*" and "*rd start*/*rd done*". They cover the *r0* statement, the *r1* statement and the *rd* statement respectively.

In this mechanism, the handshake protocols "w*r start*/*wr done*" and "*rd start*/*rd done*" can be guaranteed as atomic actions because the writer and reader cannot access the same slot at the same time if all the control variables have settled before these two statements start. The other handshake protocols cannot be guaranteed as atomic actions because there exists the potential possibility that the control variables (shared resources) are accessed at the same time by the writer and reader under no FM assumptions. This will result in metastability (see section 3.3 and Chapter 2).

In fact, in this four-slot ACM, *w1* and *r0*, *w0* and *r1* are critical sections because they involve access to shared control variables. In a self-timed implementation, we introduce mutual exclusion operations for these critical sections. This will affect the fully asynchronous property. However, because the statements *w0*, *w1*, *r0*, *r1* only work on a few one-bit or two-bits control variables, it should be very fast. We can ignore this interference.

In order to analysis this self-timed four-slot ACM we model it using PNs, since PNs are very suitable to specify this kind of system. We propose the following model for each statement. It is shown in Figure 3.13.

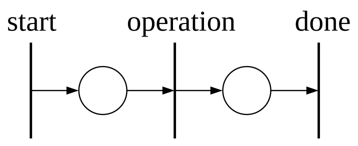


Figure 3.13 PN model for each statement.

In Figure 3.13, there are three transitions, start, operation and done. Start and done can be taken as atomic because they are only the request and acknowledge signals in practice. Operation can be very complex. It is not necessarily atomic.

As discussed above, *w1* and *r0* are a critical section. So only one, either *w1* or *r0*, can happen at a time. In the PN model, an extra place with a token is used to specify this kind of function. It is not free-choice but arbitrating choice with regard to the models of *w1* and *r0*. The same method can be used to construct a PN model for the *w0* and *r1* critical section. The whole model of the mechanism is shown in Figure 3.14.

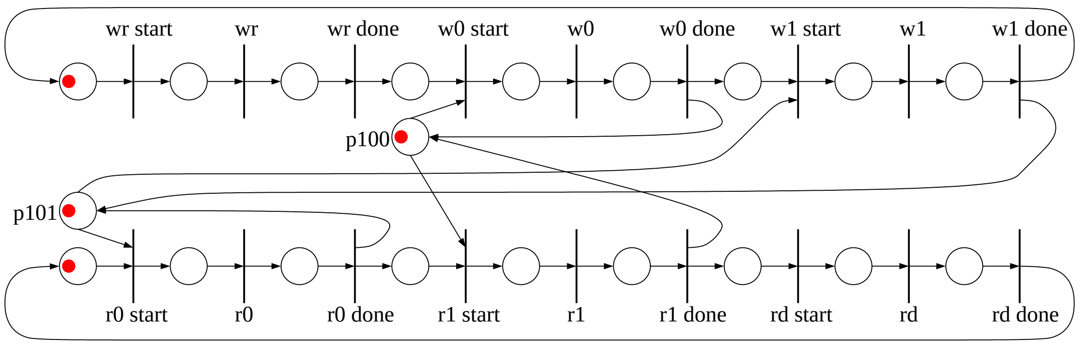


Figure 3.14 PNs model of the four-slot ACM.

In this model, there are two not free-choice but arbitrating choice places, p100 and p101, which are used to specify the two critical sections respectively. Here we call this kind of place as guards.

Using the same method introduced in [Xia 2000a], we can systematically analyse this model. After running it, the results are as expected.

For example, when *w0 start* and *r1 start* arrive at the same time or at very close times, only one of the two signals will be granted permission, and the relevant operations will happen. The other one must wait until the first one finishes.

In this PN model, the place p100 can only pass its token to one of two transitions *w0 start* and *r1 start*. As a result, only one relevant operation, either *w0* or *r1*, can happen. Following this PN model, after this operation finishes and then returns the token to p100, the other operation can start. For instance, when *wr done* finishes (*wr done*• has a token), *w0 start* is expected to happen, and at the same time, *r0 done* finishes and *r1 start* is expected to start too, because the guard (p100) only holds one token (1-safe PNs), based on the rules of PNs, so only one transition (either *w0 start* or *r1 start*) can happen. The choice is random. For example, if *w0 start* starts, *w0* and *w0 done* will start sequentially. After *w0 done* finishes, the token is returned back to p100. After that, *r1 start*, *r1* and *r1 done* can start sequentially.

This new mechanism does not affect the original function. We only put on a handshake protocol for each statement. For example, the refined w0 statement is modelled using PNs shown as in Figure 3.15.

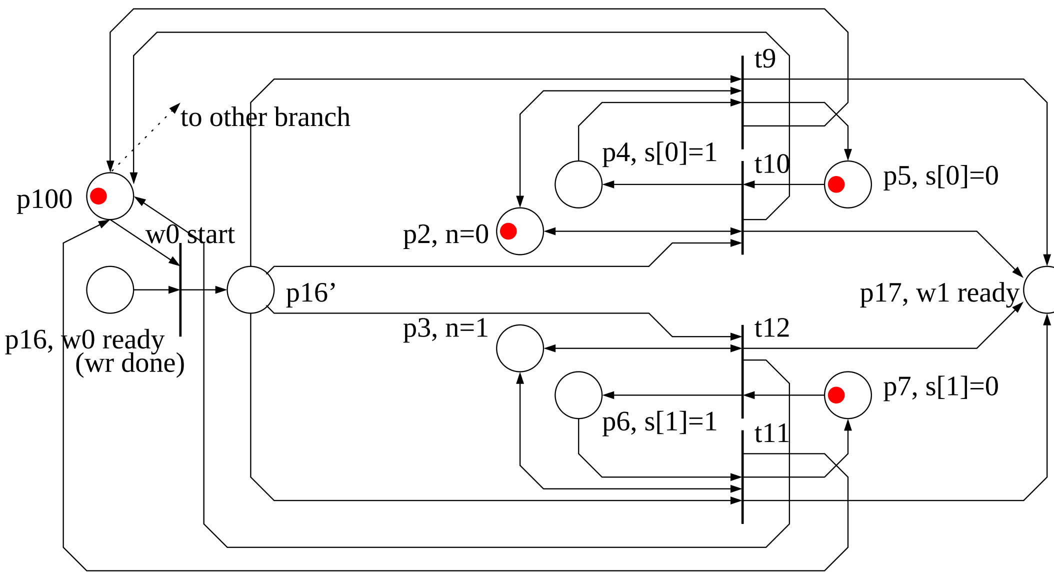


Figure 3.15 The refined w0 statement.

Compared to Figure 3.6, in this model, one place p100 acts as a guard, place p16' has been copied from place p16, and transition *w0 start* has been added. *w0 start* is a handshake signal. So the function of this model is the same as the model in Figure 3.6.

However, this mechanism has shifted metastability from the control variables to the handshake interfaces. As a result, no metastability happens on the control variables. This means that slot selection is always correct. In addition, as mentioned above, this mechanism is not a pure form of fully asynchronous communication. Because there are fine granularity data items (simple control variables) in these arbiter circuits and it only takes a short time, we can ignore this short running time. In addition, the original Simpson four-slot ACM allows that when one process is running the other process can enter several times.

3.4.1.2 Implementation

Based on the above discussions, the overall structure of the self-timed four-slot ACM is shown in Figure 3.16.

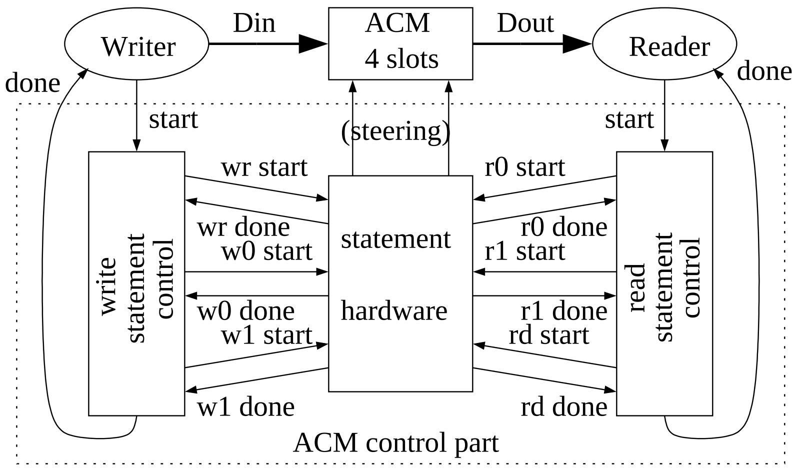


Figure 3.16 Basic structure of the modified 4-slot ACM with SI circuits.

Figure 3.16 shows that the system includes reader and writer interface control logic, and statement circuits (both control variables and data slot selection and indication). The four phase handshake bundled data protocol is used for *Din*, *start* and *done* on the writer side and *Dout, done* and *start* on the reader side. There are three internal handshake protocols (interfaces) in each process, the writer and reader. In each process, the three handshake protocols work sequentially. In practice, buffer devices can be employed to manage the three interfaces while the access processes interface with the extended ACM only once per cycle.

In order to avoid the worst case, as discussed above, handshake circuits are used to construct this ACM instead of via FM assumptions. However, because shared variables are used in this mechanism, we will deal with them using critical sections when implementing this ACM as discussed before. In self-timed circuits, Mutexes are very suitable to deal with critical sections. Based on these and the PN model shown in Figure 3.14, from the hardware implementation point of view, a hardware diagram for this self-timed ACM is shown in Figure 3.17.

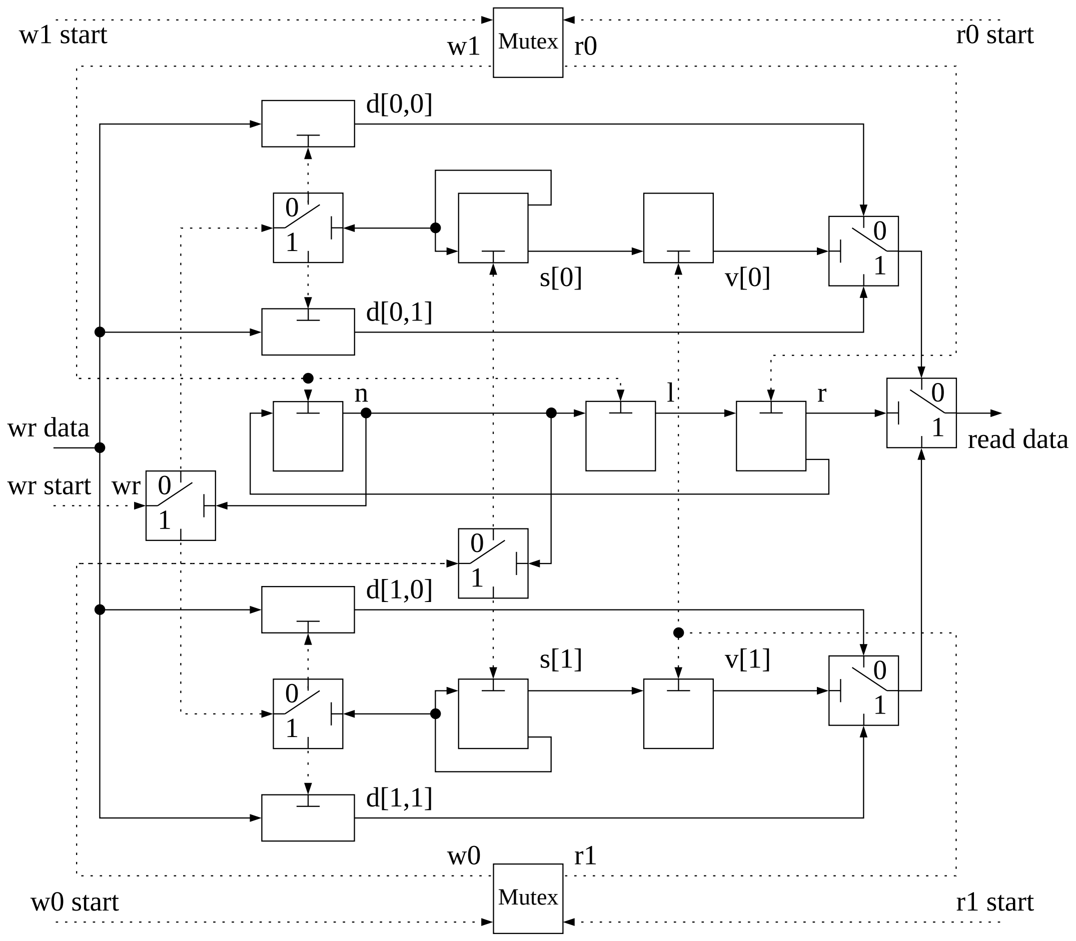


Figure 3.17 The hardware diagram of the self-timed ACM.

3.4.1.2.1 Self-timed implementation for the statements

In this self-timed implementation, we assign a handshake protocol for each statement. This means that the request signal is used to start the statement and after the statement finishes, the acknowledge signal is given. Here in order to describe concisely, we still use the original statement names, such as *wr*, *w0*, *w1*, instead of the request and acknowledge signals unless somewhere specially stated.

The *wr* and *rd* statements

The data slot access statements *wr* and *rd* are implemented with SI circuits which contain completion signals. The overall design of the hardware implementation of *wr* is shown in Figure 3.18.

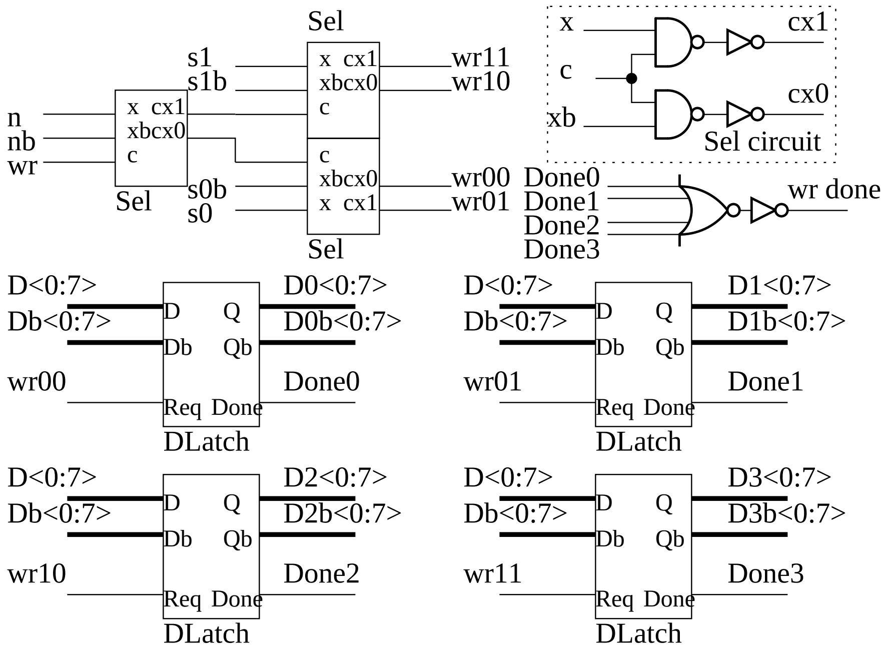


Figure 3.18 Hardware for the statement wr.

The slot steering logic consists of simple selection elements implemented in purely combinational logic. There are no hazards or SI violations; when the signal *wr* arrives, the values of *n* and *s* are entirely stable. They have been set during the previous *w0* and *w1*, which are guaranteed to have been completed.

The completion of *wr* is dependent on the data path in this implementation, which is conservative in terms of self-timed considerations. If the data items being transmitted are large in size, however, such completion may turn out to be overly complex and performance inhibiting. In this case the external protocol can be modified and the completion signal is specified to be the responsibility of the writer. Then it will be simple to implement using normal processor to memory communication assumptions.

As for *rd*, its function is to obtain the data for the reader. Because slot memory has been implemented in *wr*, here only the multiplexer is implemented which is shown in Figure 3.19. A similar method is used to implement the selection part.

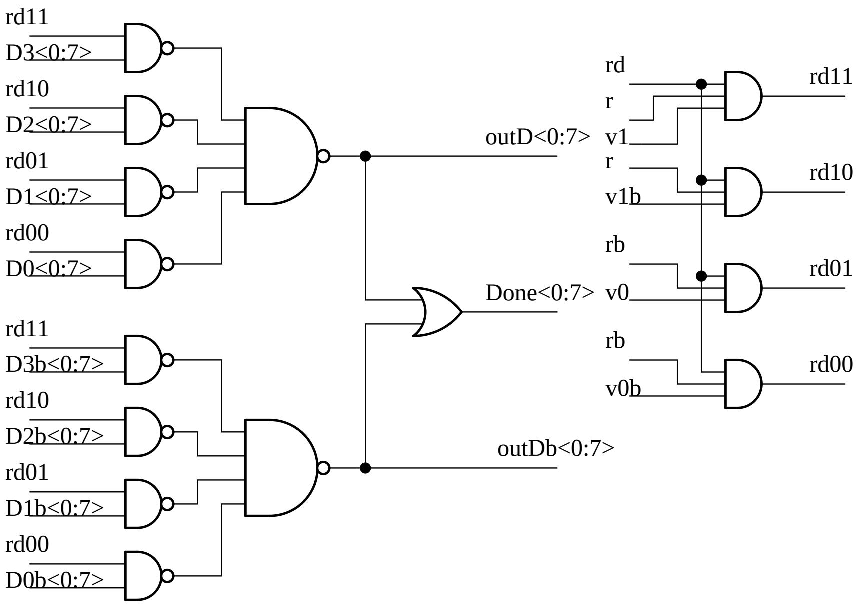


Figure 3.19 The self-timed implementation of the rd statement.

The *w0* and *r1* statements

w0 assigns a new value to control variable *s*. *r1* uses *s*. As mentioned above, *s* is a shared control variable for *w0* and *r1*. So, it is in a critical section. Only one action can start at a time. If they happen simultaneously, metastability will occur. In the above PN model, we use a guard place to guarantee that only one action can happen. In self-timed circuits, this guard can be implemented by using a Mutex introduced in Chapter 2. Two handshake request signals *w0 start* and *r1 start* go to the Mutex to apply for the grant. In Mutexes, if two requests arrive at the same time or very close, only one request signal can be granted and the grant signal will be kept until the request signal withdraws. Then the relevant operations start. The self-timed circuit implementing *w0* and *r1* is shown in Figure 3.20.

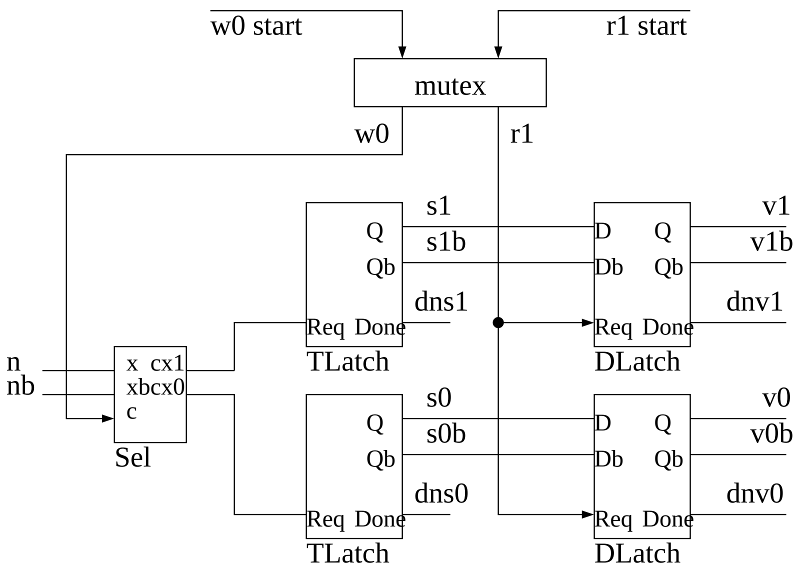


Figure 3.20 The self-timed implementation of the *w0* and *r1* statements.

Sel, DL and TL will be introduced later.

The *w1* and *r0* statements

These two statements have the same problems as in the *w0* and *r1* statements. If two statements are being carried out simultaneously, it causes metastability. This is because *r* and *l* are shared control variables. By protecting these statements with a Mutex, metastability can be avoided.

Specifically, in the case of the statements *r0* and *w1*, if signals *w1* and *r0* are generated by a Mutex so that they are never near enough in time, there will be no metastability at either *r* or *n*. In this case, any metastability would be moved to the Mutex, and only when it has settled would one of the clock pulses be generated.

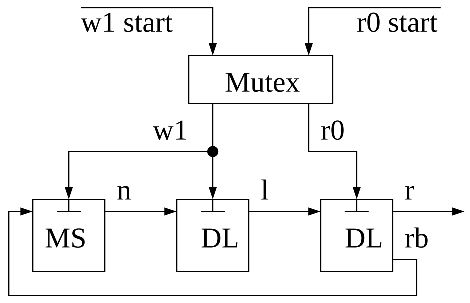


Figure 3.21 Mutual exclusion between *w1* and *r0*.

This is schematically shown in Figure 3.21, where the statement starting signals *w1start* and *r0 start* must go through a Mutex element before activating the statement hardware.

Since Mutexes require waiting for the side that lost the arbitration, the temporal relation between two processes arbitrated by such an element does not conform to full asynchronism. However, any delay is at the bit control variable level, not the slot level.

In the above implementations, the control variable assignment statements *w0*, *w1*, *r0* and *r1* are implemented by SI latch circuits within the ACM which contain completion detection signals. These latch circuits consist of self-timed dual rail master-slave latches (MS) and SI dual rail latches (DL and TL). The MS and DL are shown in Figure 3.22 and Figure 3.23 respectively. For more details please refer to [Bystrov 1999]. They fully support the handshake interface protocol of Figure 3.24.

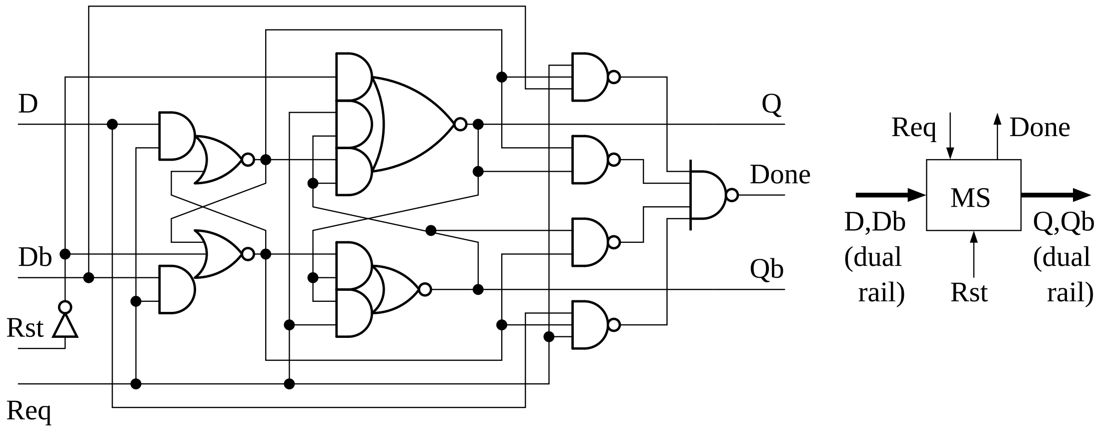


Figure 3.22 SI master-slave latch circuit and its symbol (MSLatch).

As for the other useful self-timed circuits, such as T-latch (TL), we will not introduce them here.

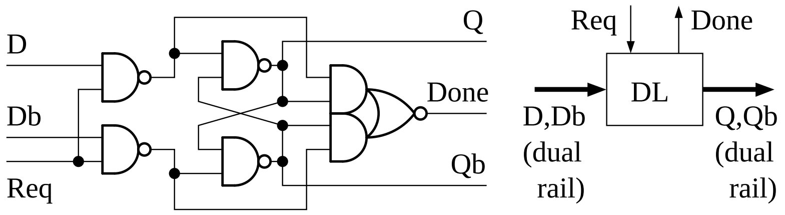


Figure 3.23 SI latch circuit and its symbol (DLatch).

3.4.1.2.2 Self-timed control circuit

This section introduces the implementation of the control circuit.

In this design, we take the approach shown in Figure 3.14 with self-timed circuits employed in the write and read statement buffers. This requires that a sequential arrangement of two statements is secured by means of control elements, so that the second statement starts only when the first has finished. It is therefore important that each statement is implemented with hardware providing a start/done handshake interface to its environment, as specified in Figure 3.25. This is shown in Figure 3.24.

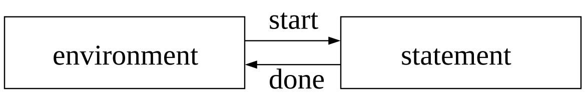


Figure 3.24 Handshake interface between a statement and its environment.

The signal sequencing of the write statement control is specified in STG format shown in Figure 3.25. This ensures the statement sequencing specified by the algorithm in Table 3.2. The reader statement control has essentially the same STG.

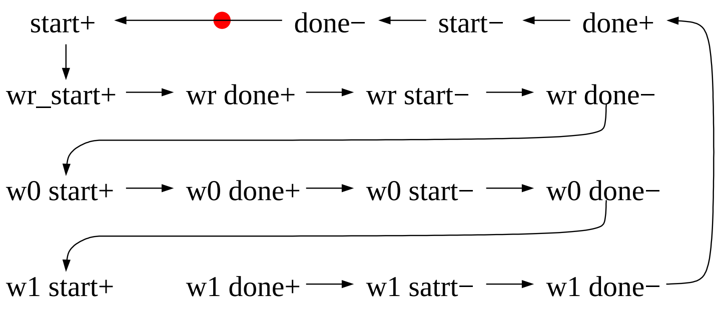


Figure 3.25 STG specification of write statement control.

In order to retain an element of regularity and extendibility, a kind of circuit known as a DC (see Chapter 2), is chosen as the building block with which to assemble this circuit.

A control circuit managing four consecutive handshakes needs four DCs connected in series. By organising the initial condition so that only one of the DCs is active (q=1 and qb=0) and the others are "spacers" (q=0 and qb=1), the circuit shown in Figure 3.26 would produce an STG shown in Figure 3.27.

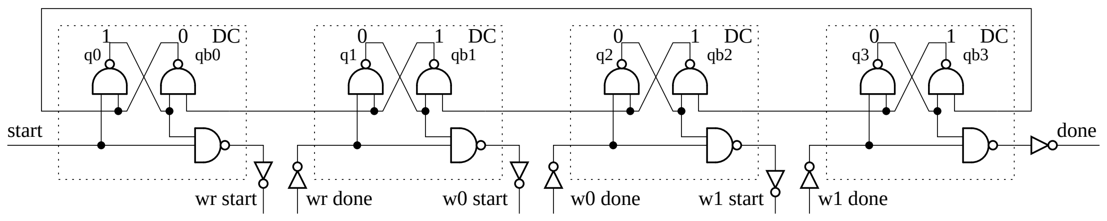


Figure 3.26 Write statement control circuits using DCs.

$$\begin{tikzcd}
\text{start+} \arrow[d] & \text{done}- \arrow[l, "\bullet"] & \text{q3}- \arrow[l] & \text{q3b+} \arrow[l] & \text{q0b}- \arrow[l] & \text{q0+} \arrow[l] & \text{start}- \arrow[l] & \text{done+} \arrow[l] \\
\text{wr start+} \arrow[r] & \text{wr done+} \arrow[r] & \text{q1+} \arrow[r] & \text{q1b}- \arrow[r] & \text{q0b+} \arrow[r] & \text{q0}- \arrow[r] & \text{wr start}- \arrow[r] & \text{wr done}- \arrow[dlllllll] \\
\text{w0 start+} \arrow[r] & \text{w0 done+} \arrow[r] & \text{q2+} \arrow[r] & \text{q2b}- \arrow[r] & \text{q1b+} \arrow[r] & \text{q1}- \arrow[r] & \text{w0 start}- \arrow[r] & \text{w0 done}- \arrow[dlllllll] \\
\text{w1 start+} \arrow[r] & \text{w1 done+} \arrow[r] & \text{q3+} \arrow[r] & \text{q3b}- \arrow[r] & \text{q2b+} \arrow[r] & \text{q2}- \arrow[r] & \text{w1 satrt}- \arrow[r] & \text{w1 done}- \arrow[uuu, bend right]
\end{tikzcd}$$

Figure 3.27 STG of the circuit in Figure 3.26.

In Figure 3.27, if we remove all q’s and qb’s events, the remainder is the same as the STG specification in Figure 3.25. Here the q’s and qb’s are the internal signals of the flip-flops within the DCs and are not directly made use of by the control logic. They serve the same purpose as the CSC signals from a Petrify solution. From the STG, it is clear that this circuit can be used for both the writer and reader statement control logic blocks for the appropriate statement handshakes.

The similar method is used in the reader part to construct the control circuit for the reader.

3.4.1.3 Circuit analysis and conclusions

This four-slot *Pool* design has been put through the VLSI design flow using the Cadence tools. Top-level simulations, both analogue and digital, have been carried out.

3.4.1.3.1 Analogue simulation results

Analogue simulations have been run for the four-slot ACM design with the Spectre simulator from within the Cadence tools. Apart from studying the entire circuit under a number of possible operating conditions, effort has been concentrated on the behaviour of Mutexes and the entire system when the statements *r1* and *w0* occur simultaneously. The result of this study is given below. Similar studies have been done for the case when the statements *r0* and *w1* occur simultaneously, with similar conclusions.

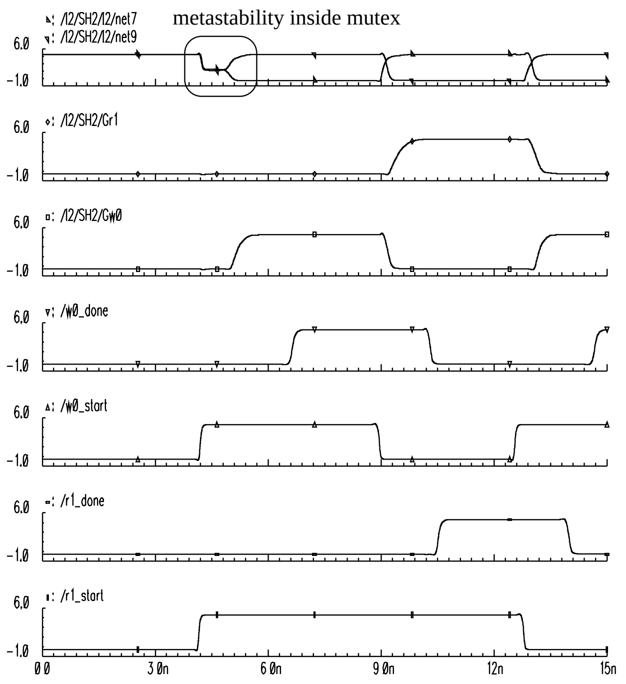


Figure 3.28 Analogue simulation waveforms with metastability within Mutex.

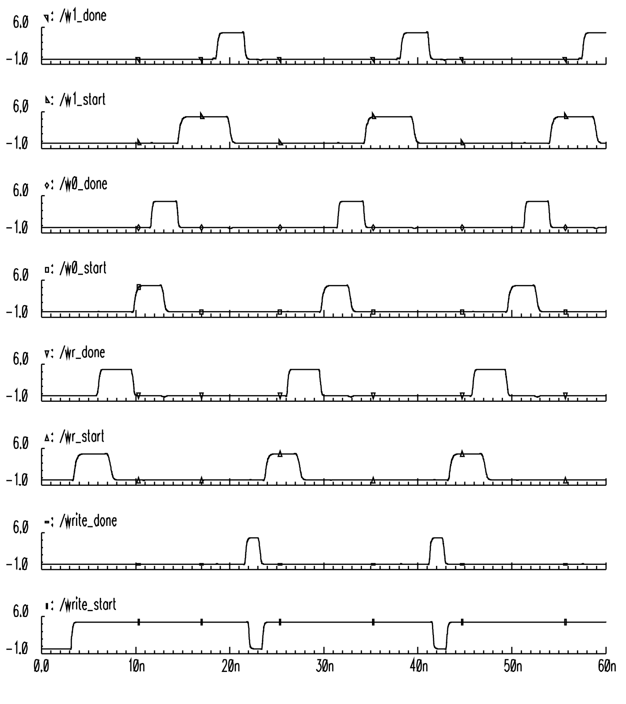


Figure 3.29 Analogue simulation waveforms showing general handshake operations.

Figure 3.28 shows the transient response of the handshake signals associated with the statements *r1* and *w0* when metastability has been generated within the Mutex between these statements, because the requests are close in time. The metastable response within the Mutex (at signals *net7* and *net9*, between 4 and 5 ns) only delays the response of the rest of the system and is never propagated out of the Mutex. The Mutex is also shown to have successfully created mutual exclusion between the two statements.

Figure 3.29 shows the general handshake operations on the writer side. Similar results have been obtained for the reader side. This conforms with the specifications give in Figure 3.25.

3.4.1.3.2 Digital simulation of the four-slot Pool

Digital simulations have been run for the four-slot ACM design from the Cadence toolkit on the circuit. In order to maximally reveal data coherence and data freshness properties, a writer process was created in Verilog code which sends byte data for 255 cycles, with the data increasing in value from 1 to 255. The data received at the reader end was then collected for analysis. The writer and reader processes have been programmed so that their extra ACM delays take exponentially distributed time lengths with mean values varying from 10 to 500ns.

From these simulations, no data coherence and data freshness violations have been observed. This is true even when, owing to the stochastic nature of the reader and writer extra-ACM delays, one side traverses many cycles with the other side stuck. This is to be expected because this *Pool* design is a faithful implementation of the four-slot algorithm which has been verified analytically to maintain these properties if the FM assumptions hold. This *Pool* design, by dealing with the issue of metastability explicitly and using SI circuits, makes sure that no statement gets started without its preceding one having completed. From the state-transition system point of view this is functionally equivalent to the FM assumption holding in the original four-slot algorithm.

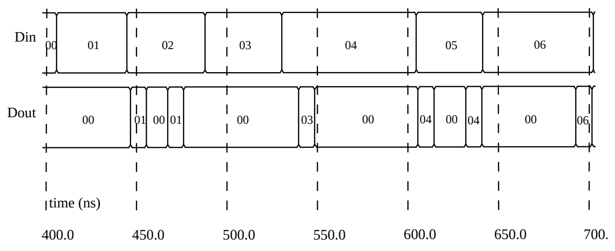


Figure 3.30 Din and Dout value sequence at the beginning of a simulation.

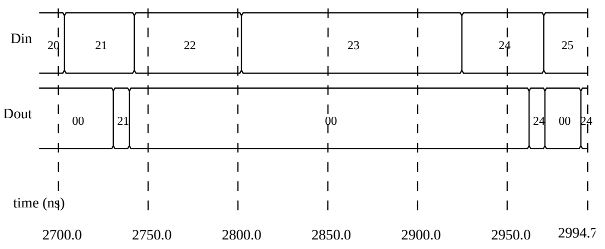


Figure 3.31 Din and Dout value sequence in the middle of a simulation.

The data value sequences for *Din* and *Dout* from a simulation run is selectively shown in Figure 3.30 and Figure 3.31. Date coherence, freshness, sequencing, loss and re-reading properties can all be obtained by observing such sequences. For instance, the loss of data items 22 and 23 can be observed in Figure 3.31, and the re-reading of data items 01 and 04 can be observed in Figure 3.30. The second reading of data value 04, while data value 05 has clearly been available for some time, does not violate data freshness according to the definition found in [Xia 1999b]. According to this definition, when the statement *r0* and *r1* overlap with the statement *w0* and *w1*, the reader is allowed to obtain the latest but one item of data in the *Pool*. This is because the location of the slot where the latest data item resides is indicated by the writer through *w0* and *w1* and obtained by the reader during *r0* and *r1*. When these statements overlap in time the *Pool* should not be expected to always pass the latest data item.

The simulations show that the functional behaviour of the circuit is as expected. Analogue simulations have established that metastability does not propagate through the system, but is contained within the Mutexes. Digital simulations have revealed the data coherence and data freshness properties for the design.

In our four-slot *Pool* implementation, we have slightly relaxed the fully asynchronous requirement. The results show that this self-timed four-slot *Pool* realizes the functions of the original fully asynchronous *Pool* very well. In addition, it should be faster and safer than the FM one (see the discussions above).

3.4.2 Differences between FM and SI solutions

The solutions proposed in [Simpson 1990] provide full asynchronism for the writer and reader. They are, however, dependent on FM assumptions, namely that the switching processes in the hardware implementation settle between adjacent statements.

In the SI solutions, there is certainly not an absolute temporal division between the reading and writing sides within the ACM, because of the waiting required by the Mutexes. It is worth noting, however, that such waiting only happens during control variable setting statements (*w0, w1*, *r0* and *r1*), and, that the data slot access statements *wr* and *rd* are not affected directly. In other words, by retaining the ACM algorithms, the broad idea of realising atomic data transfer by using safe bit registers is retained. In effect, critical sections are moved from data slots to bit variables.

Temporal independence, when required, is required between the reader and writer processes and not between the internal read and write sides of the ACM. From Figure 3.16, it is clear that there are two pairs of handshakes where such temporal divisions can be maintained in the new design. These are the global read and write *start/done* interfaces. For instance, rather than the more rigid protocol normally associated with the handshake, the writer can be specified to follow the more flexible protocol outline below:

- Issue *start* to the write side of the ACM;
- Wait for *done* from the ACM;
- In the absence of *done*, wait for a predetermined maximum time period (FM assumption);

- Continue its own cycle, knowing whether *done* or the expiration of the maximum time period has happened. This allows the writer client to decide whether to operate in FM or SI fashion. While still realising the potential of speeding up the response provided by the SI solutions, it also allows an upper bound for the complete ACM write cycle to be specified, therefore effectively decoupling temporally the writer process from the reader one. A similar arrangement can be employed at the reader side.

Such a maximum waiting period can be easily obtained by finding the normal time expenditure of all statements and assuming that metastability happens at an arbitration point (it is trivial to show for the four-slot ACM that in a single cycle of operation only one of the arbitration points could be activated, assuming that both the read and write sides of the ACM are implemented using the same hardware technology on the same chip) and then this side loses the arbitration. The statement timing can be obtained through simulations, since the entire ACM is designed in hardware "in house". The metastability settling time can be estimated by the method outlined in [Kinniment 1999], where it is demonstrated that 5ns is sufficient time for all metastability to have settled firmly in modern CMOS technology with "practically" probability 1.

3.4.3 Self-timed three-slot *Pool*

In Simpson mechanisms, because the control variables may not be stable, they may settle to the opposite values when used. In order to guarantee the properties of ACMs, four slots are shown to be needed. For more details please refer to [Simpson 1990].

Since the self-timed four-slot ACM has shifted the metastability from the control variables to handshake interfaces, all control variables used in this mechanism will be stable before they are used. The case of settling to the opposite values will not happen. On the other hand, slot-type ACMs take up a large number of memory elements in implementations. Generally they are very expensive in modern computer systems. We would like to construct ACMs limiting the occurrence of redundant slots.

In [Simpson 1990], the other three mechanisms, one-slot, two-slot and three-slot mechanisms, have been proposed as well. However, they cannot be implemented correctly under FM assumptions. The reasons were explained clearly in that paper.

In this thesis, we would like to study the one-slot, two-slot and three-slot ACMs again based on the implementation method used in the self-timed four-slot *Pool*. In the other words, we slightly relax the full asynchronous requirement and introduce mutual exclusion on the handshake interface signals to the one-slot, two-slot and three-slot *Pool* types of mechanism to see what happens.

The one-slot mechanism only contains one possible place for data transit and offers no choice to the writer and reader when they come to access the mechanism. Integrity of the one-slot mechanism is only preserved if the writer and reader never overlap. However for the general case of asynchronous writer and reader there always is the danger of a loss of coherence, although when the data read is coherent it will always be the freshest available. This is not what we expect.

The two-slot mechanism is sometimes called the swung buffer since alternate data items are written to alternate slots which are then swung into visibility for output. In any form of signal writer to signal reader asynchronous communication it can be assumed that the access operations are of finite duration and that successive operations on each side are separated by a finite interval. Thus a reading process takes a certain amount of time to obtain data (read duration) and then makes use of this data in subsequent computation before returning for further accesses. The writing process, following a write operation, takes a certain amount of time to prepare new data (interval between writes) before returning for further accesses.

In the case of the two slot mechanism, a read starting between writes will access coherent data up to the start of the next-but-one write, whereas a read starting during a write will only access coherent data up to the start of the next write. Thus, in the absence of any overall control, the mechanism can only be guaranteed to work satisfactorily if the interval between successive writes is always greater than the duration of any read. This condition holds in a range of applications so the two-slot mechanism is of some practical significance. Like the one-slot mechanism, failure results in the coherence requirement not being met.

3.4.3.1 PN modelling of self-timed three-slot *Pool*

The original three-slot mechanism from [Simpson 1990] is shown in Table 3.4.

Table 3.4 Simpson's three-slot mechanism

Writer	Reader
wr: d[n] := input;	R0: r := l;
w0: l := n;	rd: output := d[r].
w1: n := differ(l, r).	

This mechanism is similar to the four-slot mechanism in that the writer and reader processes are single thread loops with three and two statements each respectively. *n*, *r*, and *l* are control variables.

The original three-slot *Pool* does not guarantee the atomic transfer of data through the slots (a property known as “data coherence”) if the statement *r0* cannot be regarded as atomic relative to the statements *w0* and *w1*. In other words, if before the beginning and the end of a single *r0* statement, a sequence of *w0* and *w1* is started and completed, the reader and writer may clash on the same data slot simultaneously. However, if the statement *r0* can be regarded as atomic relative to the statements *w0* and *w1*, the *Pool* maintains atomic transfer of data through the slots [Xia 1999b]. This *Pool* has also been shown to maintain “data freshness”, whereby the reader always obtains the most up-to-data data item from the *Pool*, under the same assumptions of atomicity [Xia 1999a]. This is similar to our self-timed four-slot *Pool*, which is realized by introducing Mutexes in implementation.

In order to systematically study this three-slot ACM, firstly a PN model is built according to the method modelled the self-timed four-slot ACM. The PN model is shown in Figure 3.32.

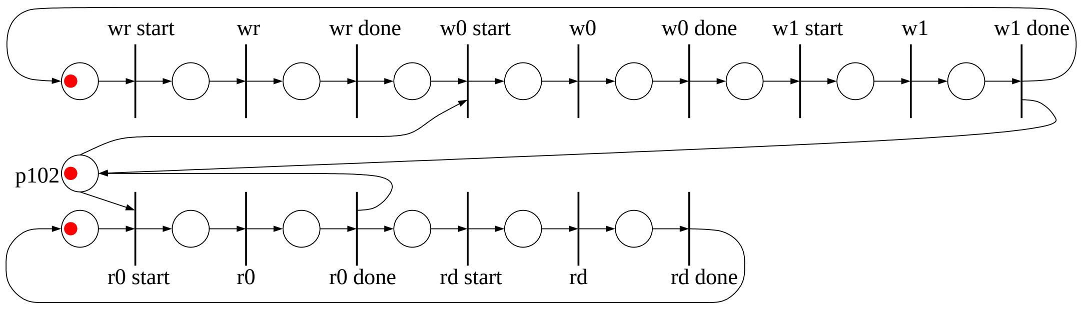


Figure 3.32 PN model of the self-timed three-slot ACM.

In Figure 3.32, we introduce a guard place (p102) in the model. It is used to guarantee that only one, either *r0 start/r0 done* or *w0 start/w1 done*, can start at a time. After running this model using the PN rules, the mechanism shows as expected.

3.4.3.2 Self-timed implementation

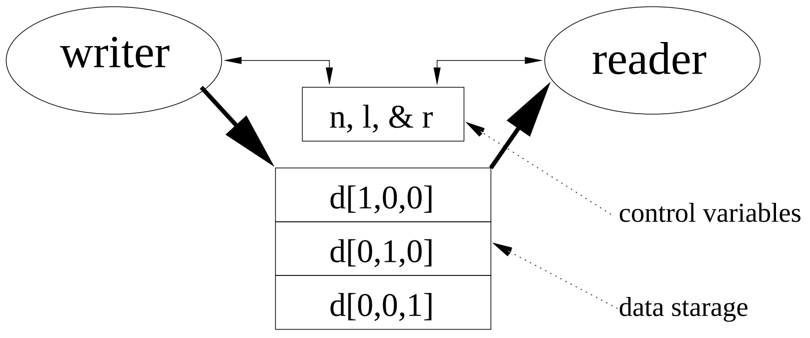


Figure 3.33 Schematic of three-slot mechanism.

The three-slot *Pool* implementation includes reader and writer interface control logic, statement circuits (both control variables and data slot selection/data path steering). The mechanism, shown schematically in Figure 3.33, maintains the storage of three data slots, *d[1]*, *d[2]* and *d[3]*, corresponding to *d[1,0,0]*, *d[0,1,0]* and *d[0,0,1]* respectively in Figure 3.33, and the control variables *n*, *l* and *r*, which are ternary signals. One-hot encoding is adopted here in order to implement SI easily. The statements *wr* and *rd* are the data accesses and the other (control variable) statements are used by the writer and reader to choose slots and indicate choices to the other side.

Based on the implementation method used in the self-timed four-slot ACM, the overall structure of this *Pool* is shown in Figure 3.34.

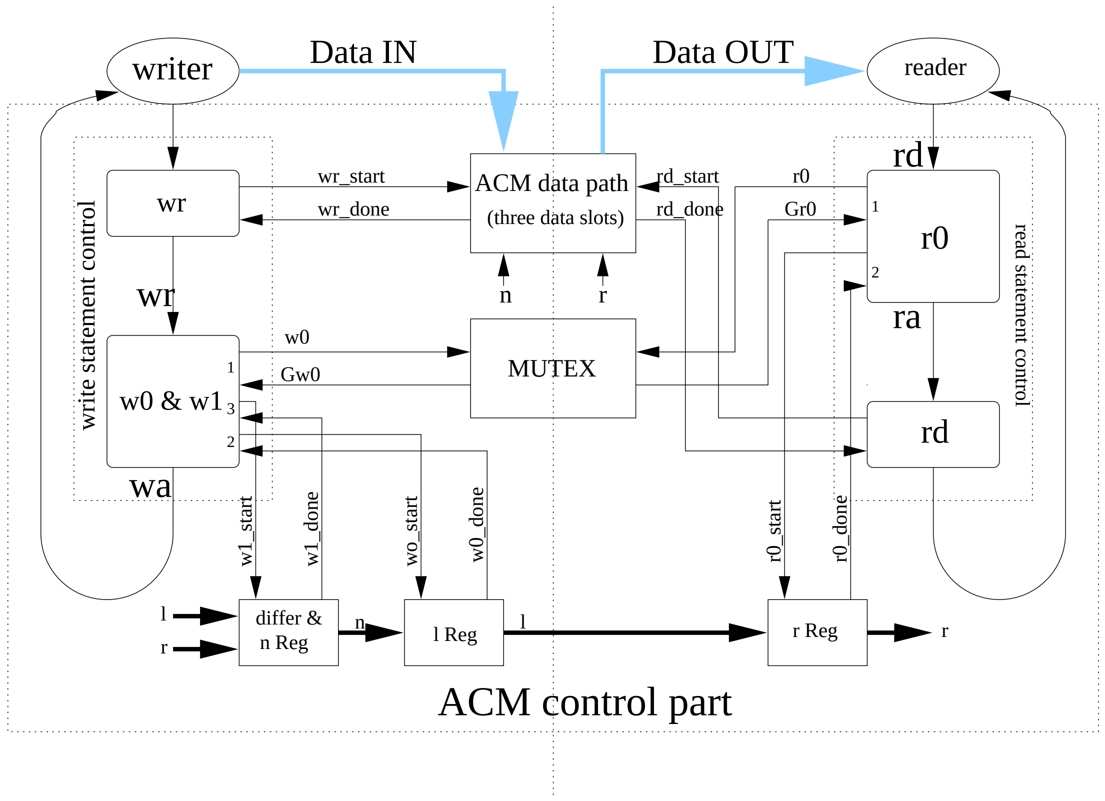


Figure 3.34 The structure of the three-slot *Pool*.

In Figure 3.34, for the same reason as in the four-slot ACM, a Mutex is used to implement the critical section in this mechanism.

3.4.3.2.1 Implementation of the wr and rd statements

This *wr* implementation is very similar to the one in the four-slot ACM. The block diagram of the *wr* implementation is shown in Figure 3.35. Because *n* is 1-hot encoded, only one of *n1*, *n2* and *n3* can be 1 at a time. This guarantees that in each cycle, only one slot can be written. Because *n* is settled before it is used, no data coherence violation happens. A similar method in the *wr* circuit of the four-slot ACM is used to construct the *rd* circuit (which is not shown here).

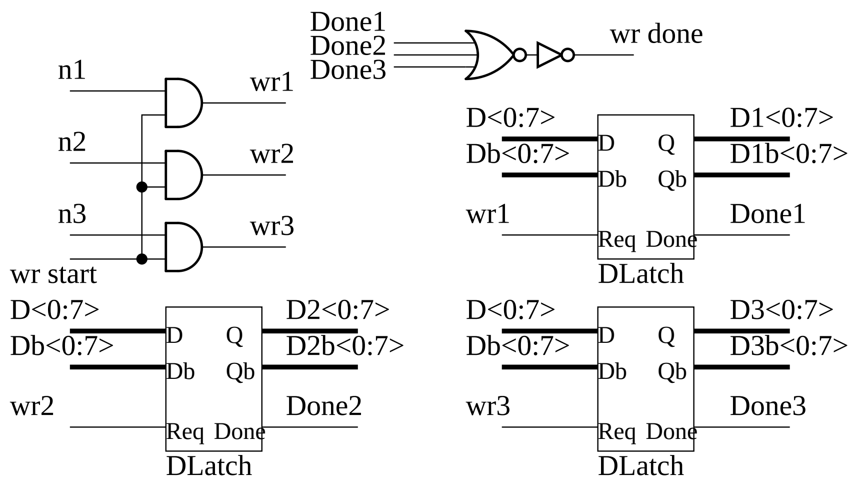


Figure 3.35 The wr statement in three-slot ACM.

3.4.3.2.2 Implementation of the w0, w1 and r0 statements

Because (*w0*, *w1*) and *r1* operate in a critical section, only one of (*w0*, *w1*) and *r1* can start. As mentioned before, Mutexes can be used to implement this function in self-timed circuits. The method used in the four-slot ACM is employed here, and the block diagram of this function is shown in Figure 3.36.

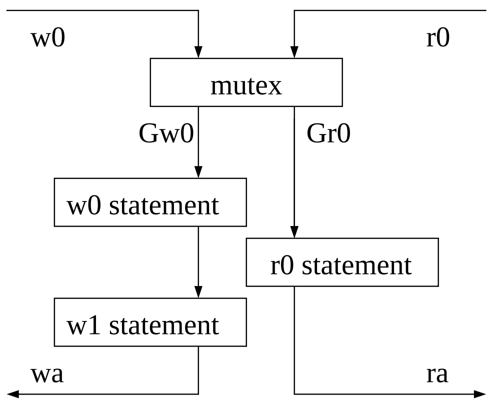


Figure 3.36 The block diagram of the function.

This only gives a solution to deal with the critical section. The following will give the details of the implementation for all statements.

In this mechanism, Statement *w1* is a special one, in which the control variable *n* is assigned a value different from the current values of both *l* and *r*. In practice, the following method can be used to deal with this efficiently:

```
differ = ((2,3,2), (3,3,1), (2,1,1));
… …
n := differ [l, r];

… …
```

The above method can be presented by using a table shown in Table 3.5.

Table 3.5 A table of differ

n		*r* =		
		1	2	3
l =	1	2	3	2
	2	3	3	1
	3	2	1	1

For instance, when *l* = 3 and *r* = 2, *n* is assigned a value 1.

The circuit implementing the *w1* statement (the differ and *n* Reg block) is shown in Figure 3.37, in which the differ and the three state latch circuits are shown in Figure 3.38 and Figure 3.39 respectively. All gates involved are simple monotonic with no more than four inputs. The differ circuit is SI because of "one-hot" encoding.

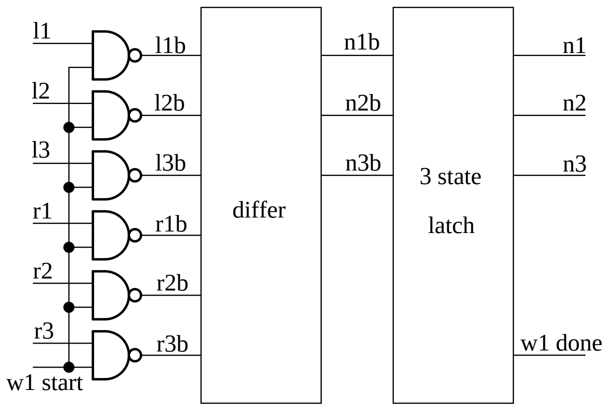


Figure 3.37 The circuit of the w1 statement.

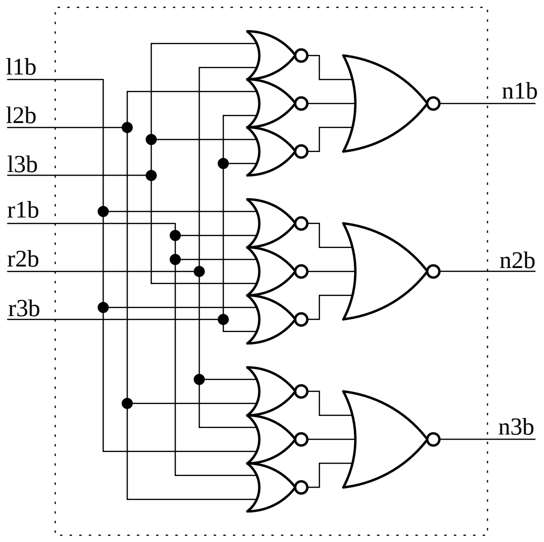


Figure 3.38 The differ circuit.

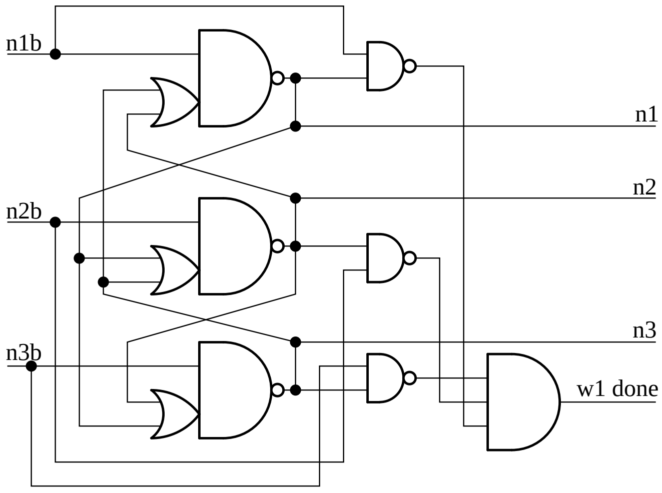


Figure 3.39 The three state latch.

The statements *w0* and *r0* are assignment statements. The registers for control variables *l* and *r* employ the three-state latch shown in Figure 3.39. We use the following circuit to implement them. The general arrangement is shown in Figure 3.40.

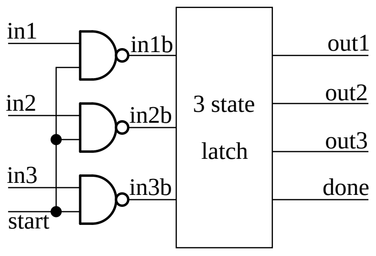
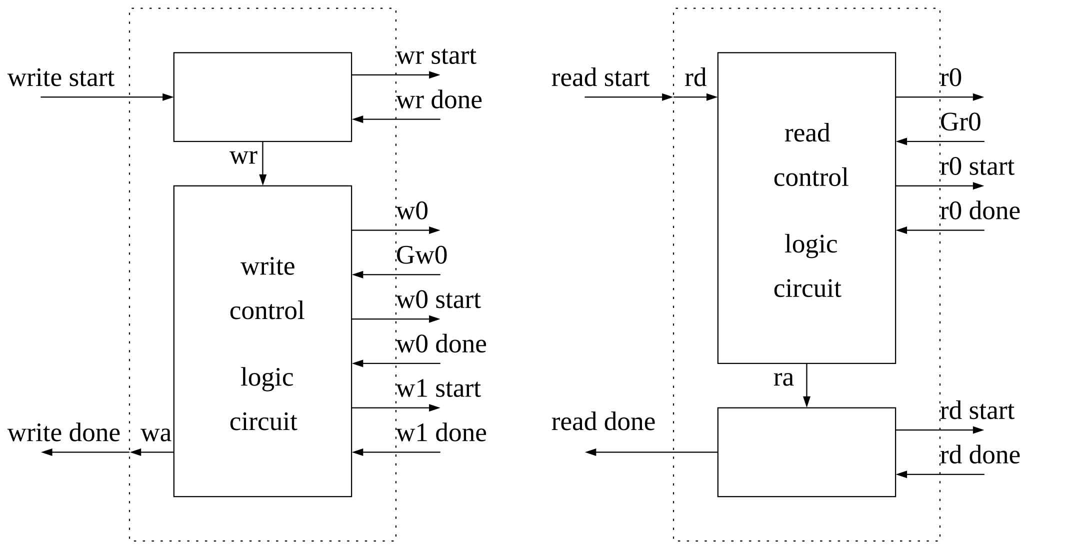


Figure 3.40 The circuits for the l and r registers.

3.4.3.2.3 Implementation of the control circuit

In this three-slot implementation, the control circuits are designed to be as concurrent as possible in order to increase speed, instead of being maximally sequential.

By employing a Mutex element carefully in this design, both the relative atomicity of crucial statements and potential metastability can be resolved simultaneously.



a) The block diagram of the write control circuit b) The block diagram of the read control circuit

Figure 3.41 The control circuits.

The statement circuits are entirely self-timed and SI, with some parallel arrangements of signals managed by a series of handshake protocols instead of via FM assumptions. Based on Figure 3.34, the schematic write and read statement control circuits are given in Figure 3.41. The signal sequences of the write and read statement control circuits specified in STG form are shown in Figure 3.42 and Figure 3.43. The two control circuits are joined at the Mutex.

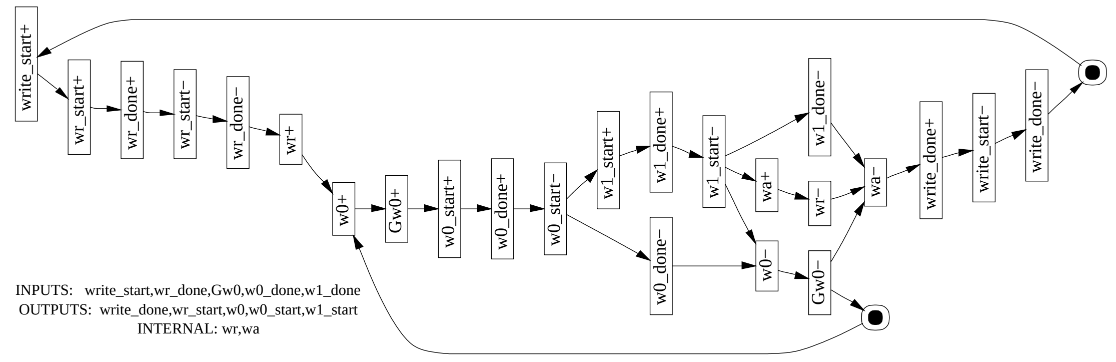


Figure 3.42 The STG of the writer.

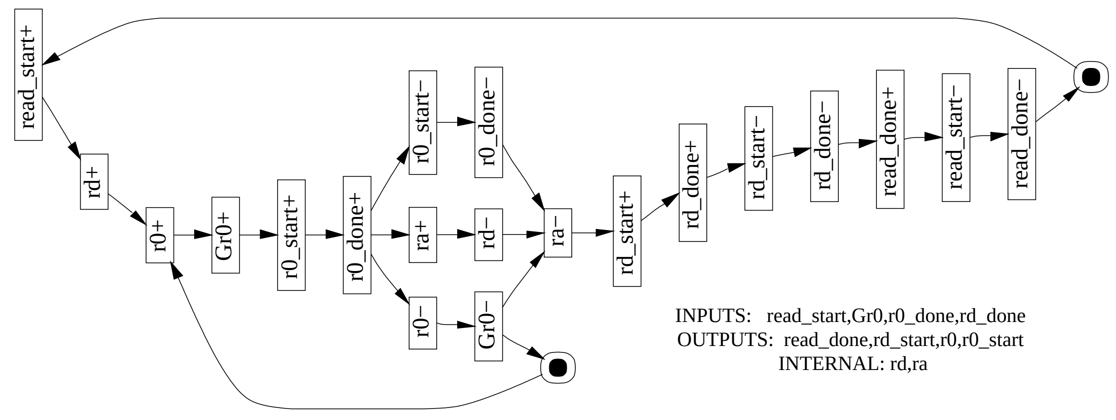


Figure 3.43 The STG of the reader.

In Figure 3.41, only one Mutex with a metastability resolver is used, compared with two Mutexes with resolvers used in the four-slot implementation. The functions of the Mutex and resolver have been introduced in Chapter 2. With the Mutex, the *r0* statement is atomic relative to the *w0* and *w1* statements.

In this interface there are three pairs of handshake signals in the writer part. They are *wr* and *wa*, *w0 start* and *w0 done*, *w1 start* and *w1 done* respectively. The other interface signals, *w0* and *Gw0*, connect to the Mutex, one for the request and the other for the grant. Figure 3.44 shows the simplified STG, derived from Figure 3.34 and Figure 3.42, for the writer statement control with only the signals directly relating to the statements being shown. Some parallelism is introduced in this circuit, such as after the falling edge of *w0 start*; while we wait for the falling edge of *w0 done*, the *w1 start* signal can change. In addition, after the falling edge of *w1 start*, the falling edge of *w1 done*, the raising edge of *wa* and the falling edge of *w0* can happen at the same time. We wait for the completion of all these signals at the falling edge of *wa*. This means that before the falling edge of *wa* is triggered, the other operations should be finished.

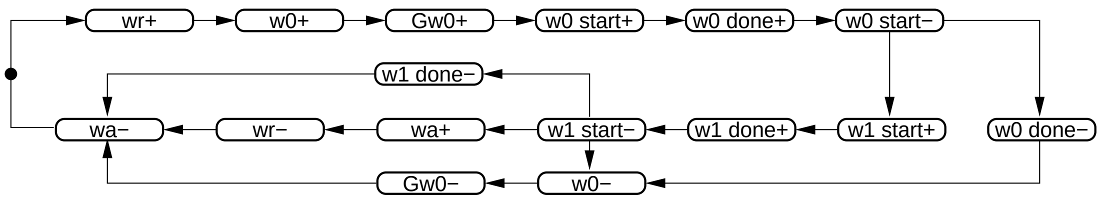


Figure 3.44 The STG form of the w0 and w1 control circuit.

The STG specifications were fed to the Petrify tool. Here only the *w0*, *w1*, *r0* statements, as well as the request signals to the Mutex, were considered. The implementation of the data access statements are included here only as an illustration of how these can be done. Effort is concentrated on the control variable statements here because data slot circuits must be tailored to particular applications and should not, therefore, be included in any generalized study.

The equations for the control circuits of the write and read parts produced by the Petrify tool are shown in Table 3.6.

Table 3.6 The equations of the reader and writer from the Petrify tool.

[r0] = rd * csc0;	[w0] = wr * csc2 + w0 done + w1 start;
[r0 start] = Gr0 * csc0;	[wa] = w1 start’ * csc2’;
[ra] = csc0’;	[w0 start] = Gw0 * csc1;
[csc0] = r0 done’ * (Gr0’ * rd’ + csc0).	[w1 start] = csc2 * csc1’ * w0 start;
	[csc1] = w0 done’ * csc1 + Gw0’;
	[csc2] = w1 done’ * (wr’ * csc1 + csc2).

In order to maximize technology independence with respect to preserving speed independence, simple monotonic gates are used to assemble the circuits implementing these equations, rather than the complex gate solutions normally assumed by the Petrify tool. This leaves the question of whether the final circuits are SI open to debate, which will be dealt with later. For the moment, the main parts of these statement control circuits are shown in Figure 3.45 and Figure 3.46. The writer statement control circuits are represented informally below. The reader statement control circuits can be derived following similar reasoning.

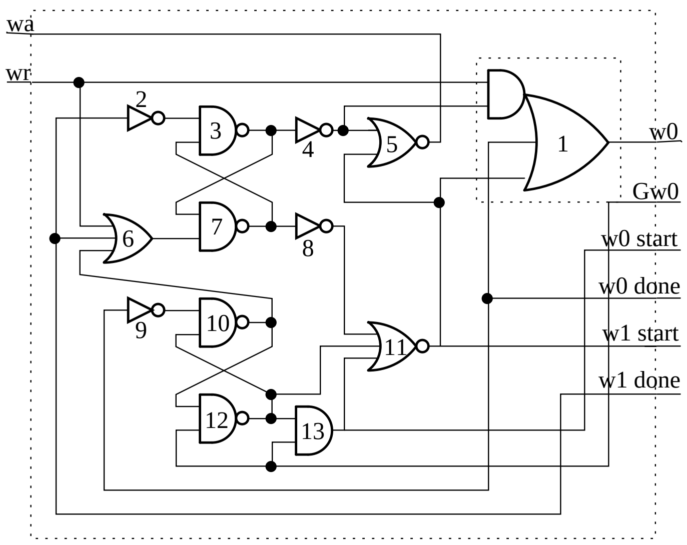


Figure 3.45 The control circuit of the w0 and w1 statements.

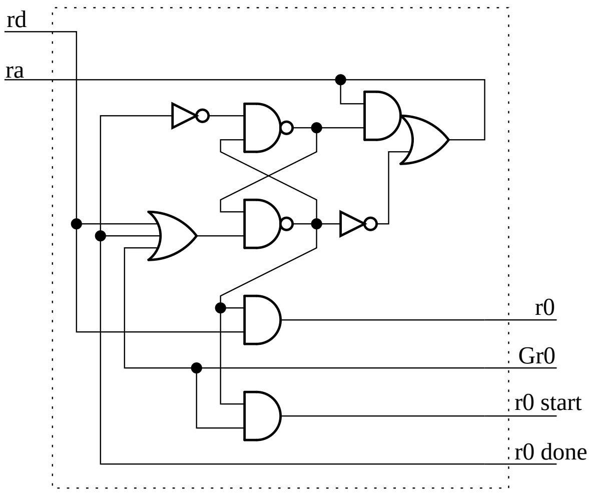


Figure 3.46 The control circuit of the r0 statement.

The writer interface connection with the environment is shown in Figure 3.41. The input signals are *wr*, *Gw0*, *w0 done* and *w1 done*. The output signals are *wa*, *w0*, *w0 start*, and *w1 start*.

In Figure 3.45, cross-coupled gates 3 and 7 form an SR latch. This "*csc2*" latch is set via gate 6 when *w1 done, wr* and *csc1b* are all low. It is reset via gate 2 when *w1 done* is high. Cross-coupled gates 10 and 12 form another SR latch. This "*csc1*" latch is set when *Gw0* is low, and reset via gate 9 when *w0 done* is high. The initial values of these latches are both set to logic one. Gate 1 is used to generate the signal *w0*. Gate 5 is used for the signal *wa*, gate 11 for the signal *w1 start* and gate 12 for the signal *w1 done*. There are also some auxiliary logic circuits.

In Figure 3.45, gates 4 and 8 are used to guarantee that the circuits are SI. Without these gates we have a circuit that may only be considered SI under certain assumptions.

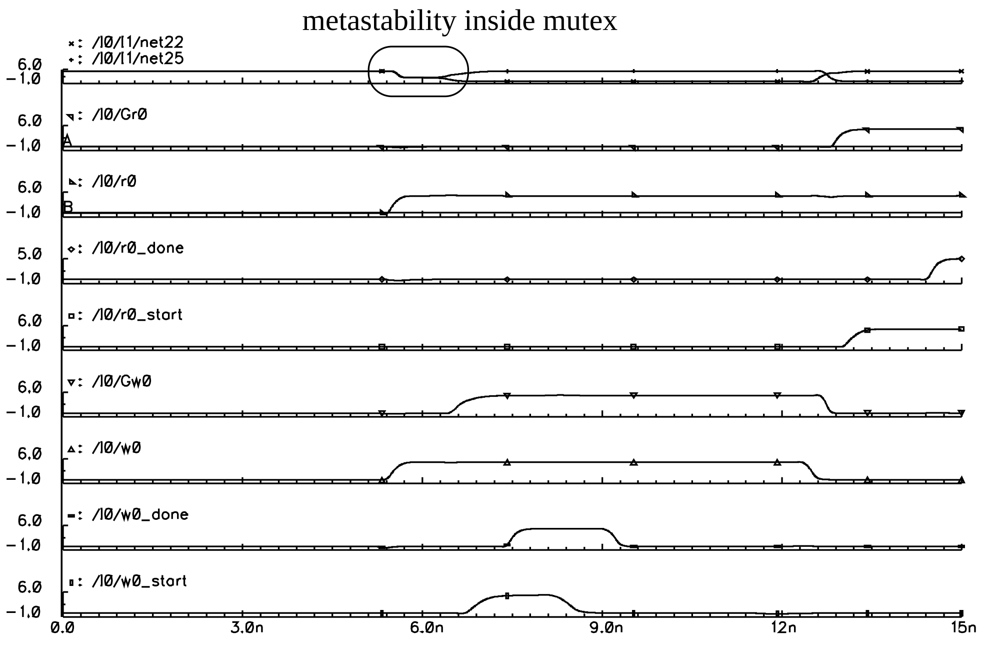


Figure 3.47 Analogue simulation waveforms with metastability within the Mutex.

The three-slot *Pool* design has been built using the VLSI design tool Cadence (0.6 micron technology). All simple gates in which there are no more than four input pins are from standard libraries.

3.4.3.3 Circuit analysis

Simulations have shown that the functional behaviour of the circuits is as expected, and any metastability does not propagate throughout the system but is contained within the Mutex (net22 and net25). The analogue simulation result is shown in Figure 3.47.

3.4.3.4 Comparison

Self-timed *Pool*s based on four-slot and three-slot algorithms have been designed and implemented in VLSI circuits. The illustrative data path implementations are the same in both cases. The crucial statement control circuits are different. The main differences are shown in Table 3.7.

Table 3.7 The differences between four-slot and three-slot *Pool*s.

	four-slot	three-slot
circuits	building block	monotonic simple gate
Mutex	2	1
control logic	Sequence	as parallel as possible
aim	Safety	faster and safety

In the four-slot implementation, in order to retain an element of regularity, circuits known to be completely SI are chosen as the building blocks with which to assemble the control circuits. In the three-slot implementation, all control circuits are designed directly by using the Petrify tool with maximum parallelism.

The physical size of the final implementation of the three-slot *Pool* should be smaller than that of the four-slot *Pool*, especially considering that there is one fewer data storage area to incorporate.

The comparative merits on data coherence and other ACM properties are discussed in [Xia 1999b].

The three-slot *Pool* is faster than the four-slot one, by virtue of having fewer and simpler control variable statements. For the implementations the speed difference is increased because the three-slot one has been designed with maximum parallelism.

Analogue simulations have been run for the three-slot and four-slot *Pool* implementations. The results are shown in Table 3.8.

Table 3.8 The comparison result.

	four-slot min time	three-slot min time
w0 + w1	9.9ns	4.2ns
r0 (+ r1)	3.5ns	1.4ns

3.4.3.5 Verification

Although the equations produced by the Petrify tool are for SI circuit solutions, whether or not the implementation is SI depends on how the equations are realized, especially when complex gates specified by the Petrify solution are replaced by simple gates. The Versify [Versify] tool has been used to verify whether the final decomposed circuits are SI. The Petrify tool assumes that inverters have zero delay and guarantees that the circuits constructed using complex gates and zero-delay inverters are SI. However it is easy to prove that certain circuits which are taken as SI solutions by the Petrify tool are not SI if input inverters do not have zero-delay, as has been found by running the versify tool on them.

Using the Versify tool, an asynchronous circuit can be verified by comparing it with the STG specification. The Versify tool takes the circuit specification in BLIF format [Sentovich 1992] comprising the gates and latches from the “versify.genlib” library and an optional user library.

The STG files and the BLIF files were compared through the Versify tool. The following results have been obtained.

The versify report generated is: for the read statement control circuits, “the number of Traversing net Reachable states is 34”; “the number of Traversing circuit Reachable states is 71”; “the circuit implements its specification”; “the circuit has no deadlock and no livelock”; and “the circuit is speed independent”.

The versify report generated is: for the write statement control circuits, “the number of Traversing net Reachable states is 46”; “the number of Traversing circuit Reachable states is 197”; “the circuit implements its specification”; “the circuit has no deadlock and no livelock”; and “the circuit is speed independent”.

3.5 Self-timed *Signal* specification and implementation

A *Pool* type of ACM is not suitable if, for instance, the reader process needs to continue only when there is new data available. This may be significant if the reader process is implemented with an asynchronous device where power savings can be realised by allowing it to wait whenever it can, or if the data being transmitted are of the interrupt and exception type.

For such applications, the other kinds of ACMs are proposed (see Table 3.1), such as the *Signal* type ACM which is more suitable for the above example. The basic *Signal* protocol does not hold up the writer but does hold up the reader if the ACM is empty. The difference between *Pool* and *Signal* is shown in Figure 3.48.

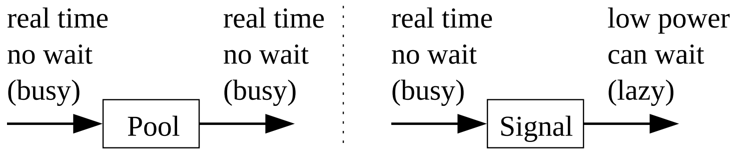


Figure 3.48 Comparing the applicabilities of the *Pool* and *Signal*.

The basic *Signal* protocol specified an ACM with capacity 1, re-reading not permitted and overwriting permitted. In other words, writing can happen when the *Signal* contains either 0 or 1 item of unread data. A write data access modifies that data state of the *Signal* to 1, but reading can only happen when there is one item of unread data in the *Signal*.

This is a special case of the general definition of the *Signal* with capacity n, $n \geq 0$, of which the "overwriting buffer" schemes found in many places in the literature (such as [Sgroi 2000]) are also special cases.

This basic *Signal* definition is captured by the PN model in Figure 3.49. Here the place 0 and place 1 are complementary to each other, one and only one being marked at all times.

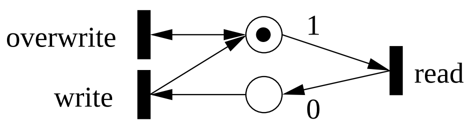


Figure 3.49 Basic definition of the *Signal* protocol.

This definition treats the write and read data accesses as atomic processes, which is not sufficiently clear for system synthesis and implementation. In reality, data accesses by the reader and writer must take time, and the timing relationship between the reader and writer processes is important for an ACM in a real time system.

In other words, apart from the timing requirements imposed on the reader and writer by the data state of the ACM, data access at one side may affect the temporal behaviour of the other side. For example, the reader may or may not be required to wait while the writer is in the middle of an access because of the implementation. The model in Figure 3.49 is not specific about such distinctions.

In order to represent the concept of non-atomicity of data accesses by the writer and reader, such accesses must be represented as distinctive states in the model. This is achieved by using the techniques introduced in [Clark 1998]. Such a refinement is shown in Figure 3.50. By treating the read and write data accesses asymmetrically, this definition maintains the possibility of full temporal independence for the writer, but prescribes waiting for the reader while either writing or overwriting is in progress. It also means that the reader will always obtain the newest item of data from the writer available at the time of reading and the writer is always allowed to access the *Signal*, regardless of the data state of the *Signal* state of the reader. This definition is used in this section.

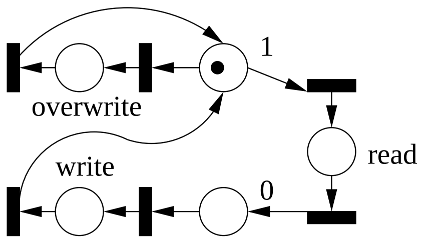


Figure 3.50 *Signal* with non-atomic writing.

Having only one token in the models implies that the writer and reader are dealing with complete items of data one at a time and that the integrity of these items of data is maintained, i.e. the reader is not supposed to obtain an item of data that is assembled from parts of different items provided by the writer, or otherwise corrupted.

Formally, the definition in Figure 3.50 specifies the following properties:

1. **Data states and their updating**: The *Signal* has a data capacity of 1. In other words, at any time, it contains either 0 or 1 items of unread data. At the start of a read data access, the *Signal*'s data state is set to 0 (empty). At the end of a write data access, the *Signal*'s data state is set to 1, the item of data provided by this write data access being unread.
2. **Conditional asynchrony for the reader**: A read data access may start only when the data state of the *Signal* is 1 and no write data access is occurring. A read data access can be arbitrarily long.
3. **Unconditional asynchrony for the writer**: The writer must be allowed to start and complete a data access at any time, regardless of the data state of the *Signal* and the status of the reader.
4. **Data coherence**: The *Signal* and the data accesses of the writer and reader processes must not modify the content of any item of data. In other words, any item of data received by the reader must not have been changed since being provided by the writer.

5. **Data freshness**: Any read data access must obtain the data item designated as the current unread item in the *Signal*, i.e. the item of data made available by the latest completed writer access.

In the terminology of multi-slot ACMs [Simpson 1990], a "data slot" is a unique portion of the shared memory which may contain one item of data. It is obvious that a *Signal* in the form of Figure 3.50 cannot be implemented with only one data slot, since it cannot possibly support writing and reading at the same time and maintain data coherence. In other words, properties 3 and 4 cannot both be satisfied by an implementation with only one slot.

Previous work has indicated that it is desirable to minimise the number of slots in multi-slot ACM implementations (four-slot and three-slot *Pool*s described above). The advantages include smaller hardware expenditure both in the actual slot memory areas and control circuits, leading to better temporal performance and higher reliability.

Most of the software solutions in the literature also spend considerable effort on the reduction of the number of slots needed for any particular ACM specification, with similar reasons.

3.5.1 State graph specification of *Signal*

The conceptual definition of *Signal* in Figure 3.50 cannot be used as a formal specification of the *Signal* protocol because it does not show the specifics of this protocol, such as how, for example, blocking on writing is avoided by using multiple slots. In this sub-section we construct a state graph specification for a two-slot *Signal*, which will define a maximally permissible automaton satisfying the required properties of the ACM. Let us first formulate those properties using the idea of states and transitions labelled by write and read actions. The reason for using a state graph for specifying the ACM protocol instead of trying to construct the PN model directly is explained as follows. State graph modelling is much clearer for reasoning about global system properties than PNs, because a state graph is based on the concept of global states and interleaving semantics. A PN would already be a decomposition of the system's states into local states (places) and this is often non-trivial when systems consist of a number of processes (write and read) and components (slots).

- Actions: Actions are processes whose start and completion are atomic events and whose durations are finite but non-atomic.
- States: A state is the result of the completion of one or more actions. During a state, actions may exist which have started, but not completed (in process), and actions may exist which may start. These "in process" and "may start" actions cannot complete, however, without resulting in a new state.
- Previous sets: The previous set of a state *s*, denoted as P_s, is the set of actions that lead to *s*.
- Next sets: the next set of a state *s*, denoted as N_s, is the combined set of actions which may start during *s* and actions which may be in process during *s*.

Here, the slots will be known as slot 0 and slot 1; write data access to slot *i* is known as wr*i* where *i* = 0, 1; read data access to slot *j* is known as rd*j* where *j* = 0, 1. These are the only actions considered at the moment. So during each write cycle the writer performs one action wr*i* for some *i* and during each read cycle the reader performs one action rd*j* for some *j*. With these assumptions, properties outlined in the previous section require the following conditions:

1. Data states and their updating, and asynchrony for reader and writer:

   wr$i \notin P_s \Rightarrow$ rd$j \notin N_s$, $\forall$ *s*, *i* and *j*; this means that if a state was not the result of a completion of a write data access (i.e., it is solely the result of the completion of a read data access), then a new read data access cannot start during it (data state = 0).

   wr$i \notin N_s$, $\forall$ *s*; this means that a write data access must be allowed to start or be in process during any state.

2. Data coherence:

$\neg(\text{wr}i \in N_s \ \& \ \text{rd}j \in N_s) \ \forall \ s, i, j$ and $i = j$; this means that there can be no simultaneous read and write access of any slot.

3. Data freshness:

   $\text{wr}i \in P_s \ \& \ \text{rd}j \in P_s \Rightarrow \text{wr}j \in N_s \ \& \ \text{rd}i \in N_s, \ \forall \ s$ and $i \neq j$; this "slot swapping" fully utilises the two available slots so that the reader always obtains the item of data provided by the latest completed write data access.

4. No "retry loops":

   $\text{rd}j \in N_{s_m} \ \& \ \text{rd}j \notin P_{s_k} \Rightarrow \text{rd}i \notin N_s, \ \forall \ s_m$ preceding s, $i \neq j$, and for all s_k on the state trajectory between s_m and s, including s_m; this is also true for write data accesses.

Condition 4 reflects the desire to avoid "retry loops" [Chen 1998a], so as to keep the solution simple for the first attempt. It means that, once the reader (or the writer) has been allocated a slot for access, it must perform this access before it can be allocated the other slot for access.

A simple state graph specification has been obtained using these conditions. It conforms to the definition in Figure 3.50 and is shown in Figure 3.51. The initial state s1 is labelled with a big arrow. In this state, the ACM starts as "empty" with only writing to slot 1 enabled.

In Figure 3.51, the dotted edge denotes that the two states at its ends are essentially the same state and the state graph is in closed form for readability.

All conditions mentioned above have been incorporated into the specification. For instance, at state s2 in Figure 3.51, rd1 cannot be in the next set because rd0 was in the next set at state s0 and has not featured in the next set of any state on the trajectory from s0 to s2 (condition 4). At state s1, no reader slot access is in the next set because there is no writer action in its previous set (condition1).

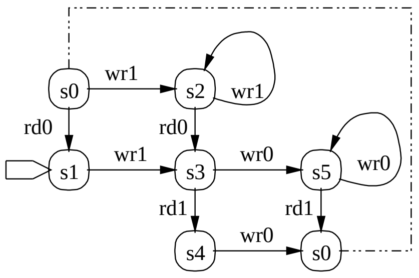


Figure 3.51 Simple state graph specification for a two-slot *Signal* (s0 is initial state).

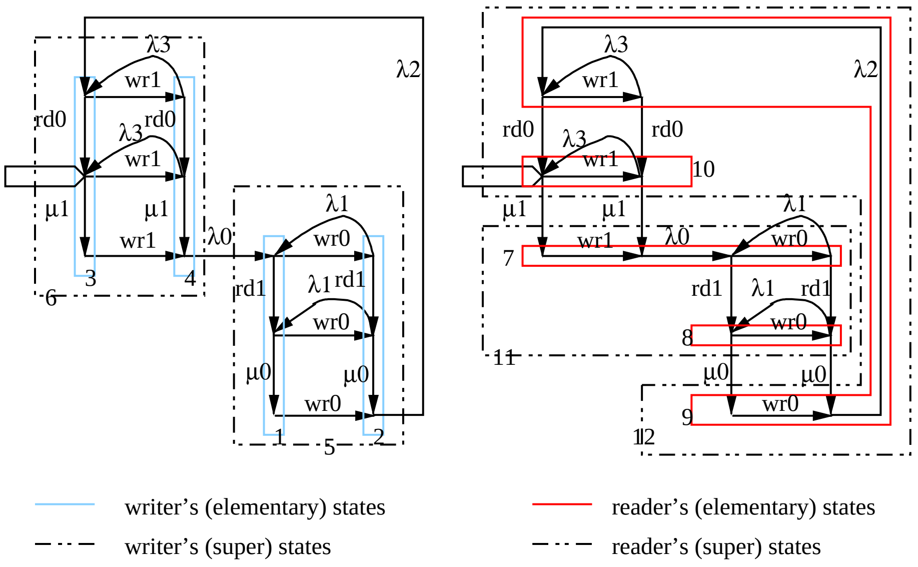


Figure 3.52 "Distributing" states between reader and writer using regions.

This specification is not detailed enough for implementation, because the actual means which maintain the slot steering for the writer and the reader processes remain undefined. A further refinement is obtained by adding "silent actions", which perform the necessary functions of the slot-management control; variable statements used in published multi-slot ACMs [Simpson 1990, Tromp 1989, Kirosis 1987, Chen 1998a and the above four-slot and three-slot *Pool*s]. The result of this refinement is shown in Figure 3.52. This refinement will also be needed in order to satisfy the separation conditions for PN synthesis, described in the following section.

Both halves of Figure 3.52 show the same state graph. The reason for duplication is simply to avoid cluttering when deriving regions corresponding to the actions related to write and read parts (cf. next section). In Figure 3.52, the λ's denote silent actions performed in the write part that separate the states with the same connections to the main actions of data slot accessing. For instance, $\lambda 0$ is the control action that separates the *Signal* for giving next assess to slot 1 to the reader and next access to slot 0 to the writer. Before $\lambda 0$ the writer was allocated slot 1 and the reader was allocated slot 0. The μ's denote silent actions performed in the read part. For example, $\mu 1$ stands for the request of the reader to start reading slot 1, and also the fact that slot 0 is no longer used for reading. This should indicate to the writer that it should move to slot 0 if it has finished or when it finishes with slot 1. Hence, depending on whether action $\mu 1$ has or has not been performed by the reader, the writer decides whether to do $\lambda 0$ (move to writing slot 0) or to do $\lambda 3$ (keep writing slot 1).

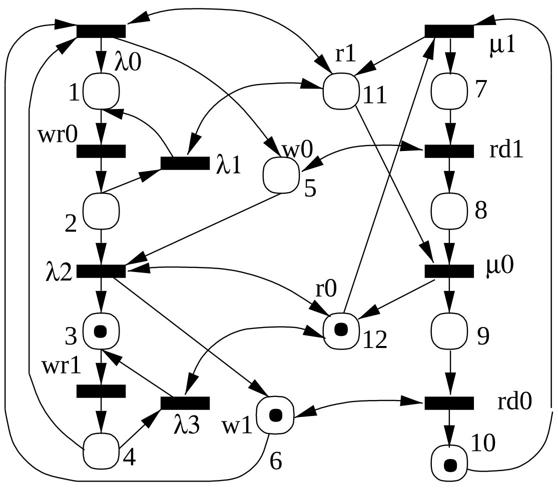


Figure 3.53 PN specification of the two-slot *Signal*.

3.5.2 Circuit synthesis

After PNs synthesis from the state graph specification, a PN specification of the two-slot *Signal* is obtained, which is shown in Figure 3.53.

More details can be found in [Yakovlev 2001].

That the PN already captures the notion of a “decomposed state” in its places is extremely important because we can exploit this distribution in the “net-to-circuit” translation.

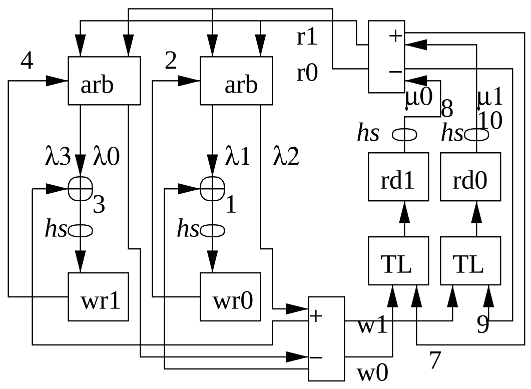


Figure 3.54 Block diagram of first circuit design.

Our first “sketch” of the *Signal*’s circuit implementation, which is basically a homomorphic translation of the PN in Figure 3.53, is shown in Figure 3.54. The circuit is built with two-phase (event-based) signalling in mind (see [Sutherland 1989]). It consists of the control skeleton part and the operational part, involving the write and read operations and the latches that implement variables *w* and *r*. These latches are equipped with pairs of control signals for set and reset handshakes, labelled “+” and “-” (note that these handshakes ensure that both the setting and resetting of *w* and *r* are properly acknowledged). The latches also produce “dual-rail” signals, *r0* (when *r*=0) and *r1* (when *r*=1) for *r*, and similarly for *w*. The write control part (see labelling of wires by numbers 1-4) is obtained by simply associating places 1 and 3, whose input transitions fire in a mutually exclusive manner, with XOR gates, and places 2 and 4 with requests to two ‘arb’ blocks, which are arbiters for sampling the (potentially changing) levels on signals *r0* and *r1*, and depending on the ‘*r0*=true’ or ‘*r1*=true’ state of the test, generating one of the event-based outputs corresponding to the appropriate λ transition. These signals then either activate the appropriate write operation, either *wr0* and *wr1*, depending on which of the two slots is supposed to be written, or send a request to toggle (set or reset) the state of *w*. In the read part, the control flow is very simple (note wire labels 7-11 to indicate correspondence with appropriate places in the PN) and it does not need arbitration. It waits, at one of the event-based DL gates (initially, it is the left one, associated with the initial position of the token in the read part at *r*=1) the arrival of the condition *w0* to be at 1 and then activates the *rd1* operation, followed by the resetting of the *r* variable. After that it performs a similar action with *r0* as soon as the *w* signal becomes 1.

This implementation is very schematic. In order to build the complete circuit for the *Signal* from it, it needs to be refined by providing interconnections with the two environment handshakes, write's request and acknowledgement and read's request and acknowledgement, which are implicit in this circuit. These two handshakes could be created by breaking the wires that are indicated in Figure 3.53 by the ovals labelled with *hs*. Of course, there have to be suitably multiplexed by using known two-phase elements, such as e.g. CALL from [Sutherland 1989].

We have also studied, at greater length (believing that this will give us a faster implementation), another circuit translation of the PN model, the one based on four-phase signalling. This translation method, described in [Yakovlev 1998], is based on the idea of a "one-hot-encoded" implementation of the PN model, in which places of the PN are associated with memory elements (called DCs, cf. Chapter 2) of the control circuit.

3.5.3 Hardware implementation

The structural idea of the circuit implementation for *Signal* is shown in Figure 3.55.

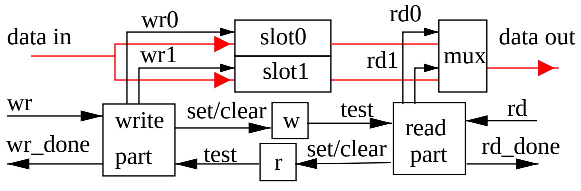


Figure 3.55 Basic diagram of *Signal* circuit implementation.

3.5.3.1 Implementation of the datapath and control variables

The slot part is implemented similarly to ones in the four-slot and three-slot *Pool*s. We will not discuss it further. In this mechanism, two control variables *w* and *r* are used. They are set by one side and reset by the other. The circuit implemented this function is shown in Figure 3.56.

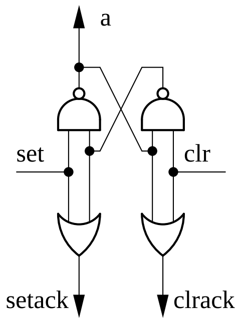


Figure 3.56 The set/reset circuit with completion logic.

In this circuit, completion logic is needed from the self-timed circuit point of view. Because set and reset actions cannot happen at the same time according to the mechanism, this simple circuit is sufficient for the purpose.

In systems with true timing heterogeneity, metastability in the ‘sync’ arbiters is unavoidable if both sides of an ACM are permitted unlimited access. In our implementation, metastability is contained within the ‘sync’ arbiter block, which is shown in Figure 3.57. Here *ck0* is the sample signal and *rbar* is the sampled signal. Only when *ck0* is active, sync can work.

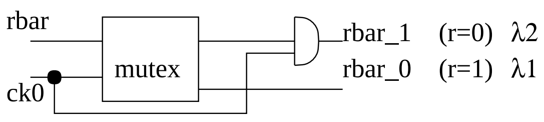


Figure 3.57 Implementation of a 4-phase 'sync'.

In this mechanism, for example, after sampling *rbar*; if *rbar* equals 0 (*r*=1), *write_ack* is set directly; if *rbar* equals 1 (*r*=0), *write_ack* is set after resetting *w* (see Figure 3.58).

3.5.3.2 Self-timed implementation of control circuit

The read part is conceptually simpler than the write part and its description is left out. The write part, whose PN specification can be traced back to Figure 3.53, is shown in Figure 3.58. This synthesis process from PNs to hardware will be shown in Chapter 6 as a case study. The research topic will be explained in Chapter 5. Here we only explain it from the optimized resulting circuit. This circuit consists of a set of DCs (shown in bold) to store the distributed state of the control and blocks representing the control logic. The controlled (operational) logic is simply inserted between the cells, by breaking the wire that signals the next call about the arrival of the token. Note that the environment itself is inserted between cells (as handshake "*done/wr*"). We first consider the DC part.

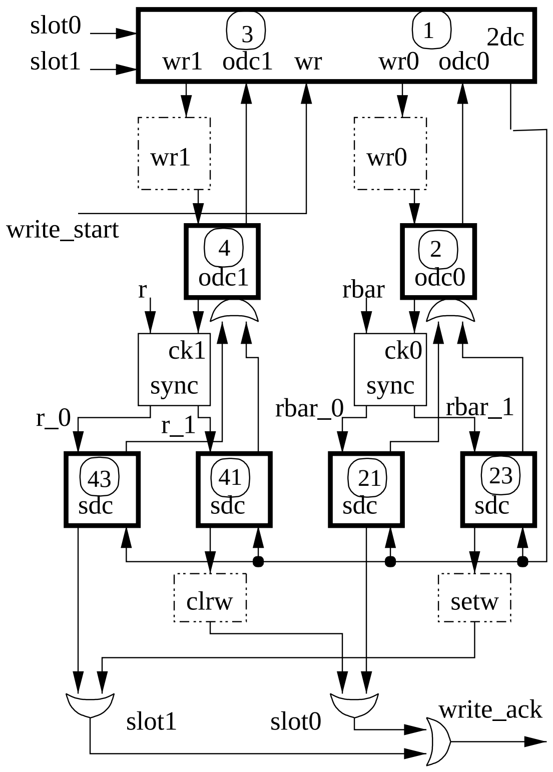


Figure 3.58 Circuit implementation of the write part using DCs.

The cells, built essentially around SR flip-flops (cf. Chapter 2), represent the marking of the corresponding places in the PN (the absence of a token in a place is associated with state 01 in the flip-flop, the presence of a token with state 10). The labels of these places are shown in circles. In particular, blocks odc0 and odc1 stand for the cells corresponding to places 2 and 4. Block 2dc encapsulates two cells that model places 1 and 3 (the internal logic structure of this block is shown in Figure 3.59). The two pairs of sdc cells, labelled with 43 and 41, 21 and 23 respectively, are added in order to implement the appropriate branching of a token from place 4 either to place 3 or to 1 (similarly, for a token from place 2 to either place 1 or 3) depending on the arbitration decision made in the 'sync' block, which samples the value of the output *r* (rbar) of the binary variable 'read'. Such a sampling corresponds to testing the marking of places 11 and 12 (by read arcs) using transitions $\lambda 3$ and $\lambda 0$.

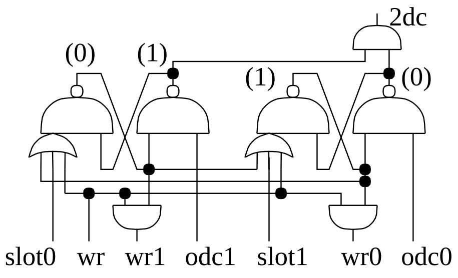


Figure 3.59 Logic for 2dc block.

The cells named sdc are all built using a simple logic structure; they model the 'linear' pass of the control token. The cells named odc0 and odc1 are slightly more complex and allow merge of acknowledgements from two mutually exclusive branches of token flow (from place 4 and 2). For example, cell odc1 (for place 4) can either be reset to 0 after the pushing of the token back to place 3 (via the sdc cell 43) or to place 1 (via sdc cell 41), the latter also involving the execution of *setw* (event $\lambda 2$ in the net). This is shown schematically by depicting images of the OR gate at the resetting input of cell odc1, which collects *ack* signals from the sdc's 43 and 41. Implementing such a more complex reset function requires the use of 3-input NANDs in place of the gate which generates *xb* in Figure 3.60.

The operation of the control logic based on DCs can be visualised by performing a sequence of transitions on the cells signals as shown in Figure 3.60. The arrival of the token corresponds to the left-hand side handshake request signal (*inr*) going to 0.

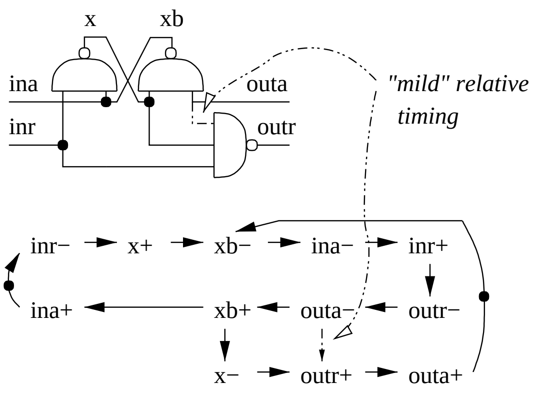


Figure 3.60 Simple DC (with extra wire for a 'mild' relative timing assumption).

A similar method can be used to construct the control circuit for the reader part.

3.5.3.3 RT implementation of the control circuit

The dotted connection (from *outa*) in all DCs should be disregarded when the circuit is built maximally SI. However, adding such a connection would introduce a relatively ‘mild’ delay dependency (cf. relative timing [Stevens 1999]), concerned with the fact that the return of *x* back to its quiescent state (0) is left unacknowledged. The gain from this is that transitions xb+, x- in the above sequence are executed concurrently with the forward propagation of the token. Under realistic delays in the gates, even with a zero delay for the controlled operations between the cells, there is sample time to complete these transitions (reset of the token) before the ‘front’ of the new token comes back through the control loop.

A bit more ‘aggressive’ relative timing is applied when the ‘front’ of the token is allowed to propagate forward as soon as it has been recorded in the cell, without even waiting for the completion of the preceding handshake. Again, under realistic delays, and assuming that the operational part takes at least a couple of inverter delays, this should be sufficiently robust. The modified logic of the case of a simple cell (linear transfer) is shown in Figure 3.61. Here the delay of passing the token through the cell is absolutely minimal – it takes two inversions, *x* and *xb*, from inr- to outr-.

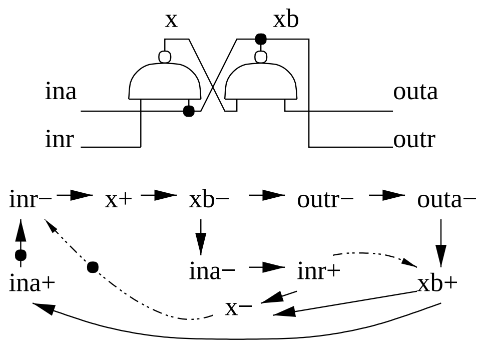


Figure 3.61 Simple DC with 'aggressive' relative timing.

3.5.3.4 Circuit simulation

Extensive analogue simulations (for 0.6 μm CMOS technology) have been conducted on two circuit implementations, one maximally SI using DCs and the other with aggressive relative timing (RT) [Stevens 1999] using the circuit in Figure 3.61. Both implementations have been confirmed qualitatively to satisfy their specifications. Their relevant full (write, read) cycle times are given in Table 3.9. These times are measured between the adjacent rising edges of the write requests (write start+ $\rightarrow$ write start+) and read requests (read start+ $\rightarrow$ read start+). For writes the two modes considered are those that do SR and do not involve NSR switching (setting or resetting) the value of flag of w. For reads, which can be naturally blocked by the absence of new data, only one mode, called “no waiting” for new data (NW), is considered.

Table 3.9 Cycle times

type	Write		read
	NSR	SR	NW
SI	9.0ns	10.4ns	9.0ns
RT	4.8ns	6.3ns	6.6ns

In our implementation, metastability is contained within the ‘sync’ arbiter block (Figure 3.57) and analogue simulations (Figure 3.62) confirm that it does not propagate through the system. Note that, after the write request is set to high and *ck0* is generated, in order to sample the current value of variable *r*, the latter is marking its transition from high to low. This puts the SR flip-flop (we use a standard Mutex implementation due to Seitz [Seitz 1980]) into a metastable state (see the upper window in Figure 3.62), which is eventually resolved in favour of *rbar_0*, i.e., the old value of *r* (high). The outputs of the Mutex, *rbar_0* and *rbar_1*, produce clean edges. These signals are then used to generate *ack* signals leading to the control logic, which in this case passes control to one of the DCs, labelled 21 and 23, to perform either nothing or ‘*setw*’ (setting **w** to 1), followed by the rise of the write *ack* signal.

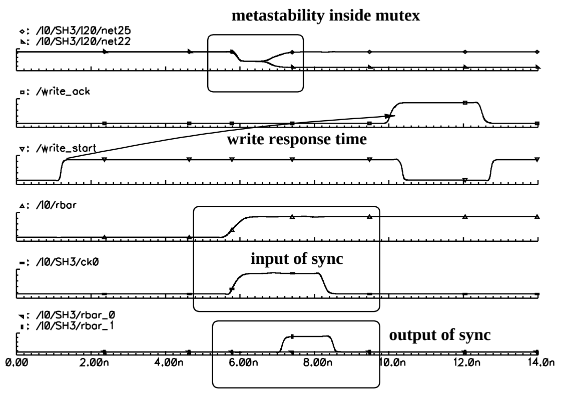


Figure 3.62 Illustrating metastability.

3.6 *Message*, the dual of *Signal*

The new ACM classification in Table 3.1 makes it possible to define a more useful dual for the *Signal*. This is the new *Message* type which permits re-reading but not overwriting.

The *Message* therefore does not hold up the reader, but does hold up the writer when the data state is equal to the ACM’s capacity (all data items in the ACM unread). Such an ACM is useful when loss of data items is not permitted and when the reader process must be given temporal independence from the ACM’s data state. Roughly speaking, from the writer side, new *Message*s may not be generated if previous ones have not been received, as compared with *Signal*s being generated regardless of previous ones not being received. The reader, on the other hand, will not wait for a new *Message* to appear but will re-use the previous one if needed, and will stop and wait for a new *Signal* if one is not available.

Obviously, the *Message* is a simple mirror image of the *Signal*. The defining models in Figure 3.49 and Figure 3.50 can be simply reversed to form similar definitions for a 1-capacity *Message*, and it can be said that the implementation above is also that of a simple *Message* with full asynchronism for the reader. It just needs to be connected to the writer and reader processes the other way round from the *Signal*.

3.7 Conclusions

A new classification for ACMs was presented in this chapter. From the timing point of view, the new classification is more meaningful than the original one. In addition, all basic ACMs can be implemented in hardware, especially self-timed circuits.

Four-slot *Pool*, three-slot *Pool*, and two-slot *Signal* were implemented by using self-timed circuits, and two-slot *Message* was discussed in the end of this chapter, which is a dual of the two-slot *Signal*. As a result, after implementing the two-slot *Signal*, it is easy to implement this two-slot *Message*.

Both digital and analog simulations were done for the four-slot *Pool*, three-slot *Pool*, and two-slot *Signal* designs. The results show that they function as expected.

The ACM designs, within their local boundary, are not fully asynchronous (in the sense of non-blocking) by virtue of the unpredictable waiting introduced by Mutexes. However, no metastability propagates through the Mutexes, and it is settled down in a limited amount of time.

The difference between the FM solution (four-slot *Pool* by Simpson) and our self-timed solution (SI four-slot *Pool*) was investigated in this chapter. The performance of the worst case of the SI solution equals the one of the FM one. The SI solution has the potential to run as fast as possible, because metastability is such a rare event. In addition, metastability was shifted from the control variables to the handshake interfaces. The SI solutions are safer than the FM ones.

Especially, it is possible to make some modification in the SI solutions in order to obtain fast ones. Although they are not SI, they are safe under some timing assumptions. The example in this chapter is the RT implementation of the three-slot *Pool*.

In addition, several different design styles were used in the implementations.

Chapter 4: ACM testing

4.1 Introduction

As identified in the introduction, testing is a hot topic in both synchronous circuits and asynchronous circuits. Many testing techniques have been developed for synchronous circuits. However, the testing techniques used for synchronous circuits are not generally suitable for asynchronous circuits, although examples exist in [Kondratyev 2002]. This is because asynchronous circuits are event driven, and not timing driven as synchronous circuits. In addition, because of variety of timing in asynchronous circuits, it is very difficult to test this kind of circuit.

Although some techniques for asynchronous circuits have been presented which can be used to test normal asynchronous circuits, including self-timed circuits, they are not satisfactory for testing fully asynchronous communication circuits such as our self-timed ACM. The reason is discussed in this chapter.

With advent of SoC, communication between multiple independent systems built on one chip should be needed. Testing of this kind of system is worthwhile to be studied.

In this chapter, we introduce a method to test this kind of circuit, and develop a method to test our self-timed four-slot *Pool* type of ACM.

4.2 Asynchronous circuit testing methodology

Testing is an essential part of the overall realization of logic circuits, especially VLSI circuits. The main purpose of testing is to verify the correct operation of a circuit and detect the possible presence of certain processing faults [Shepherd 1996].

However, several aspects of asynchronous circuits make them harder to test than synchronous circuits. The reasons are as follows [Hulgaard 1994]:

1. Asynchronous circuits have by definition no global synchronization signals (clock). This drastically reduces the amount of control over the circuit as it cannot easily be "single stepped" through a sequence of states, which is a common way to test synchronous circuits;
2. Asynchronous circuits tend to have more state holding elements than synchronous circuits, so generating test vectors is harder, and design techniques to ease testing will have a higher area overhead;
3. Asynchronous circuits may have hazards or races when faulty, and these delay faults are notoriously difficult to detect.

Fortunately, the other aspects of asynchronous circuits tend to make them easier to test. Because asynchronous circuits use local handshakes instead of global clock signals to synchronize operations, a stuck-at fault on the signals used for handshaking will cause communicating modules to wait indefinitely, an effect that is easily observable. These differences lead to new approaches for testing asynchronous circuits or a re-evaluation of the trade-offs involved when applying techniques developed for testing synchronous circuits [Hulgaard 1994].

4.2.1 Terminology

A number of terms are used in testing:

1. **Primary outputs**: the outputs of the circuit under test are called *primary outputs*, and we assume that they can be easily observed.
2. **Primary inputs**: the inputs of the circuit under test are called *primary inputs*, and these are assumed to be easily controllable.
3. **Controllability**: the *controllability* of a circuit is the ability to establish a specific signal value at each node in the circuit by proper setting of the circuit's primary inputs.
4. **Observability**: *observability* of a circuit is the ability to determine the value at any node in the circuit by observing the primary outputs while controlling the primary inputs.
5. **Testability**: the *testability* of a circuit is a measure that attempts to reflect the ease with which a circuit can be tested. A circuit with high testability generally has a higher degree of observability and controllability than one with low testability.
6. **Fault detection**: *fault detection* is the process of determining whether a given circuit contains one or more faults. This is done by applying a sequence of input values (called test vectors) to the circuit and observing the primary outputs. If the outputs differ from the specification, a failure has occurred and a fault is present in the circuit.
7. **Fault coverage**: in a test, a set of test vectors are applied to the circuit in order to detect as many faults as possible. The effectiveness of a test is measured by the *fault coverage*, which is the ratio of faults potentially detected by the test to the total number of possible faults in the circuit.
8. **Fault model**: a *fault model* is employed to test a circuit efficiently based on its structure rather than on its functionality. The fault model is an abstraction of the physical faults we try to detect. The more detailed the fault model, the more actual (physical) faults can be modelled. But this higher precision is obtained at the expense of more complex test generation algorithms, longer test generation times, and longer test times.

4.2.2 Asynchronous circuit testing methodology

4.2.2.1 Self-checking circuits

Generally, circuits that have the property that they halt for all faults are called self-checking [Beerel 1992a, Varshavsky 1990]. A test for a self-checking circuit attempts to toggle all nodes at least once, i.e., during a test all nodes are driven both high and low.

However, as mentioned in [Hulgaard 1994], the method used for testing asynchronous circuits should be different from the one used for synchronous circuits. In synchronous systems special codes or state assignments are used so that the circuit produces an illegal output in the presence of a fault. Because a global clock signal exists in this kind of circuit, it is easy for a separate circuit (a checker) to synchronously detect the illegal output code and raise an error signal. Faults are detected while running the circuit at its operation speed (called on-line testing). While a synchronous circuit can easily be single-stepped through different states by using the global clock, this is much harder (sometimes impossible) for asynchronous circuits. The lack of global clock signal means that synchronization must be achieved by other means. Two general approaches have been used. One approach, used in the design of classical asynchronous state machines, is to make timing assumptions about the delays of the gates and wires (FM assumption). In addition, in order to avoid critical races and hazards it is often necessary to add extra, functionally redundant, circuitry and appropriate delays. The state machines have timing constraints that must be met to ensure correct operation (such as the FM assumption) and these constraints are hard to satisfy when composing multiple machines. This makes it very difficult to fully test this class of circuits. The other approach is to use explicit handshake signals for local synchronization. Because no absolute timing assumptions are made on the handshake, circuits are robust and easily composable, a property that has made this design approach popular. While the lack of global synchronization decreases the controllability of the circuit and thus makes an asynchronous circuit harder to test, the local synchronization tends to increase the observability.

Similar self-checking approaches used for testing synchronous circuits have been applied to testing asynchronous circuits. The method is based upon the theory of classical asynchronous state machines. The circuit will be brought into a special state when a fault exists [Sawin 1974, Mukai 1974]. However, designing asynchronous circuits using the classical state machine approach and related approaches has turned out to be problematic for large systems. Mostly it is not suitable for testing asynchronous circuits.

4.2.2.2 Test generation

The purpose of test generation is to determine input sequences that will cause a faulty circuit to behave differently from its specification [Hulgaard 1994]. This is a very popular method to test synchronous circuits. Basically, a test procedure includes three main steps: test pattern generation, applying the set of test patterns to the circuit under testing (CUT), and evaluating the responses observed on the outputs of the CUT [Petlin 1994, Petlin 1996].

The test pattern generation step is to derive those tests which will detect all possible faults. The test patterns can be applied in two ways. The first way is to use external test equipment to apply test to the CUT and check the responses. The second way presumes the application of test patterns inside the CUT. The method of applying test patterns internally is equivalent to the arrangement of the self-checking procedures discussed above. The results of the process of evaluating the responses obtained from the CUT can help to resolve two test tasks: the definition of a faulty circuit and the indication of the position of the fault in the CUT (fault location testing).

However, in asynchronous circuits, because of no global clock signals to synchronize the circuits, if the circuits have redundant logic, the behaviour of a faulty circuit may depend on the delays of the circuit elements. For example, a fault may cause hazards or races that only occur for certain combinations of delays. On the other hand, we cannot expect an event to happen at a given time in asynchronous circuits. So to guarantee that these circuits will work under a large range of conditions, the values of the actual delays must be checked. This needs special care in order to guarantee that no hazards and critical races we introduced during the test [Petlin 1994]. Generally this is impossible, especially for self-timed circuits.

4.2.2.3 Design for testability (DFT)

Generally, there are two key concepts involved in testing techniques: controllability and observability. A level of controllability and observability may not be sufficient to test all possible faults [Hulgaard 1994]. For example, to test for a premature firing, the circuit must be held in a state where the premature firing occurs and the faulty transition propagates to a primary output. This is not always possible. As a consequence, in order to increase the testability, techniques for testing asynchronous circuits by adding test circuitry during the design phase, usually called Design For Testability (DFT), have been proposed.

The procedure for DFT assumes that modifications to the circuit are made in order to ease the generation and application of test vectors to the circuit to be tested. To improve testability three groups of DFT techniques have been used: ad hoc strategies, structured approaches and built-in self-test techniques. There are several basic criteria which must be taken into account when choosing the most suitable DFT method for designing a VLSI circuit. These are as follows [Petlin 1994]:

- Impact on the original VLSI design: the increase in silicon area; effects on performance; the testability of the extra logic,
- The ease of implementation of the technique chosen,
- The effects on test pattern generation: reduction in computational time; improved fault coverage; reduction in engineering effort,
- Additional requirements for automatic test generation tools.

The simplest way to increase testability is to introduce a test point into the circuit. Test points are of two types. An observation point is used to access an internal node by making the node a primary output. A control point is used to set the value of an internal node from a primary input. A test point can also be both an observation and a control point. The schematic types of test points are shown in Figure 4.1. Where test points should be inserted to minimize the total number is a difficult problem.

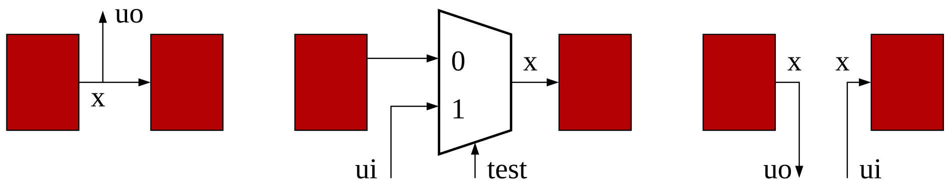


Figure 4.1 Introducing test points for signal *x*. An observation point (left), a control point (middle), and both (right)

Generalizing this idea leads to the popular method of simplifying test generation by introducing a scan-path. The registers in the circuit are extended to become scan registers, illustrated in Figure 4.2 using a conventional clocked register.

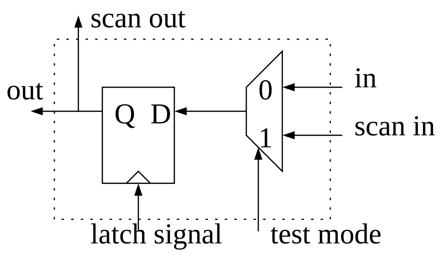


Figure 4.2 A scan-register.

In normal operation (*test mode* = 0), the scan registers work exactly the same as the original registers. In test mode, the scan registers form a shift register as the scan-output of one register is connected to the scan-input of the next. Their content can then be serially shifted out to the scan-output, and new values can be shifted in on the scan-input. Full observability and controllability is obtained for the values stored in the registers.

4.3 Testing method of ACM circuits

As described in Chapter 3, the ACM circuit has three important properties: asynchrony, coherence and freshness. These properties mean that an ACM can work as fast as possible or as slow as possible, even if one process of an ACM, the writer or the reader, stops. For example, the writer (the reader), one of two independent processes in an ACM, works on its own running cycle no matter what the reader (writer) does. In addition, our ACMs are implemented by using SI circuits. That means that the ACM is not only a self-timed circuit but also a globally unsynchronised system, and it is not a pure control unit but a communication mechanism with data transfer.

So testing this kind of circuit is a difficult problem, even more difficult than testing self-timed circuits (just circuits without a global clock).

Some methods for testing DI/SI circuits, such as the methods introduced in [Petlin 1996], have been proposed. However, most of them focus on the stuck-at fault problems using DFT techniques. This is not enough for testing ACMs, especially when testing the data coherence and freshness properties.

In order to test ACMs, some new methods are needed. In [Xia 1999b], a testing method has been proposed in which the property of freshness could be verified by testing for data item sequence. Based on the above asynchronous testing methods, this chapter shows that such ACM systems can be tested by looking at a sequence of data items, rather than individual items, whilst varying the rates of data communication. A schematic illustration of the method is shown in Figure 4.3, in which an evolution of testing methodology is shown. For fully ACM kinds of asynchronous circuits, both DFT (built-in) and varying delay methods are used from the testing evolution.

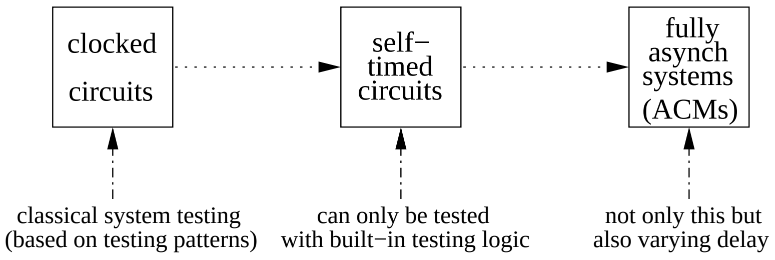

clocked circuits
self−timed circuits
fully asynch systems (ACMs)
classical system testing (based on testing patterns)
can only be tested with built−in testing logic
not only this but also varying delay

Figure 4.3 Evolution of testing methodology.

4.4 Test aims and environment

In our case, the three properties of ACMs (see Chapter 3) need to be verified. In addition, because this is a VLSI chip, testing is needed to find whether it works at all and whether it works as expected. Apart from these concerns, such additional problems as data loss need to also be investigated compared with the simulation results produced by the Cadence tool [Xia 2000b, Shang 2000b].

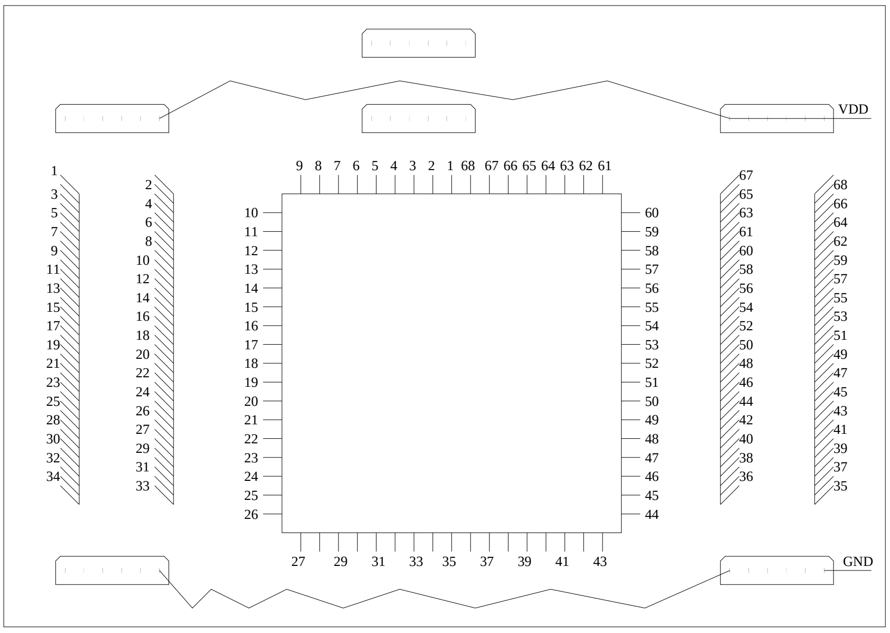


Figure 4.4 A simple testing board.

In order to test the chip, some facilities are needed. A simple testing board was built, which includes a 68 pin socket and connectors which are linked to the pins of the socket as shown at the middle part in Figure 4.4.

A digital analysis system (DAS 9100), a digital oscilloscope, an analogue oscilloscope and a multi-meter were also required.

A photo of our testing environment is shown in Figure 4.5.

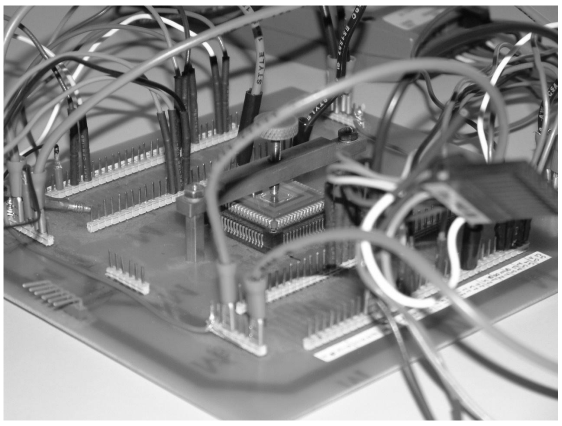


Figure 4.5 A photo of our testing environment.

4.5 On-chip testing circuit

During the development of the hardware implementation of the four-slot ACM, the problem of testing the important properties and general functions in hardware arose [Xia 2000b]. Although the general functions and the properties have been verified both theoretically and via digital simulation (using the Cadence tool), testing for such properties in hardware is necessary to show that the fabricated chip functions as designed. The ACM hardware was implemented with relatively fast VLSI technology, as testing was desired to be performed at maximum performance. In order to force the ACM to work at maximum speed, the testing circuits, the writer and reader hardware, had to be built into the same chip as the ACM and kept as simple as possible.

Based on the method proposed in [Xia 1999b], a simple solution consisting of a counter serving as the writer and a FIFO buffer serving as the reader is proposed. An on-chip FIFO buffer is needed because external connections cannot keep up with the production of data from the reader part of the ACM, and dumping data out directly implies slowing down the reader via its external handshakes. With an on-chip FIFO, data from the reader can be downloaded off-line after an experiment. In order to test the ACM efficiently, a function part which can vary the cycle length is also needed. This is implemented by a delay part which is adjustable. Of course, control circuits are needed in the testing circuits. The whole block diagram is shown in Figure 4.6. In order to keep the testing circuit simple and the testing process efficient, the testing circuits and the ACM can be run in several modes which are set up before testing.

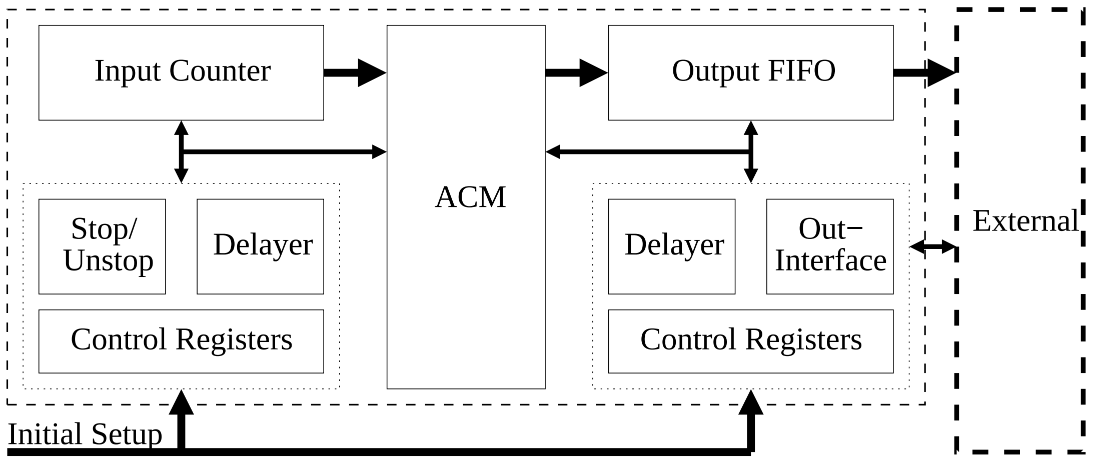


Figure 4.6 The block diagram of the on-chip testing circuits and the ACM.

The testing circuits work as the environment of the ACM. They can supply data to the ACM, and collect data from the ACM. In order to save space, only 8-bit data path width is used. Because all circuits are SI, they are easily connected together.

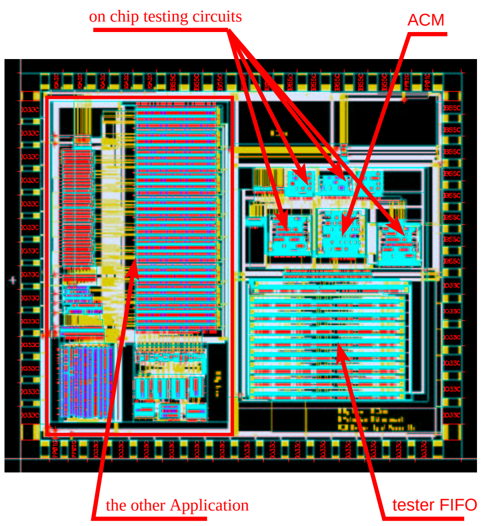


Figure 4.7 The floorplan of the chip.

The 8-bit input counter, which is a self-timed dual-rail ripple carry counter, is designed as the data generator (the writer). Data with values from 0 to 255 can be sent to the ACM in increasing order. On the opposite side, the FIFO buffer collects the data from the ACM as the reader. In order to analyze the data collected from the ACM, the FIFO should be big enough to store as much data as possible. On this chip, the ACM and its on-chip testing circuits share the chip area with circuits from other projects. As a result, there is not enough space to build a really big FIFO. Eventually, a 64 stage FIFO, which occupies more than half of the total chip area used by the ACM and its testing circuits, was included on-chip. The floorplan of the chip is shown in Figure 4.7.

In order to test the properties of the ACM, some relative timing assumptions are needed. So in the on-chip testing circuits, two delay parts are designed for this purpose for both the writer and reader respectively. For each delay part, there is an 8-bit counter using the same technology as the input counter. The delay length of these counters can be set up by writing an initial value, which is chosen from 1 to 255 based on the requirements of the control circuit. In testing, when the counter counts up to the initial value, it takes some time. We use this method to realize delays for the writer and reader. However, from a time point of view, the counter is discrete with too large a granularity (about 2ns for each step). In order to adjust the delay as finely as possible, another type of delay is also included. This part consists of inverters connected in series. In our case, there are 64 inverters in total through the input data bus which can supply eight kinds of different delays. These provide the fine-tuning required.

In the on-chip testing circuits, control circuits are needed to control the writer, the reader and the delay parts. For example, one control circuit is used to decide whether the circuit should continue counting or whether it should stop (*done*). The function of the circuit described in STG format is shown in Figure 4.8. When a delay request (*delay_start*) arrives, the counter will add one, and then compare the result with the value set up initially. If these are equal, the *done* signal will be produced and the *delay_start* signal will be withdrawn; if not, the *go_on* signal will be given to stimulate the counter again. The circuit implementing this function is shown in Figure 4.9. This circuit was derived by using the Petrify tool, and was subsequently optimized by hand.

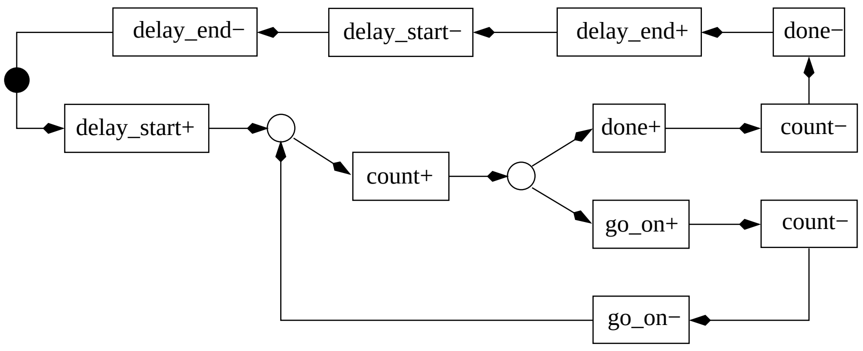


Figure 4.8 The STG for a control circuit.

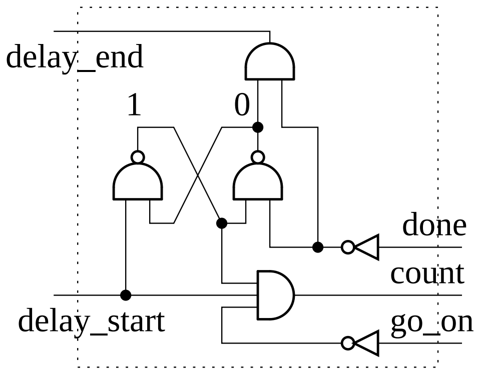


Figure 4.9 The logic circuit of choice function for the delay counter.

In order to test the chip, at the writer side, an input counter with two working modes was designed. The first mode is stoppable: when it counts to 255, the writer will stop. This is used for downloading the data from the FIFO to analyse. The other mode is unstoppable. The writer will send data from 0 to 255 and repeat this cycle. This is used for debugging the circuit on-line. This function is realized by introducing a simple switcher gate. We will not show it here.

The operations on the writer side are:

- Preparing the data;
- Writing into the ACM;
- Checking whether to finish and
- Inserting a delay i.e. (adjusting the delay length).

The PN model of the counter circuits for this part is shown in Figure 4.10. The control circuits are built by using DCs and translated from the PN model directly by hand. The circuits are shown in Figure 4.11. An automatic tool used to translate PN model to DC circuits is introduced in Chapter 5.

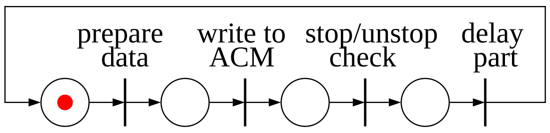


Figure 4.10 The PN model of the writer part.

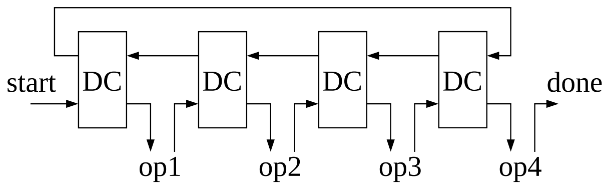


Figure 4.11 The control circuits of the writer part.

At the reader side, the main control circuits are implemented by using a method which is different from the writer side (which uses DCs). Four operations, reading the ACM, writing to FIFO, reading from the FIFO and setting delay, are connected using handshake circuits. This FIFO works as a Micropipeline [Sutherland 1989]. So, after the ACM is read and the reading operation is done, a writing request to the FIFO will occur. After the writing to the FIFO operation is done, the delay action, which is used to adjust the delay length in testing, will happen. The FIFO reading operation is only used by external connections for collecting the data from the chip. At the read side, when the FIFO is full, reading from the ACM will stop. However, we have included the external data interface (to an outside computer for example) in the FIFO, so it is possible to incorporate some degree of on-line downloading of reader output data.

4.6 Testing methods and results

As stated above, certain basic testing circuits, such as the input counter, the FIFO buffer, the delay parts and the control circuits, have been built on the same chip with the ACM in order to obtain results when the ACM is running at full speed. Testing functions supported by the on-chip testing circuits are described below:

- The input counter can produce a sequence of data items with ascending value. Since the order of the data going into the ACM from the input counter is known, the data retrieved from the FIFO can be compared, to verify data sequencing.
- There are two working modes for the writer and reader sides, one is stoppable and the other is unstoppable. These can be set up initially based on the testing requirement.
- The relative timing is variable for the writer and reader by adjusting the delay lengths of the writer and reader parts in order to realize the different speed of the writer and reader.

What we do when testing is to analyze the data collected from the observation points, including the output of the FIFO, and the timing relationship for the control logic signals.

In our testing circuits, the data values coming from the input counter, the ACM and the FIFO can be observed. In addition, some control signals, such as "*start*" and "*done*" handshake signals, can also be observed by using oscilloscopes.

What we consider is how to use these functions provided by the testing circuits and observation points to analyse the data and timing relationship for the properties of the ACM and the general function of the circuits better. In order to test the properties of the ACM, the simplest method is to analyze the data and the timing by adjusting the relative timing between the writer and reader. Initially, some control parameters, such as the delay lengths for the writer and the reader, need to be set up. This is done by writing data chosen from 1 to 255 to the relevant registers in the testing circuits.

The concepts of asynchrony, data coherence and freshness have been defined in Chapter 3. Based on these concepts and the functions provided by the testing circuits, some testing methods for these properties are proposed.

The following test programs analyze the data collected from the observation points, including the output of the FIFO, and the timing relationship for the logic signals. They are:

- The writer and reader are both run as fast as possible;
- The writer is run faster than the reader;
- The writer is run slower than the reader;
- With the writer stopped, the reader is run as fast as possible;
- With the reader stopped, the writer is run as fast as possible.

4.6.1 Testing general functioning

The general functioning of the ACM and on-chip testing circuits need to be tested. If they are not correct, any subsequent testing for the properties of the ACM can not be considered.

We analyzed the timing of *start*/*done* handshake signals, initial values, wrote the data to the ACM and then read it from the FIFO. The results demonstrated that the general functioning is as expected.

4.6.2 Testing asynchrony

In asynchrony testing, the assumption is that if the writer and reader do not run independently, it means that the writer (or the reader) will wait for something coming from the reader (or the writer) and when the writer (or the reader) runs slowly or stops, the reader (or the writer) should be affected. In other words, if the two sides are temporally dependent, the faster side will be slowed down by the slower side. In fact, no matter how we adjust the delay time for the reader and/or writer, no waiting occurs according to the waveforms observed on both a digital and an analogue oscilloscope. This indicates that the ACM on this chip does support asynchrony.

4.6.3 Testing data freshness, coherence and data loss

The same testing methods are used for these properties as were used in the above section (4.6.2). The difference is in the observed signals. Here we focus on the data results collected from the ACM when we adjust the delay time for the writer and reader. Since the order of the data going into the ACM from the input counter is known if we adjust the relative timing for the writer and reader, the data collected from the FIFO buffer should be regular and have the same order as the input data. Otherwise data freshness is violated. In our case, in order to save chip space and money, the width of the data transferred to the ACM and the FIFO buffer is 8 bits. With such a data packet size, it is very difficult to verify possible violations of data coherence. In the circuits, the logic guarantees that the writer and reader cannot access the same slot (packet) at the same time. If the writer and reader can access the same slot at the same time, the data will then be combined and the frequency of the data appearing will not be right. This implies that loss of data coherence should show up in the observation as loss of data freshness. So by testing for data freshness we are also testing for data coherence.

In actual testing, we have obtained a wide range of results. Two examples are shown in Figure 4.12 and Figure 4.13.

With Twrite =80ns, Tread = 110ns, Tsample = 40ns.											
1.	41	7.	44	13.	46	19.	49	25.	4C	31.	4F
2.	41	8.	44	14.	46	20.	49	26.	4C	32.	4F
3.	41	9.	44	15.	46	21.	49	27.	4C	33.	4F
4.	42	10.	45	16.	48	22.	4A	28.	4D	34.	50
5.	42	11.	45	17.	48	23.	4A	29.	4D	35.	50
6.	42	12.	45	18.	48	24.	4A	30.	4D	36.	50

Figure 4.12 Testing results.

With Twrite =80ns, Tread = 80ns, Tsample = 40ns.											
1.	73	7.	76	13.	79	19.	7C	25.	7F	31.	82
2.	73	8.	76	14.	79	20.	7C	26.	7F	32.	82
3.	74	9.	77	15.	7A	21.	7D	27.	80	33.	83
4.	74	10.	77	16.	7A	22.	7D	28.	80	34.	83
5.	75	11.	78	17.	7B	23.	7E	29.	81	35.	84
6.	75	12.	78	18.	7B	24.	7E	30.	81	36.	84

Figure 4.13 Testing results.

From the testing results, we can see that the data from the FIFO is very regular without any obvious violations to data freshness and coherence, and the patterns of data overwriting and re-reading (characteristic of the four-slot ACM) can be clearly seen.

In addition, when the speed of the writer and reader does not match, data loss may happen. The loss rate depends on the difference between the speed of the writer and reader. The results are similar to the simulation results obtained by using the Cadence tool.

4.7 Conclusion

The ACM used for testing is a four-slot *Pool*. It is implemented using SI circuits under the Cadence VLSI design tool (0.6 micron technology). The layout is shown in Figure 4.14. In addition, it is fabricated by EuroPractice. It is packed in a 68 pin JLCC68 package. It is shown in Figure 4.15.

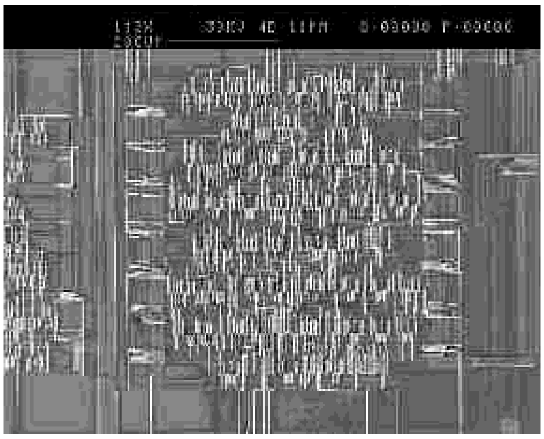


Figure 4.14 The layout of the ACM.

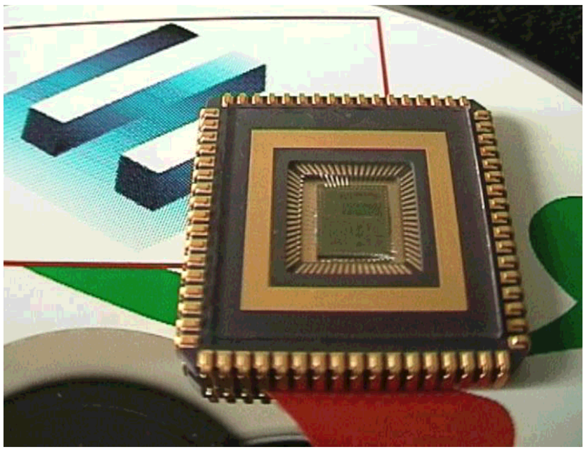


Figure 4.15 The VLSI chip of the ACM.

The testing results show the slot ACM implemented by using self-timed circuits performed as expected. The mini-interlock introduced by using Mutexes does not affect the operation of the ACM appreciably. No obvious violations of the three important properties have been observed. In addition, the testing results corresponded with the simulation results obtained by using the Cadence tool, which indicates that the implementation did not introduce errors.

The testing methodology, in addition, is shown to be appropriate. The testing is successful and on-chip testing circuits performed as expected.

Chapter 5: A New Direct Translation Synthesis Method and the PN2DCs Tool based on Petri nets

5.1 Introduction

In spite of a large amount of research done in the last two decades, asynchronous circuits are still difficult to design, especially manually. One main reason is the absence of mature asynchronous CAD tools to support asynchronous circuit designs [Sutherland 2002].

In Chapter 1, we mentioned that a large number of automatic asynchronous design techniques have been studied so far, and some asynchronous CAD tools have been developed. However, they are not mature. There are some problems in these CAD tools. They will be discussed in this chapter.

In Chapter 1 and Chapter 2, we found that although asynchronous circuits show many benefits and can be expected to show benefits in the future, the above problems will affect the use of asynchronous techniques. As main stream techniques, they should possess properties such as quickly adapting to market changes and new technology. In order to meet the above requirements, automated design tools are needed.

From the asynchronous experiences shown in Chapter 3, we obtain a method for designing self-timed circuits, especially SI circuits, to solve the above problems.

In this chapter, an attempt will be made to develop a direct translation synthesis method based on PNs.

5.2 Existing synthesis techniques and problems

Significant efforts have been spent on studying and proposing automatic asynchronous circuit design techniques that can alleviate the burden of design.

Automated logic circuit design flow for both synchronous and asynchronous circuit designs generally consists of several parts, such as input, synthesis, simulation, placement, routing and so on. A traditional design flow is shown in Figure 5.1.

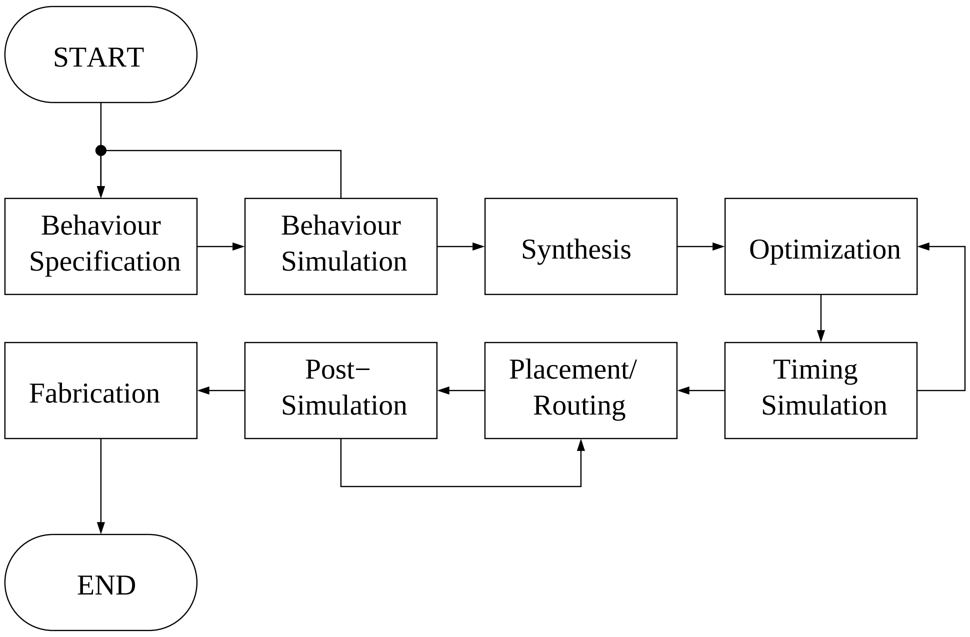


Figure 5.1 The traditional design flow.

The main difference between synchronous and asynchronous design flow is the synthesis techniques. In asynchronous circuit design, the correctness of an asynchronous circuit not only depends on its structure, but also on the timing behaviour of the individual gates and their interaction. So synthesis techniques in asynchronous circuit designs are more important than in synchronous ones.

As an important stage in the automatic asynchronous circuit design flow, logic synthesis for asynchronous circuits has been the focus of attention of many researchers recently. Many attempts in this area have been made. Some successful synthesis tools, such as the Petrify tool, have been presented.

In [Jung 1999 and Carmona 2001], a brief summary of asynchronous logic synthesis techniques was reported. In general, synthesis techniques can be classified into two types, one is based on the presentation of state space with abstract variables and the other is direct (or syntax-driect) translation.

5.2.1 Logic synthesis

Logic synthesis techniques have been proposed in [Chu 1987, Meng 1989, Beerel 1992b, Couvreur 1994, Lavagno 1991, Myers 1993, Cortadella 2002]. They are based on a state graph, in which each reachable state is assigned a binary code with the value that represents the value of each signal at the state. Deriving logic equations requires the generation of the binary codes for all states. There are two methods to assign the binary codes to the states in the graph.

Most synthesis tools perform an exhaustive token flow analysis to obtain the complete reachability graph and all binary codes. In this kind of tool, it is inevitable that the state explosion problem is encountered.

Recently a new kind of method, called structural method, has been proposed [Pastor 1998, Carmona 2001], in which “structural” means “at Petri net level” without requiring the explicit generation of the reachability graph. The methods are based on a structural encoding of the systems.

Unfortunately, none of the above methods has been able to effectively tackle the problem of finding an encoding of the specification that guarantees an implementation, even with known structural methods working on some subclasses of STGs [Carmona 2001].

Apart from the above problem, an assumption is introduced when SI circuits are constructed. It is that the used gate library contains basic gates such as ANDs, ORs and Muller C-elements with arbitrary fan-in and fan-out, and any number of inverters with zero delay attached to the inputs. In fact, it is impossible to find such libraries.

Although these problems exist, there are several academic tools that use this kind of logic synthesis method to produce circuits. This is because these tools work at the logic level and attempt to optimize the resulting circuits by using variations of state-of-the-art minimization techniques. As a result, the circuits produced can be very effective and concise.

5.2.2 Direct translation

To overcome the state explosion problem, direct translation techniques that linearly depend on the specification complexity have been proposed. Direct translation techniques have a long history originating from Huffman’s work [Huffman 1954], where a method of one-relay-per-row realisation of asynchronous sequential circuits was introduced. This approach has been further developed by many researchers, such as Unger [Unger 1969], Hollaar [Hollaar 1982] and Varshavsky [Varshavsky 1996].

Although direct translation methods have no state explosion problem and are therefore more likely to guarantee an implementation, they do not exploit the potential optimizations that can be performed at the logic level. Direct translation methods usually generate circuit structures that cannot be locally transformed to derived succinct representation of the same behaviour. Generally the size of the obtained circuits is linear to the size of the specification. So they have problems in cost and performance. On the other hand, because a direct translation method is structure related, normally the circuits obtained should work under timing assumptions.

Varshavsky’s method, which guarantees that resulting circuits are SI, belongs to the class of direct translation algorithms. Based on PN specifications, this direct translation method translates each place to a DC (refer to Chapter 2) and each transition to an event, which is inserted between two relevant DCs.

A problem with this method is that within the DC definition (**Definition 2.8**), with different set and/or reset functions, there exist a large number of different DCs. It is a non-trivial problem to select the right type of DC for any particular place in a PN specification.

Fortunately there are only five basic elements in PNs. They are linear, event join, exclusive join, event fork and alternative fork [Varshavsky 1996, Yakovlev 1998]. A PN specification should be constructed using these kinds of basic elements (maybe not all of them). Because of the properties of this direct translation method, place to DC and transition to event, what we need to do is to translate all basic structures to logic circuits and then to construct all circuits obtained to a complete circuit.

The implementations and PN specifications of the above basic elements with two or less predecessors and successors, such as linear, event join, exclusive join, event fork and alternative fork, can be found in [Varshavsky 1996]. The simple schematic representations are given in Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5, and Figure 5.6 respectively. For more details please refer to that paper.

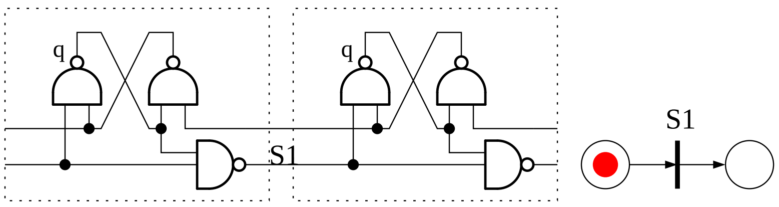


Figure 5.2 The implementation and PN specification of a linear fragment.

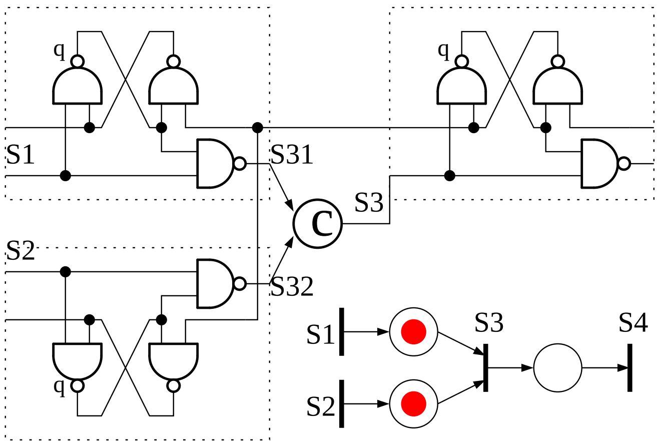


Figure 5.3 The implementation and PN specification of an event join fragment.

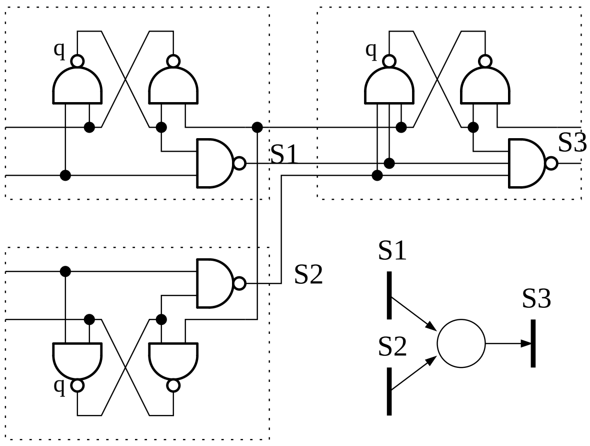


Figure 5.4 The implementation and PN specification of an exclusive join fragment.

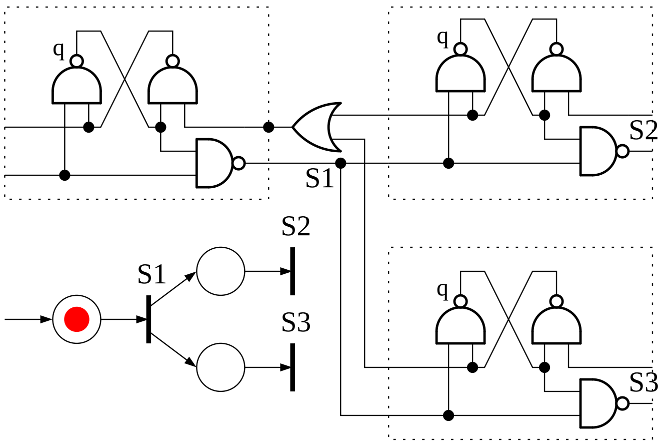


Figure 5.5 The implementation and PN specification of an event fork fragment.

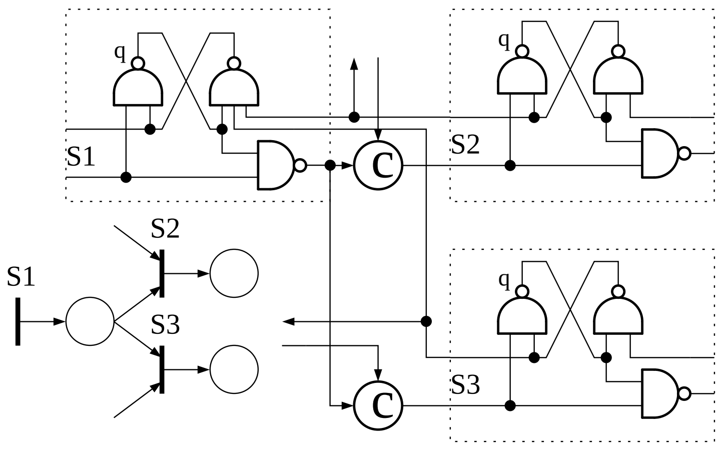


Figure 5.6 The implementation and PN specification of an alternative fork fragment.

Here we consider only two or less predecessors and/or successors in join and fork type basic elements. In fact, it is possible that there are three or more predecessors and/or successors in a joined or forked basic element. Based on the rules of PNs, such a complex specification can be decomposed to one constructed using simple basic elements with only two or less predecessors and/or successors. This will be discussed in detail in section 5.5.3.

From this, it may sound as if it is easy to translate specifications to DC circuits. However the circuits implemented directly using Varshavsky's method are autonomous (no inputs/outputs). The only reasonable interfaces between the control circuits and the environment are a set of abstract processes, implemented as request-acknowledge handshakes, which are inserted into the breaks in the wires connecting the DCs. This restricts this method to high (abstract) level design. The DC circuits obtained are usually the control part in systems which also contain events being controlled (data paths) as the environment of the control part. Autonomous circuits include both control and data parts. We may treat low (signal) level signals as abstract processes. However this will affect the functionality, the silicon cost and performance. We use an example (Figure 5.7) to demonstrate these points.

Although in [Varshavsky 1996], an attempt to apply this method to low (signal) level designs and to introduce external signals (inputs) was mentioned, it is not complete. Inputs especially were still represented as abstract processes and some additional DCs were introduced when implemented [Bystrov 2001].

Low level design is inevitable for most synchronous and asynchronous logic circuit designers. Normally in low level design, the actions of signals are described very clearly. Only when a system is translated into the low level, an appropriate gate level circuit can be easily generated and then it can be placed, routed and finally fabricated.

In low level event control systems, we assume that events are changes of binary signals. In other words, our systems control the setting to 1 or resetting to 0 of binary signals. This corresponds with 1-safe PN specifications [Yakovlev 1998].

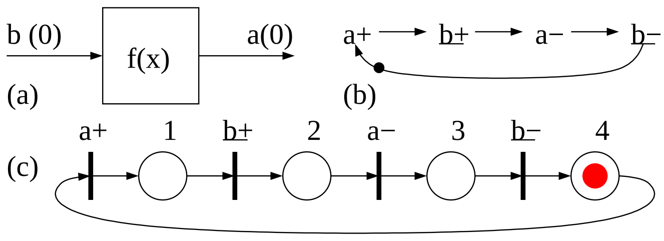


Figure 5.7 (a) The block diagram (b) Its function in STG format (c) Its PN specification.

Figure 5.7 gives an example specified at a low level. Signal *a* is an output and signal *b* is an input. The block diagram and function specification in STG format are shown in Figure 5.7 (a) and (b), respectively. In order to demonstrate the method, we convert the STG specification to the PNs which are shown in Figure 5.7 (c). This is the STG with the explicated places. Note that the most useful STGs are 1-safe PNs, because they adequately capture the behaviour of control flow in hardware [Yakovlev 1998].

Based on the direct translation method, each place is translated into a DC and each transition is mapped to an event. This generates the control circuit shown in Figure 5.8 (a), in which all events have been inserted. Because this example is a simple one with only linear basic elements but no join and fork basic elements, only the simplest DCs are used. This circuit does not include circuits implementing the events a+, b+, a- and b-, which will need separate attention.

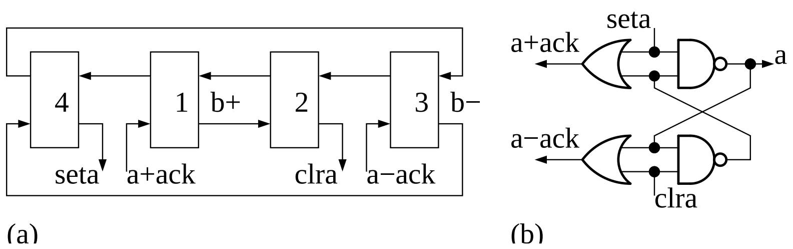


Figure 5.8 (a) An implementation (b) An SR latch with completion detection logic.

Because signal *a* is an output, we can use a modified SR latch shown in Figure 5.8 (b), in which a normal SR latch with completion detection functions implements events a+ and a-. Note: in this chapter, the modified SR latch with completion detection logic is called an "SR latch" and the normal SR latch is called "normal latch" to distinguish them. When DC4 produces a request signal (*seta*=0), this circuit changes signal *a* from 0 to 1. After the SR latch settles down, i.e. one state (signal *a*) of the circuit has become 1 and its complementary state has become 0, an acknowledgement signal (*a+ack*), which indicates that event a+ has been accomplished, will be sent to DC1. Event a- can be implemented in the same way.

If we use the same method to implement events b+ and b-, according to the direct translation method, b+ will be inserted between DC1 and DC2, and b- between DC3 and DC4. They will then be stimulated by request signals from DC1 and DC3. This is obviously wrong because signal *b* is an input. From the original specification, the events b+ and b- fire only depending on signal *a* in the environment. In other words, events "b+" and "b-" are started by signal *a* directly. On the other hand, it is unnecessary to keep input signals using SR latches the same as output signal *a* in control circuits. Note that the control signals *seta* and *clra* cannot happen at the same time (period). It is a reasonable assumption which does not affect the specification.

This is a problem for the direct translation method in implementing asynchronous event control systems specified at a low level. In addition, as mentioned in [Varshavsky 1996], the simplicity of the general technique itself does not guarantee efficient and simple realizations.

Although Varshavsky's direct translation method has disadvantages, his method has a strong appeal to us. Because of the straightforward nature of this method, automation is a distinct possibility. In addition, the circuits obtained by using this method are SI. However, all of the above problems must be solved during the development of any viable automatic tool.

5.3 Our synthesis design flow

In order to generate a good performance and low silicon cost circuit automatically, we propose a synthesis design flow based on the design flow of synchronous circuit designs.

Our asynchronous design flow consists of the following parts:

1. Initial specification;
2. Decomposition and refinement of specification;
3. Synthesis;
4. Technology mapping;
5. Optimisation.

Note that because the solution produced is SI, the circuit can work correctly under any technology. That means we do not care what kind of technology is provided. So the synthesis and technology mapping processes in our design flow are combined into one process. We named it “direction translation method” (DT).

The schematic diagram of our design flow is shown in Figure 5.9.

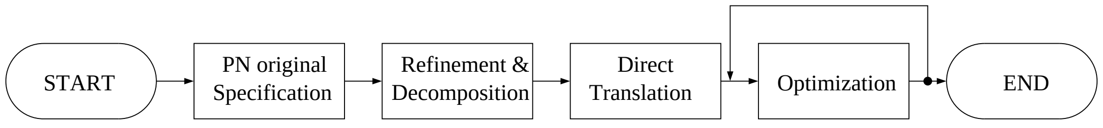


Figure 5.9 Our synthesis design flow.

All processes will be discussed in detail in the following sections.

5.4 New direct translation method and specification refinement

5.4.1 New method and specification decomposition

Varshavsky’s method [Varshavsky 1996] cannot translate input events specified in a low level correctly (refer to section 5.2.2). In order to solve this problem, we propose the following method.

5.4.1.1 New direct translation method (NDT)

Basically this method is based on Varshavsky’s. The same example shown in Figure 5.7 is employed here to introduce this new method. The control circuit shown in Figure 5.8 (a) has been generated based on Varshavsky’s method. All output signals can be implemented directly using memory elements, such as SR latches, which are inserted between the appropriate pairs of DCs in the control circuit.

Input signals, however, should be controlled and generated by the environment. Only after the environment senses a change in signal *a* will it cause event b+ or b-. The control circuit should be made to wait for changes in signal *b* at the relevant points. Specifically, DC2 should wait for event b+ and DC4 should wait for event b-. In this example, a NAND gate and an INV gate between DC1 and DC2, with the *fw* signal of DC1 and signal *b* as inputs and the *s* signal of DC2 as output, implement the DC2 wait. The detail of the DC2 wait implementation is that the *fw* signal passes the INV gate and then connects to one input pin of the NAND gate; the input signal *b* from the environment goes to the other input pin of the NAND gate directly; the output of the NAND gate connects to the DC2. This circuit is SI and implements the wait function correctly. The reasons for this will be explained in section 5.5.3. Similarly, an OR gate can be used between DC3 and DC4 to implement the waiting for event b-. The complete circuit is given in Figure 5.10. Note that the DCs being used happen to have the simplest possible set and reset functions (f(set) and f(reset) in **Definition 2.8**). This is because of the simplicity of the system specification (no forks or joins in the PNs). However, even with forks or joins in the specification PN, set and reset functions for the affected DCs would be easy to derive.

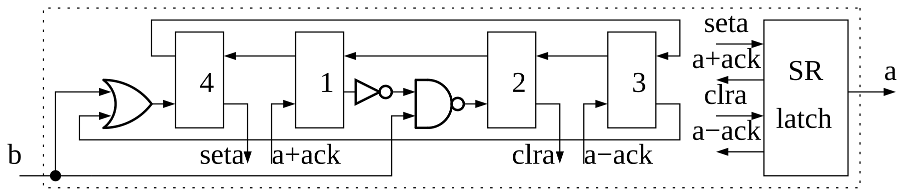


Figure 5.10 An implementation.

Theoretically, any persistent PN specification can be translated into a DC circuit using this method. Each place is translated into an appropriate DC, and each transition into either a memory element holding this output signal or some logic circuits waiting for the input signal. This is guaranteed under the assumption that all kinds of DCs exist in libraries. However, a given library must have a finite numbers of gates. This assumption is not practical. In order to implement real-life DC circuits, we must restrict the number of the gates in a given library.

According to the DC definition (**Definition 2.8** in Chapter 2), in a DC, only the set function and reset function are changeable. Generally they are decided by the number of predecessors and successors of the mapped place respective. The number of the fan-in in the set function is the same as the number of the predecessors. The number of the fan-in in the reset function is the same as the size of the successors. So in order to limit the numbers of a given library, we must restrict the numbers of the predecessors and successors for each place.

On the other hand, from the logic design point of view, fan-in and fan-out affect the performance of logic gates. Generally, gates with three or less fan-in and/or fan-out are preferred. Based on this discipline, we can define a library used for our method. It will be introduced in section 5.5.2. At the same time this discipline restricts PN specifications too. In order to implement a specification using the given library, a PN specification must be transferred to one in which each basic element has no more than three inputs or outputs. Here we prefer two or less inputs and/or outputs for each basic element.

Similar to Varshavsky, we only study the five kinds of basic elements in our method. This does not affect the generality of our method because based on the rules of PNs, a complex specification fragment can be decomposed into a simple one. Complex event join, exclusive join, event fork and alternative fork specifications can be decomposed into more simple elements with the techniques introduced below.

5.4.1.2 Event join case decomposition

A complex event join fragment with more than two inputs is shown on the left side in Figure 5.11.

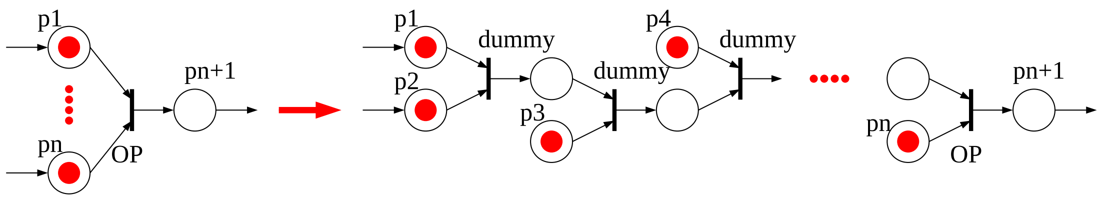


Figure 5.11 A complex event join case and a possible decomposition.

In Chapter 2, we mentioned that all PN specifications in this thesis are 1-safe. Based on PN rules, the transition in this fragment may fire only when all its predecessor places have a token. If we directly implement this fragment based on our method, there should be multiple inputs to a C-element or a dummy DC (see section 5.5.2), which is unlikely to be available in the library. In order to solve this problem, decomposition is needed. The method is to introduce some dummy events and places. A complex fragment is replaced by using a tree-structure basic element in which each simple fragment has two or less inputs. All input places are divided into several groups. For each group, one dummy transition and place are introduced. If the two places have tokens, this transition can be fired and then the token will be transferred to the dummy place. All the dummy places introduced are coped with using the same method. Finally only two or three dummy places are synthesized at the transition used in the original fragment. This process is illustrated in Figure 5.12.

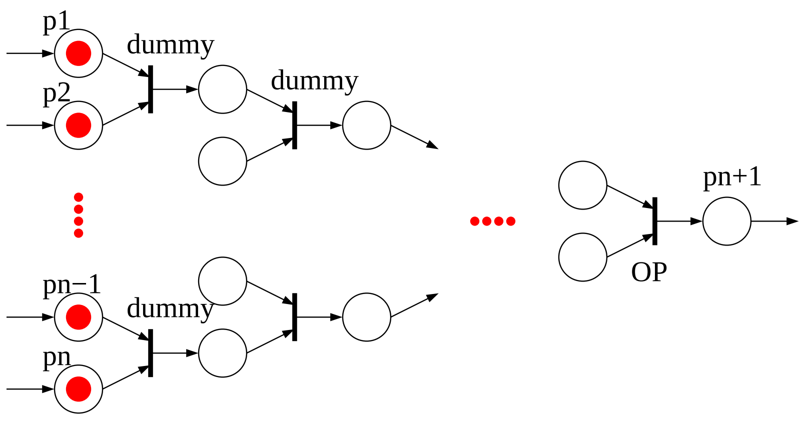


Figure 5.12 Another possible decomposition.

Another option is that firstly two input places are synthesized at a dummy transition and the token is transferred to a dummy place, and then this dummy place is synthesized with the third input place at another dummy transition and another dummy place is used to keep the token, and so on. Finally, all input places are synthesized at the original transition. The illustrating diagram is shown on the right side in Figure 5.11. These techniques will allow a specification of an arbitrarily large *n* to be implemented with DCs with two or less input links.

5.4.1.3 Exclusive join case decomposition

As in the case above, it is unreasonable to assume that there are only two inputs to a place. So in this case, before translating, decomposition is needed. A similar method as the one dealing with the event join cases is used here based on PN rules.

A complex exclusive join fragment is shown on the left in Figure 5.13. Because of the one-safe PN limitation in this thesis, among all input transitions of a place, only one transition can fire each cycle in this exclusive join case. The above method used to deal with the event join cases is used here to construct a tree structure in order to replace the original complex fragment. In this tree structure, each place has two or less input transitions. Dummy transitions and places need to be introduced. The illustrating diagram is similar to one shown in Figure 5.12 (not shown here). Another option is that two transitions are joined at a dummy place and this place can fire a dummy transition if there is a token in one place. After that this dummy transition and the third transition join at another dummy place and so on. Finally the last transition and a dummy transition (introduced earlier) join at the place specified in the original fragment.

The illustrating diagram of this method is shown on the right in Figure 5.13.

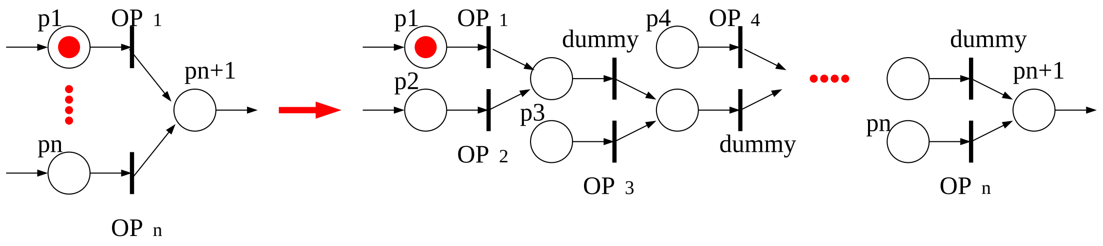


Figure 5.13 A decomposition method for a complex exclusive join case.

Note: there are two general methods to decompose complex PNs. We only present one diagram here to illustrate the decomposition.

5.4.1.4 Event fork cases decomposition

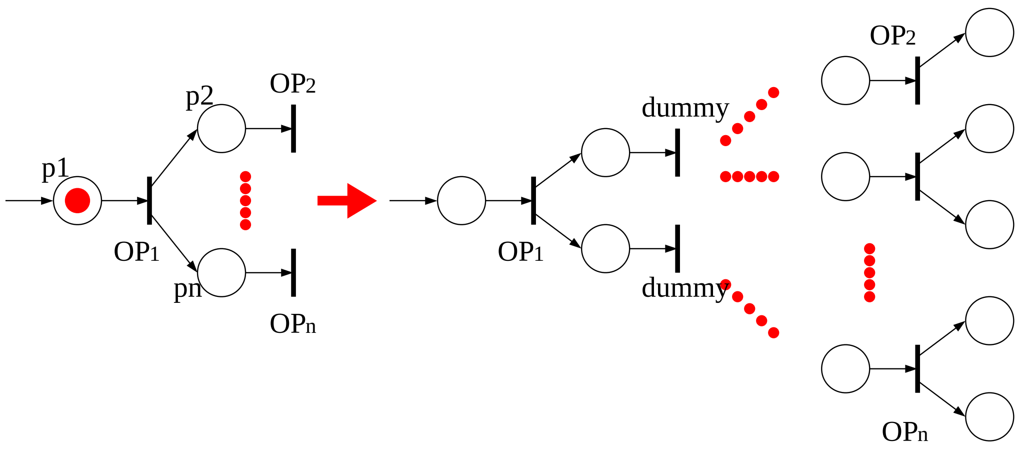


Figure 5.14 A decomposition method for a complex event fork case.

A complex event fork fragment is shown on the left in Figure 5.14. Since PNs do not place limitations on the number of successors of this kind of place, the *fw* signal of the DC mapped from p1 place (see Figure 5.14) or the acknowledgement signal of *fw* will go to or from several other DCs. This introduces fan-out problems. In addition, fan-in problems still exist, as in the above event join and exclusive join cases. This is because all the next DCs will give back the *bk* signals when all of them hold a token. All *bk* signals will go to the DC mapped from p1 to reset it.

So in order to generate a reasonable circuit, it is necessary to decompose this kind of fragment. This can be dealt with using the methods introduced in section 5.4.1.2. One possible decomposition method is shown on the right in Figure 5.14.

5.4.1.5 Alternative fork cases decomposition

Both fan-in and fan-out problems exist in this kind of case due to the absence of limitations of alternative forks (numbers of successors). It is possible to decompose the original fragment into a tree structure to solve the problems based on the rules of PNs using similar methods as discussed above. One possible decomposition method is shown on the right in Figure 5.15.

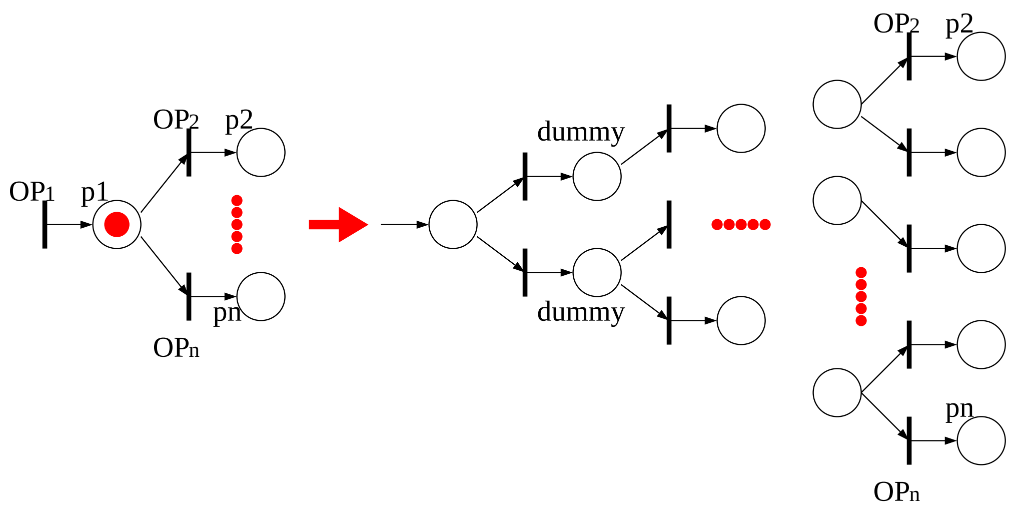


Figure 5.15 A decomposition method for a complex alternative fork case.

Note: this NDT method is relatively straightforward. It can be used to deal with decomposed PN specifications directly.

5.4.2 Refinement to the method and specifications

5.4.2.1 Basic ideas of specification refinement

Although with all the techniques introduced so far a DC based circuit can be found for any PN specification, the resulting circuit is likely to be large and slow. This is because of the large number of memory elements presented in such circuits. For example, the solution shown in Figure 5.10, although straightforward, uses five memory elements. This is mainly the result of strictly keeping the control path (the DCs) and data path apart.

In fact, duplications of memory elements exist in the circuits obtained using the NDT method which was introduced in section 5.4.1. By eliminating these duplications and assuming that the responsibility of holding any signal rests with the part that generates it, savings can be made.

We use the example shown in Figure 5.7 to introduce this refinement. In section 5.4.1, the circuit shown in Figure 5.8 was obtained. In this case, DC1 and DC3 combined hold the states for events b+ and b- internally. However, because signal *b* is an input, the environment must hold its value somewhere else at all times. DC4 and DC2 combined hold the states for events a+ and a- internally. Although signal *a* is then stored by its own holding latch for interfacing with the environment, DC4 and DC2 cannot be eliminated. This is because DC4 and DC2 must perform token propagation functions. Although DC1 and DC3 have the same function as DC4 and DC2, because signal *b* is an input and maintained elsewhere at all times, they are not necessary. They can be eliminated by combining the events a+ and b+, a- and b- into two events, one called a+,b+ and the other called a-,b-. A refined specification based on these ideas is shown in Figure 5.16 (a).

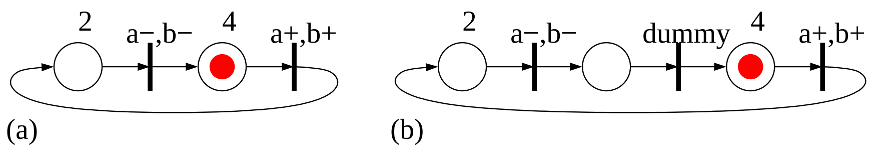


Figure 5.16 Refined specifications.

In this PN specification, there are only two transitions and places. This introduces another problem for the direct translation method. In general, it is impossible to directly realize cycles of a length less than three with DCs, because with the accepted discipline of token change, deadlocks appear [Varshavsky 1996]. Two methods were proposed to solve this problem in the same paper. In order to keep our method simple, we transform a cycle of length less than 3 into a cycle of length 3. So in this example, a dummy place and a dummy transition are employed as shown in Figure 5.16 (b).

This new specification can be used to directly translate. The events are a+,b+, a-,b- and a dummy one. However, because the polarity of DCs is not as conveniently organized as before, the “waiting for b to change” part needs further consideration.

5.4.2.2 Refinement technique

Fortunately the signals *a* and *b* are a pair of handshake signals. The SR latch holding *a* can be modified to accommodate both directions of change for *b*. This is shown in Figure 5.17 (b). Here we open up a connection in the original SR latch. Let the output signal *a* go to the environment directly, and the input signal *b* go to the SR latch directly replacing the internal connection (signal *a*). Using this modified SR latch, when the *set* signal arrives, signal *a* will go to 1 and this is passed to the environment. After that, the environment should respond by setting signal *b* to 1, passing it back to the control circuit in reply, and then the state of the SR latch settles down. Then an acknowledgement signal (*setack*) will be given to the next DC. The event a-,b- can be implemented in the same fashion. The dummy event is empty, so no extra circuits are inserted. The *fw* signal of the Dummy DC goes directly to DC4. The resulting circuit is shown in Figure 5.17 (a). There are four memory elements in this implementation. There are also some small savings on the number of simple gates in the NDT, because the signals *a* and *b* form a handshake pair.

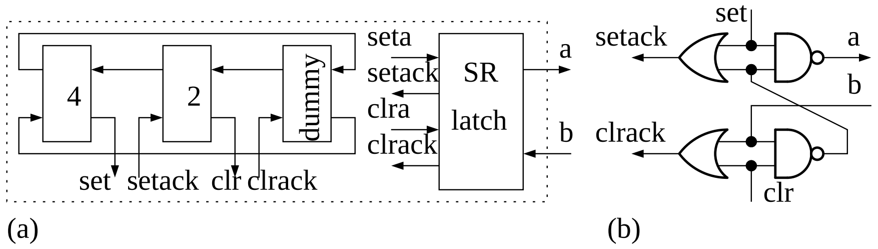


Figure 5.17 (a) An implementation (b) A modified SR latch.

However, in general, it is unreasonable to expect that input and output signals always form handshake pairs. In case handshake pairs cannot be easily identified among input and output signals, additional simple gates, such as INV, NAND, OR gates and so on, may need to be employed to deal with the polarity issues of input signals in a similar fashion to NDT method.

In this simple example, event a+,b+ indicates setting signal *a* to 1 and then waiting for signal *b* at logic high (1). Here *a* is an output and *b* input, which means that DC2 waits for b+. The same method as NDT is used. So a NAND is employed, in which one input comes directly from the signal *b* and the other input comes from the acknowledgement signal (*a+ack*) of a+. After *a+ack* arrives and *b* changes to 1, DC2 can be set. If *a+ack* is active high, no further action should be taken. However, the acknowledgement signal from the SR latch shown in Figure 5.8 (b) is active low, so an inverter is needed to convert active low to active high. In general, two techniques can be used to deal with the polarity issues conveniently. One is slightly modifying the SR latch shown in Figure 5.8 (b) and the other is putting additional simple gates on acknowledgement signals in SR latches. Here we adopt the former method rather than adding an inverter. This modification involves the replacement of the OR gate in the set part of the SR latch in Figure 5.8 with a NOR gate. The modified circuit is shown in Figure 5.18 (b). In fact these two methods have the same functions. When both the acknowledgement signal and the input signal *b* are high, the token can be transferred. The dummy DC waits for signal *b* to become 0. The polarity of both signal *b* and acknowledgement signal of b- is the same. We do not need to modify the SR latch in the reset part. According NDT method, an OR gate is inserted before the dummy DC. The resulting circuit is shown in Figure 5.18 (a). Similar to the result from NDT method, this circuit is SI. The proof will be given in section 5.5.3.

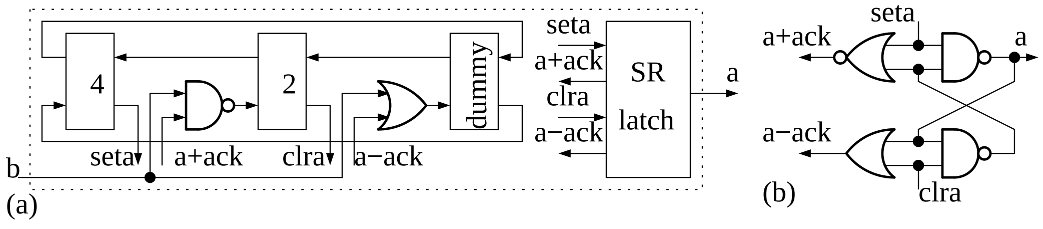


Figure 5.18 (a) An implementation (b) A modified SR latch.

This technique has the same memory element count as the NDT method, with extra simple gate logic added so that it can be used when input and output signals do not form handshake pairs.

It should be noted that savings from this refinement technique can be much greater proportionally if the target circuit mainly contains longer cycles so that not many dummy DCs are needed.

However, the original PNs cannot use this refinement technique directly. From the above discussion, refinement on PN specifications is needed.

5.4.2.3 Systematic refinement of specifications

The discussions in this thesis assume that in a PN specification there are no multiple consecutive input events in any path, i.e. any two input events along a path must be interleaved by at least one output one. This assumption does not affect generality, because at the specification stage, consecutive input signals can be bundled into a single PN event.

The refinement technique uses the fact that some input events may not need to be translated. These input events can be combined with their previous output events. As a result, the places of the input events may be eliminated.

In order to automate the elimination of DCs controlling certain input events, we introduce the following formalization into concrete steps for the first part of the refinement within refinement technique:

1. Convert the original specification (e.g. STG format) to PN format.
2. Modify the PN format specification by removing all removable input events and their places and replacing all un-removable input events and their places with dummy events and places. Note that we define the places which are given tokens after an event happens as the places of this event. In order to keep this refining process simple and easy to automate, only those events which have one output and the output has only one preceding transition ($\bullet$(its output) = 1) can be removed with their places. More complex removable

cases need investigation in the future, which are outside the scope of this thesis. For each removed event, after removing it and its places, in order to keep the specification complete, its next event should be connected with its preceding place. This means that after the previous event has happened, the following events can fire if we do not consider the input events. Here a place not only holds a token but also identifies that its preceding events have been finished. An example shown in Figure 5.19, in which each preceding event of the removed input event has only one successor (input event itself) and the removed input event has only one successor too, is used to illustrate this.

3. Construct the environment of the related control circuit via an explicit representation for the value of each input signal, in the form of a complementary pair of places connected with event transitions. This is the standard PN representation of a binary signal, and is shown in Figure 5.19 (b). The implicit assumption here is that the event transitions are in the environment and controlled by the environment and not the control circuit, so such controls are not drawn in the specification.

4. According to the original specification, connect the results of steps 1 and 2 with consuming or read arcs based on the movement of tokens. For example, for each removed event, in order to match with the original specification, the preceding places of a removed event should carry out functions: 1) providing tokens to the environment to fire the removed event; 2) consuming tokens from the environment after the removed event in the environment has finished; 3) supplying tokens to the next event of the removed event directly. In addition, in the environment, the place of the input event removed from the original specifications has the same functions as the places described above. This is shown in Figure 5.19 (c).

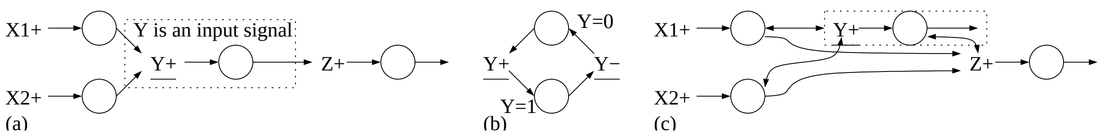


Figure 5.19 A refinement example.

Following these steps, the original specification shown in Figure 5.7 can be changed to a new one shown in Figure 5.20 (a). We will call this type of specification “refined specification” (RS). The pure output signal (event control circuit) part in this new specification is similar to the specification shown in Figure 5.16 (a). A direct translatable specification of the example is shown in Figure 5.20 (b). This RS can be translated into the DC circuit shown in Figure 5.17 and Figure 5.18 after using the refinement technique.

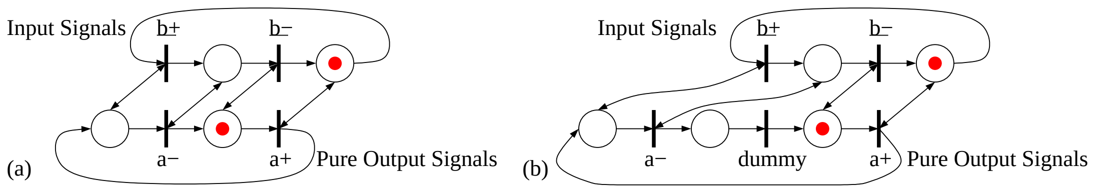


Figure 5.20 Refined specifications (RS).

The refinement technique includes broadly two parts. The first is elimination of unnecessary DCs controlling input events and the second is the fine tuning with additional logic (such as INV, NAND and OR gates and so on) to implement the correct polarity of signals. The first is entirely absent in the NDT method. For the second, the additional logic needed for the NDT method is directly known from the direction of signal changes, while the refinement technique needs much more complex considerations. It has yet to be formalized into algorithmic steps so far.

5.5 Direct translation

5.5.1 The PN2DCs tool

PN2DCs (Petri Net to David Cells) is an automatic tool which can directly translate PN format specifications to DC circuits using a given library. So far no other similar tool has been reported.

This tool is based on the above methods including Varshavsky's, our contributions (a new method and its refinement), and a library defined by ourselves in section 5.5.2. The methods have been introduced above. Here only the input/output format of this tool will be introduced.

5.5.1.1 Input format

The Petrify tool is very popular. Its input is in Signal Transition Graph (STG) format, which is an interpreted PN. Most asynchronous circuit designers are familiar with STGs. In order to make our PN2DCs tool easy to use, an input format similar to that of the Petrify tool is used.

However, this format is used for full PNs rather than STGs because this is demanded by our method.

The following format is defined similar to that of the Petrify tool:

Places and transitions are specified following the key words ".places" and ".transitions" respectively. Places with initial tokens are specified following the key word ".marking". All places and transitions are distinguished by spaces in the specification. Comments are defined following the '#' character. In the ".graph" section, the connections between the places and transitions are specified. For each line in the ".graph" section, the first parameter is a place or transition. Following the place (transition) are a number of transitions (places), which are the successors of the place (transition), connected with the first place (transition) directly. In the case of the first parameter being a place, only one of the following transitions can be fired if the place has a token. In the case of the first parameter being a transition, if the transition fires, each of the following places will receive a token. All of these satisfy the PN definitions.

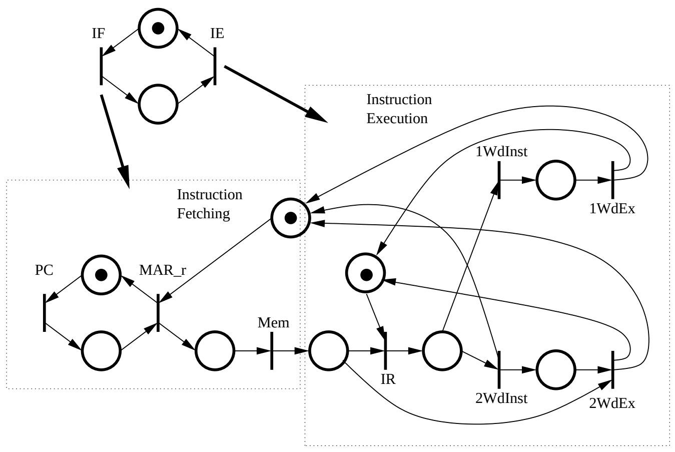


Figure 5.21 An example.

The following is an example input file, whose corresponding PN specification is given in Figure 5.21.

```
.model I_execution

# Declaration of signals

.places p1 p2 p3 p4 p5 p6 p7 p8 p9
.transitions  pc mar_r mem ir t1wdinst t2wdinst t1wdex t2wdex
.marking p1 p4 p5

# Petri net
.graph
p1 pc
pc p2
p2 mar_r
mar_r p1 p3
p3 mem
mem p6
p6 ir t2wdex
ir p7
p7 t1wdinst t2wdinst
t1wdinst p8
t2wdinst p4 p9
p8 t1wdex
t1wdex p4 p5
p4 mar_r
p5 ir
p9 t2wdex
t2wdex p4 p5

# initial marking
#.marking {<p1,pc>}
.end
```

The example is an asynchronous processor specified at the top level. For more details please refer to Chapter 6. In this example, at this level, the behaviour of the processor consists of two actions. One is Instruction Fetching (IF) and the other one is Instruction Execution (IE). They are performed sequentially. After refining, there are nine places and eight transitions. Among the places, places p1, p4 and p5 have an initial token.

So in this input file, after key_word ".places", p1 to p9 are listed. After ".transitions", all abstract events, such as PC, MAR_r, are added. In the ".graph" section, according to the PN specification shown in Figure 5.21, all connections between places and transitions are given.

5.5.1.2 Output format

From the modern circuit design point of view, the synthesis process is only one part of the whole circuit design flow. Digital electronic systems are increasing in complexity over time. This fact, coupled with decreasing product lifetimes and increasing reliability requirements, has forced designers to dramatically increase both the productivity and the quality of their designs [Ashenden 1996]. Generally, the results of the synthesis process cannot be manufactured directly. They should be fed into the next process, which normally is the simulation process, in order to verify the design proposal. This process is very important in modern logic circuit designs.

In order to simulate electronic hardware designs, structural description of the design is required [Coelho 1989]. This is what we consider in this sub-section. Currently most CAD tools support netlist, schematic, or HDL (Hardware Description Languages) as inputs.

Generally all of them can satisfy our requirements. Using schematic input has a friendly interface. However, schematic input is too complex. Auxiliary tools are needed to generate schematic inputs. In addition, different CAD tools require different schematic entry formats. It is difficult to design a universal format which satisfies all CAD tools. Especially, schematic input is not easy to cope with in the next process such as simulation. Normally it should be changed to netlist format.

Netlist input is a text file, in which all signals in a system are named. Following each named signal, all pins and components that this signal is connected to are listed. However, a netlist is unreadable. Generally, it is a good as intermediate format, but not as an independent input format. In our case, PN2DCs is an independent tool. We wish it would be used independently, which means that a friendly interface is expected. So, netlist input is not of interest.

HDLs were developed in response to this requirement. Most existing CAD tools support hardware description languages, such as VHDL and Verilog. VHDL (VHSIC Hardware Description language) has emerged as one of the most important electronic design languages in both the commercial and military electronic design areas. It is one of the most powerful HDLs [Armstrong 1993, Ashenden 1996, Coelho 1989].

An important feature of VHDL is its effectiveness as a design tool. The VHDL language can be used independently from any tools. VHDL provides a wide range of abstraction levels from the behaviour level down to the gate level. From this point of view, VHDL is what we want.

The behaviour and/or functional specification guarantees the correctness of the model, but it does not tell us how to implement the design, especially how to implement SI/DI circuits, because of not having all possible kinds of complex gates in a given library. Generally, synthesis is a translation process from high level to gate level design. As for this high level design, before it can be translated to a gate level solution, it must be translated to an intermediate specification, such as a STG or PN format specification [Blunno 2000a].

Gate level design uses standard off the shelf components. If the timing properties of these components are acceptable, this solution can be manufactured directly. It is suitable as an output format of synthesis processes.

In the PN2DCs tool, we use PNs as the low level specification format to develop asynchronous circuit designs. After synthesis and optimization, we expect to obtain an implementable hardware circuit. The VHDL gate level satisfies our requirements. VHDL has therefore been chosen as the output format of the PN2DCs tool.

The results obtained from PN2DCs are based on a practical library, and employ the low level structure type of design style. The following are definitions of the modelling method:

Definition 5.1 A DC Netlist is a non empty set of wires represented implicitly in a circuit model. The model consists of a non empty set of logic components used in the circuits, and all input and output signals connected with all the components used. This means that the Netlists can be represented using a number of components and signals used in the circuits. Here one element of the set of logic components defined above should belong to the library. Each logic component can be a DC, C-element, Mutex, and general gate, such as AND2, OR2. It is presented in the Netlist using its name. In addition, a DC signal is defined as a wire in the circuits, which connects some logic components directly based on logic circuits. It is presented by using its name, which is a character string containing the instance name of each component connected with the signal and the used pin names of each the component. Each instance name and its pin name are joined together. The instance name is at the beginning and followed by the pin name. They are separated by the character ":". All connected sub-parts are organized as a string in which the output instance name and its pin name are at the beginning. Then, follow is the other sub-parts arranged incrementally (in alphabetical order). All sub-parts are separated by the character "%".

To explain these DC signals clearly, one line in a DC netlist shown in Figure 5.22 is taken as an example.

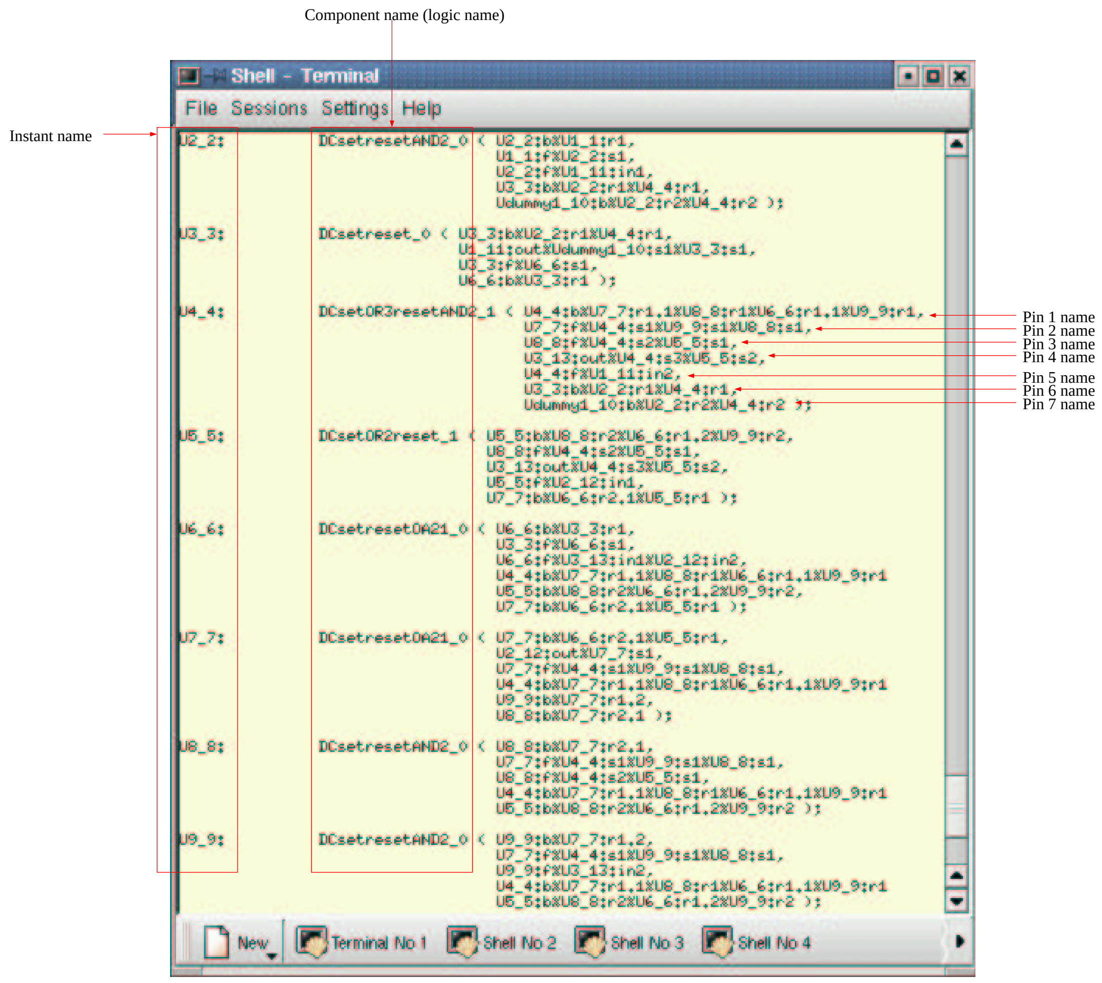
```
U4_4:     DCsetOR3resetAND2_1 ( U4_4:b%U7_7:r1.1%U8_8:r1%U6_6:r1.1&U9_9:r1,
                                U7_7:f%U4_4:s1%U9_9:s1%U8_8:s1,
                                U8_8:f%U4_4:s2%U5_5:s1,
                                U3_13:out%U4_4:s3%U5_5:s2,
                                U4_4:f%U1_11:in2,
                                U3_3:b%U2_2:r1%U4_4:r1,
                                Udummy1_10:b%U2_2:r2%U4_4:r2 );
```

Here the first U4_4 is the instance name of this DC. DCsetOR3resetAND2_1 is its name of this DC. From this name, we know that there exist one backward signal (*bk*), three setting signals (*s1*, *s2*, and *s3*), one forward signal (*fw*) and two resetting signals (*r1*, *r2*). In addition, the initial state (1), setting function (OR) and resetting function (AND) of this DC are known too. The U4_4:b%U7_7:r1.1%U8_8:r1%U6_6:r1.1&U9_9:r1 is the first pin name. All pins of the DCs are arranged in the order part 1, part 2, part 3, and part 4. All pins are separated by the character ','.

- The first part (part 1) is the *bk* signal;
- The second part (part 2) is the list of setting signals. In this part, at least one signal exists. How many signals there are depends on the set function;
- The third part (part 3) is the forward signal;
- The fourth part (part 4) is the list of reset signals, defined in a similar manner as the set part. At least one reset signal is needed. How many depends on the reset function.

The above example is the first pin. It should be signal *bk*. The name stands for the passing path of a signal. All logic components are distinguished by '%'. So this signal passes components U4_4, U7_7, U8_8, U6_6, and U9_9. As we know each component has a number of pins. In order to give a correct path, which pin(s) is connected should be identified. So here pin name b of U4_4, r1.1 of U7_7, r1 of U8_8, r1.1 of U6_6 and r1 of U9_9 are given.

A snapshot of an output file from PN2DCs tool is shown in Figure 5.22. The DC circuit of this example is shown in Figure 6.3.



```
U2_2:          DCsetresetAND2_0 ( U2_2:b%U1_1:r1,
                                  U1_1:f%U2_2:s1,
                                  U2_2:f%U1_11:in1,
                                  U3_3:b%U2_2:r1%U4_4:r1,
                                  Udummy1_10:b%U2_2:r2%U4_4:r2 );

U3_3:          DCsetreset_0 ( U3_3:b%U2_2:r1%U4_4:r1,
                              U1_11:out%Udummy1_10:s1%U3_3:s1,
                              U3_3:f%U6_6:s1,
                              U6_6:b%U3_3:r1 );

U4_4:          DCsetOR3resetAND2_1 ( U4_4:b%U7_7:r1,1%U8_8:r1%U6_6:r1,1%U9_9:r1,
                                     U7_7:f%U4_4:s1%U9_9:s1%U8_8:s1,
                                     U8_8:f%U4_4:s2%U5_5:s1,
                                     U3_13:out%U4_4:s3%U5_5:s2,
                                     U4_4:f%U1_11:in2,
                                     U3_3:b%U2_2:r1%U4_4:r1,
                                     Udummy1_10:b%U2_2:r2%U4_4:r2 );

U5_5:          DCsetOR2reset_1 ( U5_5:b%U8_8:r2%U6_6:r1,2%U9_9:r2,
                                 U8_8:f%U4_4:s2%U5_5:s1,
                                 U3_13:out%U4_4:s3%U5_5:s2,
                                 U5_5:f%U2_12:in1,
                                 U7_7:b%U6_6:r2,1%U5_5:r1 );

U6_6:          DCsetresetOA21_0 ( U6_6:b%U3_3:r1,
                                  U3_3:f%U6_6:s1,
                                  U6_6:f%U3_13:in1%U2_12:in2,
                                  U4_4:b%U7_7:r1,1%U8_8:r1%U6_6:r1,1%U9_9:r1
                                  U5_5:b%U8_8:r2%U6_6:r1,2%U9_9:r2,
                                  U7_7:b%U6_6:r2,1%U5_5:r1 );

U7_7:          DCsetresetOA21_0 ( U7_7:b%U6_6:r2,1%U5_5:r1,
                                  U2_12:out%U7_7:s1,
                                  U7_7:f%U4_4:s1%U9_9:s1%U8_8:s1,
                                  U4_4:b%U7_7:r1,1%U8_8:r1%U6_6:r1,1%U9_9:r1
                                  U9_9:b%U7_7:r1,2,
                                  U8_8:b%U7_7:r2,1 );

U8_8:          DCsetresetAND2_0 ( U8_8:b%U7_7:r2,1,
                                  U7_7:f%U4_4:s1%U9_9:s1%U8_8:s1,
                                  U8_8:f%U4_4:s2%U5_5:s1,
                                  U4_4:b%U7_7:r1,1%U8_8:r1%U6_6:r1,1%U9_9:r1
                                  U5_5:b%U8_8:r2%U6_6:r1,2%U9_9:r2 );

U9_9:          DCsetresetAND2_0 ( U9_9:b%U7_7:r1,2,
                                  U7_7:f%U4_4:s1%U9_9:s1%U8_8:s1,
                                  U9_9:f%U3_13:in2,
                                  U4_4:b%U7_7:r1,1%U8_8:r1%U6_6:r1,1%U9_9:r1
                                  U5_5:b%U8_8:r2%U6_6:r1,2%U9_9:r2 );
```

Figure 5.22 An output file example.

The tool is used after the decomposition and simple specification refinement in our synthesis design flow. So far only the main part has been coded into the tool, whose development is on going and will incorporate the other methods in the future.

5.5.2 Basic components and the given library of PN2DCs tool

The results obtained from synthesis tools should be implemented based on an existing library with a finite number of components. From an automation point of view, this is very important, because it is unreasonable to ask designers to construct logic components from scratch. Some existing synthesis tools have this kind of problem, such as the Petrify tool.

In our synthesis design flow, the decomposition method introduced in section 5.4.1 is used, and it guarantees that a decomposed PN specification can be translated into a DC circuit implemented using a given library with a finite number of components.

The basic components in this library consist of all the components of a normal standard library, such as AND, OR and INV gates, and a set of extension components, such as DCs, Muller C-elements, Mutexes and SR latches and so on. Here we introduce only the extension part of the library.

Definition 5.2 A DCname is the name of a DC (refer to **Definition 2.8** in Chapter 2). It is used to identify the type of the DC in resulting circuits obtained from the synthesis tool. It is a string of characters and the first two letters are "DC", followed by a sub-string "set", the name of the set function of the DC, a sub-string "reset", the name of the reset function of the DC, underscore "_" and the initial state (value 1 indicates presence of a token, value 0 indicates absence of a token) of the DC.

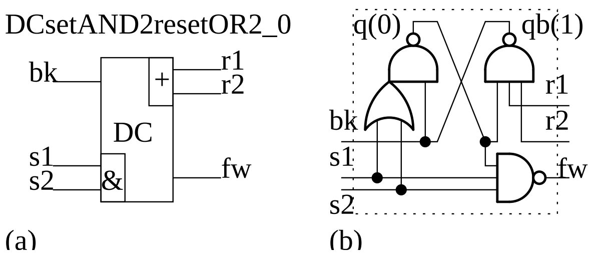


Figure 5.23 An example of DC and its implementation.

An example of this Definition is shown in Figure 5.23, in which there are two set signals, *s1* and *s2*, two reset signals, an *fw* signal, and a *bk* signal. The function of this DC is when both set ginals, *s1* and *s2*, are active (*s1*=0 and *s2*=0), this DC is set up to 1 which indicates a token is presence. When one of reset signals is active (*r1*=0 or *r2*=0), the DC is reset to 0. This means that the token is removed. So the set function is "AND2" and the reset function is "OR2". Initially there is no token in this DC. So the initial state is 0. Finally this DC is named DCsetAND2resetOR2_0. The logic circuit of this DC is shown in Figure 5.23 (b).

The syntax definition is given in Figure 5.24.

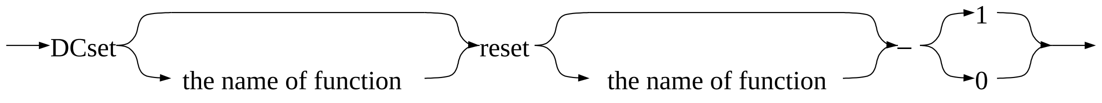


Figure 5.24 The Syntax diagram for Definition 5.3.

Mostly we do not mention the initial states of DCs when we describe them in the context except in the results obtained from the tool. It is assumed that the initial state will be assigned when it is needed based on the situation. Normally, the initial state will be assign to state 0. In some special case such as a DC functioning as a Token, it may be assigned to state 1.

Definition 5.3 The name of a C-element is a string with the sub-string "C_ele" at the beginning. Following it is the fan-in number of the C-element, i.e. C_ele2, C-ele3. (see **Appendix *A***)

Definition 5.4 The string "Mutex" indicates a Mutex with two request inputs and two grant outputs. (see **Appendix *A***)

Because complex specifications can be decomposed into ones with only simple PN basic elements, so that, only one Mutex and two C-elements: C-ele2 and C-ele3, are included. Note that for each C-element there are two outputs, a positive and a negative one. The library also includes a number of DCs.

The following DCs mapped from the basic elements of PNs are needed:

1. Normal DCs, DCsetreset, are used with the linear type of basic element in specifications. The schematic representation of this type of DC is illustrated in Figure 5.25.

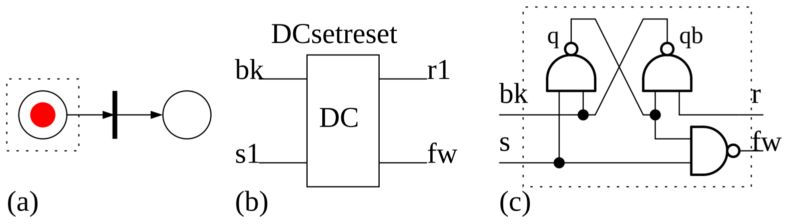


Figure 5.25 The normal DC.

2. A modified DC, DCsetAND2reset, is employed to deal with the event join type of basic element in specifications. This kind of DC is shown in Figure 5.26.

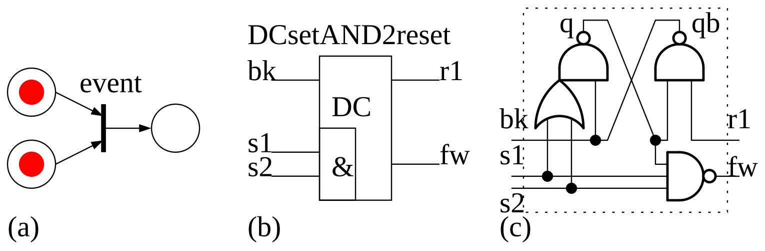


Figure 5.26 A modified DC, named DCsetAND2reset.

Here the event join frame shown in Figure 5.26 (a) is refined by introducing a dummy transition and place shown in Figure 5.27. In this case two events synchronize at the dummy DC as shown in Figure 5.26 (b, c).

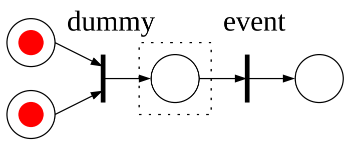


Figure 5.27 A refined event join fragment.

There is another method to realise this event join fragment. This method introduces a C-element and the using the following circuit shown in Figure 5.28 to implement it.

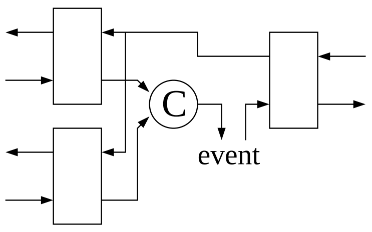


Figure 5.28 An additional implementation.

In this solution two events synchronize at the C-element.

3. Another modified DC, DCsetOR2reset, is employed. An exclusive join type of basic element in specification can be translated to this type DC which is shown in Figure 5.29.

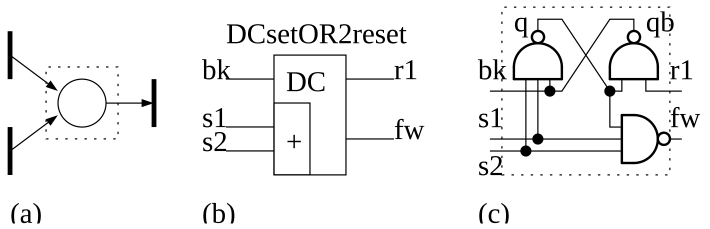


Figure 5.29 A modified DC, named DCsetOR2reset.

4. A third modified DC, DCsetresetAND2, is used with the event fork type of basic element in specifications. A schematic DCsetresetAND2 is shown in Figure 5.30.

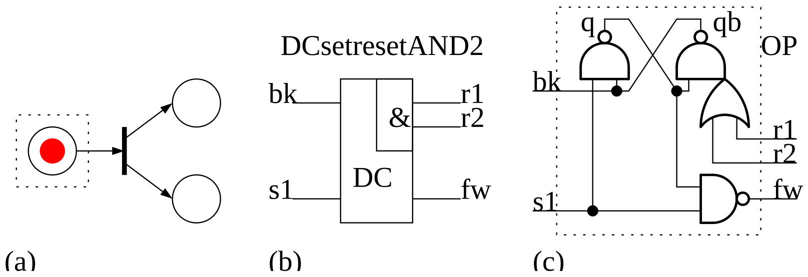


Figure 5.30 A modified DC, named DCsetresetAND2.

5. A fourth modified DC, DCsetresetOR2, is used. It can be used with the alternative fork type of basic element in specifications. The schematic representation of DCsetresetOR2 is given in Figure 5.31.

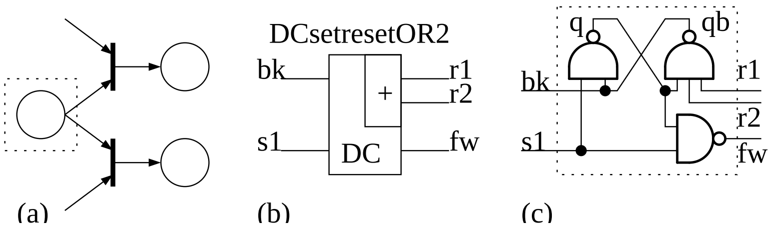


Figure 5.31 A modified DC, named DCsetresetOR2.

Note: we do not give an implementation solution for free-choice fragment. In Chapter 3 we showed that it can be implemented by using Mutexes.

The above DCs can be used to cope with 5 types of basic PN elements. For more complex specifications, other complex DC types, such as DCsetAND2resetAND2, DCsetAND2resetOR2, DCsetOR2resetAND2 and DCsetOR2resetOR2 which are shown in Figure 5.32, Figure 5.33, Figure 5.34, and Figure 5.35, are provided in the library. They are used after optimization to deal with combining event join and event fork, combining event join and alternative fork case, combining the exclusive join and event fork case and combining the exclusive join and alternative fork case specification respectively.

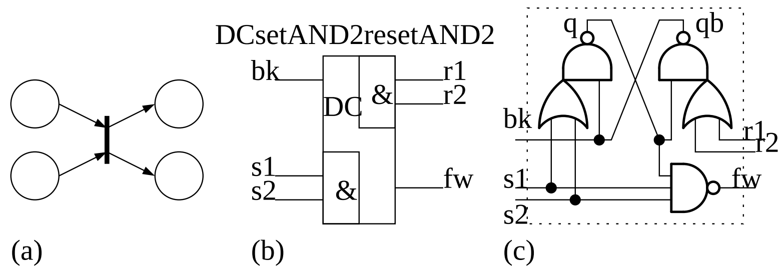


Figure 5.32 A DCsetAND2resetAND2.

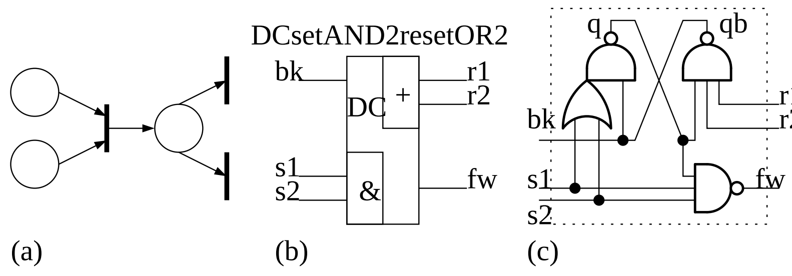


Figure 5.33 A DCsetAND2resetOR2.

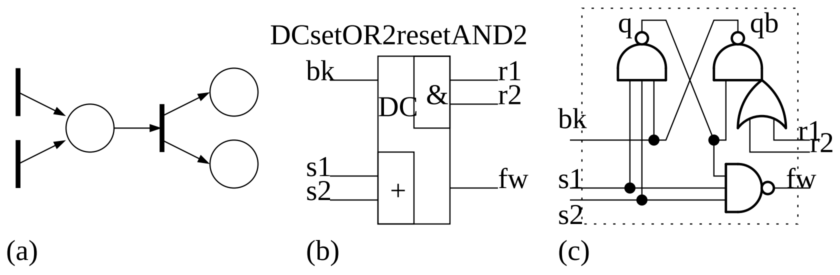


Figure 5.34 A DCsetOR2resetAND2.

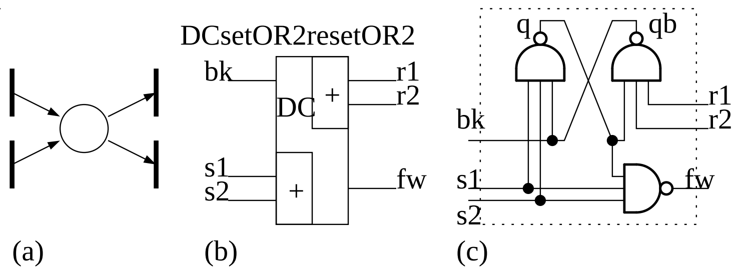


Figure 5.35 A DCsetOR2resetOR2.

The above DCs are adequate to implement the specifications because decomposition is used before translation. However in order to provide more convenience, some DCs with three or four inputs at set and reset functions are also supplied in this library. In addition, some slightly more complex DCs, such as DCsetOA21resetOA21, DCsetAO21resetAO21 and so on, are included in the library too. Here "OA" stands for OR-AND gates. "AO" stands for AND-OR gates. Apart from these, their VDC (see section 5.6.2) counterparts are also supplied in this library. All (V)DCs in the library of the PN2DCs tool are listed in Appendix *A*.

Apart from the above components, such as C-ele2, C-ele3, Mutex, DCs, memory elements, such as SR latches, are also provided in this library.

5.5.3 Proof of SI solutions

Because of the advantages of SI circuits, most circuits in this thesis are designed as SI ones. Here, we present some, rather informal, proofs to guarantee that the circuits designed using our methods are SI.

The classic SI definition in Chapter 2 (**Definition 2.7**) is based on state diagrams, not on signal transition graphs (STGs), because classical asynchronous circuit designs usually describe circuit behaviour using a state diagram. Hence the cardinality of the input set, the number of states, is exponential in the number of signals. The first condition ensures that the circuit is live; the second condition ensures that the circuit is deterministic; and the third condition ensures that if a transition takes longer to complete than other concurrent transitions, it will still be enabled before it is actually fired.

We use PN specifications as the inputs of the PN2DCs tool. To facilitate the proofs, we convert these conditions defined on state diagrams to ones based on PNs.

The sufficient conditions for each of the three SI conditions on an STG can be easily derived, based on the fact that a circuit state diagram is the interpreted reachability graph of an STG. One sufficient condition for the third condition in **Definition 2.7** is that the corresponding STG is live. A live STG guarantees the existence of a state assignment for each marking. The token marking mechanism on an STG asserts that the enabling token will stay unconsumed on the enabling arcs until the enabled transition is actually fired, no matter which state (marking) the circuit might be in; hence, an enabled transition will remain enabled in the second state if not fired in the first state, satisfying the third SI condition. Therefore, any live STG, if each state can be uniquely assigned, represents a SI circuit by definition [Beerel 1991].

From the above discussion, we know that specifications based on state diagrams can be easily transferred to ones based on STGs and STGs are interpreted PNs. So we can use the conditions on STGs to prove the validity of our circuits.

Here, the assumption is that PN specifications are provided by designers. They are required to supply suitable PN specifications which must be live and 1-safe. In this kind of PN, all transitions can be taken as operations (including empty operations). An example is shown in Figure 5.36.

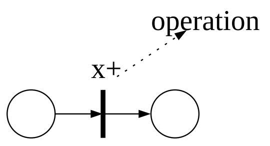


Figure 5.36 A PN example.

Each operation can be implemented on an element of controlled logic circuits, i.e., the binary signal *x* shown in Figure 5.36. At every reachable marking of the PN at most one transition of *x* is enabled, and the operation on *x* is consistent with its current state. For example, if *x* = 0, then *x*-, and if *x* = 1, then *x*+ [Semenov 1997a].

In order to avoid deadlock, at least three places in each loop in PNs are required [Varshavsky 1996].

However, although the specification should guarantee that an SI circuit implementation should be possible based on the above requirements, whether the final circuit is SI depends on the implementation details because decomposition is used in implementations.

From a circuit consideration, an SI circuit is one which operates correctly regardless of gate delays; wires are assumed to have zero or negligible delay [see Chapter 2].

According to this, we model a circuit using the following method to prove it is SI:

1. Inserting a delay component on the fan-out of each gate. The sum of the gate delay and the appropriate wire delay are assigned to the delay component. Note that all the delays in delay components are finite and in the range (0, ∞);
2. Ignoring all delays on all logic gates and wires. In other words all delays on the logic gates and wires are treated as 0.

If the circuit still works correctly no matter how long the delay components are as long as they are finite, the circuit is SI.

A schematic representation of this modelling method is illustrated in Figure 5.37.

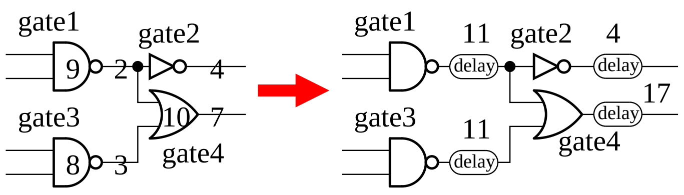


Figure 5.37 SI modelling method.

In Figure 5.37, number 9 on the left side identifies the delay of gate1. Number 2 identifies the delay of the wire connected between gate1 to gate2 and gate4. The same meanings are assigned to the other numbers. To model this circuit shown on the left in Figure 5.37, we add a delay component for each gate shown on the right in the same Figure. Using a delay component replaces the appropriate gate and wire delays. So all logic gates and wires are treated as zero delay in this new model. The delay on the delay components is unbounded but finite.

From [Varshavsky 1996], we know that the control circuit constructed by using the simplest DCs is SI. This is because (see Figure 2.11 (b)) they are connected by using handshake protocols. This can easily be deduced from the STG specification shown in Chapter 2 (Figure 2.12). NDT is an extension of Varshavsky's method. It introduces many complex DCs, such as DCsetAND2restOR2 and so on, and allows insertion of operation events. However, all components are still connected according to handshake protocols (request/acknowledge). So if all components used in the circuits are proven to be SI, the circuits are SI.

In our case, the circuit is built using DCs, C-elements, Mutexes, SR latches and some other simple standard gates, such as NAND gate, OR gate and so on. So firstly we need to prove that all components are SI. Here we use the simplest DC as an example, in which signals *s* and *r* are inputs and signals *fw* and *bk* are outputs. Additionally, a pair of complementary states, *q* and *qb*, exist in this DC. According to the SI modelling method, the simplest DC will be modelled by inserting delay components for all gates as shown in Figure 5.38.

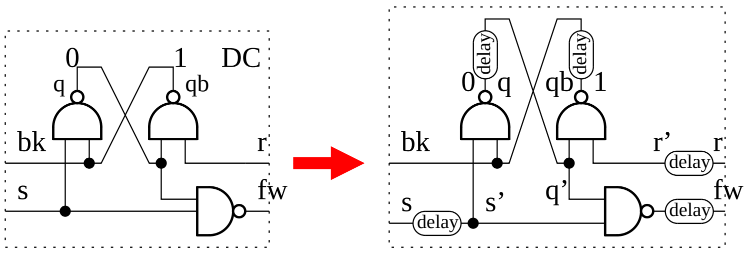


Figure 5.38 An example of SI model.

In order to prove that this DC is SI, apart from three delay components added for the three NAND gates, two additional delay components are added on the input signals, *s* and *r*.

The original STG specification for this DC (Figure 5.38) is shown in Figure 5.39.

$$\begin{tikzcd}
\underline{s-} \arrow[r] & q+ \arrow[r] & qb- \arrow[r] & bk- \arrow[r] & \underline{s+} \arrow[r] & fw- \arrow[d] \\
\underline{r+} \arrow[u, "\bullet"] & fw+ \arrow[l] & q- \arrow[l] & bk+ \arrow[l] & qb+ \arrow[l] & \underline{r-} \arrow[l]
\end{tikzcd}$$

Figure 5.39 An STG of the simplest DC.

After SI modelling, we obtain a new STG specification, which is shown in Figure 5.40.

$$\begin{tikzcd}
\underline{s-} \arrow[r] & s'- \arrow[r] & q+ \arrow[r] & q'+ \arrow[r] & qb- \arrow[r] & bk- \arrow[r] & \underline{s+} \arrow[r] & s'+ \arrow[r] & fw'- \arrow[r] & fw- \arrow[d] \\
r'+ \arrow[u] & \underline{r+} \arrow[l] & fw+ \arrow[l] & fw'+ \arrow[l] & q'- \arrow[l] & q- \arrow[l] & bk+ \arrow[l] & qb+ \arrow[l] & r'- \arrow[l] & \underline{r-} \arrow[l]
\end{tikzcd}$$

Figure 5.40 An STG specification.

From this new STG specification, we know that no matter how long the delays inserted are, the circuit works correctly. For example, signal *s* is an input. After s-, no matter how long the delay on this signal (as long as it is finite), only s- passes the delay component, a NAND gate senses s'- and then q+ happens. The same for q+ to qb- (bk-), s+ to fw-, r- to qb+ (bk+), qb+ (bk+) to q-, and q- to fw+. The bk- to s+, fw- to r-, fw+ to r+, and r+ to s- transitions are decided by the environment. So this DC is SI itself.

The simplest DC is SI. Next we prove the modified DCs are SI. We use the DCsetAND2resetAND2 shown in Figure 5.32 as an example. The SI model for DCsetAND2resetAND2 and the STG of this DC are shown in Figure 5.41.

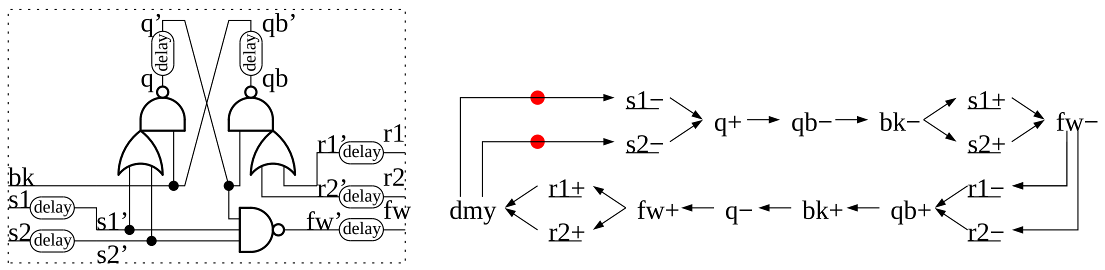


Figure 5.41 An example to prove it SI.

The function of this DC is when both setting inputs are active (s1- and s2-), the token will be transferred in this DC from its previous DCs and then signal *bk* will be generated to reset all previous DCs (remove all tokens and withdraw all setting input signals). After that, because this DC holds the token and the two active input signals have been withdrawn, signal *fw* happens (fw-) to fire related events and then to transfer the token. This is followed by the reset signals resetting this DC and this DC withdrawing *fw* (fw+). From the STG of this DC shown in Figure 5.41, it is easy to understand its function.

The same method is used to model this DC. The STG specification of the SI model is shown in Figure 5.42.

$$\begin{tikzcd}
 & \underline{s1-} \arrow[r] & s1'- \arrow[dr] & & & & & \underline{s1+} \arrow[r] & s1'+ \arrow[dr] & & \\
 & & & q+ \arrow[r] & q'+ \arrow[r] & qb- \arrow[r] & bk- \arrow[ur] \arrow[dr] & & & fw'- \arrow[r] & fw- \arrow[dd] \arrow[dddd] \\
 & \underline{s2-} \arrow[r] & s2'- \arrow[ur] & & & & & \underline{s2+} \arrow[r] & s2'+ \arrow[ur] & & \\
 & r1'+ \arrow[dl] & \underline{r1+} \arrow[l] & & & & & & & r1'- \arrow[dl] & \underline{r1-} \arrow[l] \\
dmy \arrow[uuuur, "\bullet"] \arrow[uur, "\bullet"] & & & fw+ \arrow[ul] \arrow[dl] & fw'+ \arrow[l] & q'- \arrow[l] & q- \arrow[l] & bk+ \arrow[l] & qb+ \arrow[l] & & \\
 & r2'+ \arrow[ul] & \underline{r2+} \arrow[l] & & & & & & & r2'- \arrow[ul] & \underline{r2-} \arrow[l]
\end{tikzcd}$$

Figure 5.42 An SI model STG specification.

The input setting signals, *s1* and *s2*, synchronize at the left ON21 gate when both of them are active (*s1*=0 and *s2*=0). Only then can the transition fire, no matter how long the delays from *s1* to *s1'* and from *s2* to *s2'* are. After that, q+, qb-, and bk- happen sequentially. We can take them as one-hot because each time only one signal changes. So the above transitions from s1- and s2- to bk- are SI. The environment should guarantee this after bk-, s1+ and s2+ happen. We do not care whether they happen at the same time. This DC can synchronize them at the NAND gate. From this circuit, only when signals, *q*, *s1'* and *s2'* (here *s1'* is the signal *s1* after a delay component and *s2'* is the signal *s2* after a delay component), are 1, fw- happens. Before *s1'* and *s2'* become 1, they are all 0. So here logic 1 is expected for them. No matter how long the delays are, the NAND gate expects all inputs to be logic 1. After fw-, this DC should be waiting for both r1- and r2-. This DC satisfies this requirement owing to the right ON21 gate shown in Figure 5.41 no matter how long the delays are at the resetting inputs. After that, this DC removes the token and withdraws the *fw* signal. All of these can be taken as one-hot behaviour. After fw+, the environment provides both r1+ and r2+. However the ON21 gate responds to change on only one of them. Fortunately, the "length of three" rule of constructing DC circuits can guarantee that before starting the next cycle, the signals *r1* and *r2* must have been changed to 1. So this DC is SI. The same proof can be applied to all modified DCs.

C-elements are asynchronous components. Only when all inputs are active is the output active. Otherwise the output keeps the old value. Adding C-elements to SI circuits therefore does not make them non-SI.

The following will prove that circuits with Mutexes are SI. Only a free-choice and an arbitration-choice fragment need Mutexes to construct DC circuits here. This kind of fragment belongs to the type of alternative fork fragment. The PN specification and implementation of this fragment are shown in Figure 5.6 (b). However, which branch is taken depends on the other conditions; it is not strictly free-choice. In this case, one of two branches is selected randomly. The PN specification is shown in Figure 5.43.

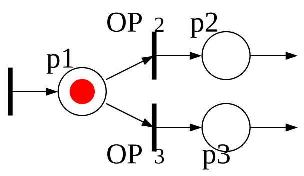


Figure 5.43 A free-choice fragment.

We propose a circuit which is shown in Figure 5.44 to implement this PN fragment.

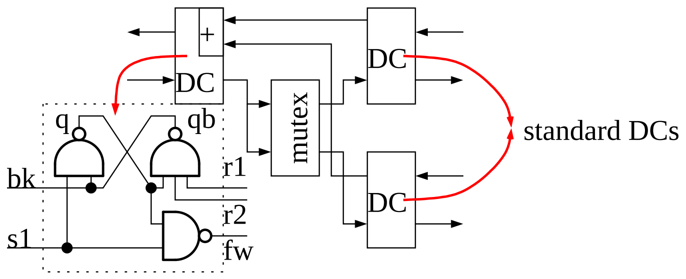


Figure 5.44 An implementation.

We have proven that all DCs are SI. The *fw* signal of the DC mapped from place p1 goes to two inputs of the Mutex at the same time. Which branch is chosen depends on the grant signal of the Mutex. Only one branch can be selected directly during each cycle because of the 1-safe PN specification. After the appropriate operation finishes, the *fw* signal will be withdrawn, and then the grant signal will be withdrawn too. This means that the choice of operation depends on the *fw* signal. Only when the *fw* signal arrives, does the appropriate operation work. After the operation finishes, the *fw* signal will be withdrawn and then start the next operation. So this circuit is SI.

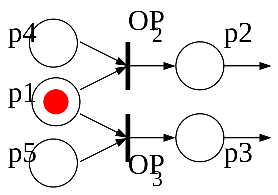


Figure 5.45 An arbitration-choice fragment.

However, free-choice without inputs does not make much sense. A very popular PN fragment is not free-choice but arbitration-choice, which is shown in Figure 5.45.

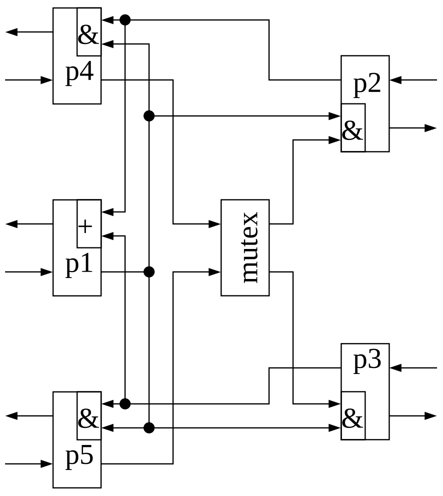


Figure 5.46 An implementation of the arbitration-choice fragment.

We proposed a circuit which is shown in Figure 5.46 to implement this PN fragment. Using the above methods, it is easy to prove this circuit is also SI.

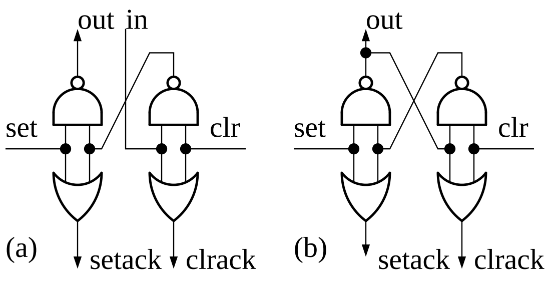


Figure 5.47 SR latches with completion detection.

SR latches with completion-detection logic are another type of component in the library. A simple example of such circuits is shown in Figure 5.47.

In Figure 5.47, (a) is used when handshake signals are distinguished explicitly. Otherwise (b) is used.

After modelling these latches using the SI modelling method, under the reasonable assumption that signals *set* and *clr* cannot happen at the same time (which can be guaranteed by inserting a DC between these two signals), they can be shown to be SI. Only when signal set arrives (maybe the SR latch has been waiting for long time due to delays), the output signal will be set. Furthermore only when the normal SR latch settles down, the completion (*ack*) signal happens. The same is true for the *clr* signal. So from this point of view such circuits are SI.

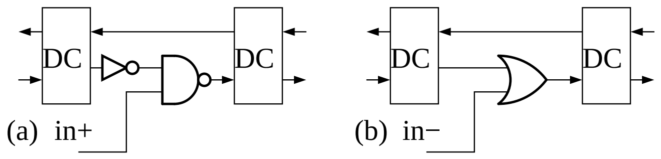


Figure 5.48 Basic models.

As mentioned before, when directly translating low level specifications, in order to produce concise circuits, sometimes a number of simple gates are employed in DC circuits. This is because there exist input signals which are triggered and held in the environment and not in the control circuit (see section 5.4.1, NDT method). The basic model of this kind of circuit is shown in Figure 5.48, in which an input signal from the environment goes to the simple gate. From the specification, we know that each input signal should be set and reset once in each operation cycle. In addition, set and reset operations should appear as a pair. Furthermore, once the input signal changes, it will remain stable in the period between two DCs, because of the properties of DCs, the operations before the DC and after the DC cannot happen at the same time. So only the signal from the previous DC can withdraw the setting input signals. Informally, this signal from the previous DC can be taken as one-hot signal. So inverters can be inserted without affecting SI.

Now we have proven that all components used to construct circuits are SI. Because all connections between components are handshake protocols, the circuits built using this method are SI.

5.6 Optimization

Working SI circuits can be generated after the above decomposition, refinement and direct translation process. However, such SI circuits tend to be large and slow. This is because the specification refinement process cannot normally manage to eliminate all memory element redundancies.

Therefore, there is usually scope for further optimization after the direct translation step. A two-part optimization process, which specially target resulting circuits from the NDT step, will be described in this section.

5.6.1 Reducing the number of DCs

As introduced in the refinement technique, so far only those input events which have one output place and $\bullet$(the output place) = 1 and their places can be removed. According to the NDT method, there should be a large number of memory elements in a DC circuit.

In practice, two closed transitions have always a sequential relationship. After one fires, the other one should fire in turn. Apart from this, they do not share hardware resources. A simple example for this case is shown in Figure 5.49.

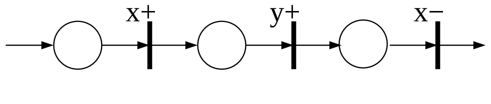


Figure 5.49 A simple example.

This is an incomplete STG; it is just a fragment of a PN specification. Generally, we can optimize the circuit obtained from the NDT method using the following method.

Both signals *x* and *y* are outputs. After x+ happens, y+ should be fired. Except for the ordering, they do not have any relationship. Using the NDT method to realize this fragment, three DCs are needed. The schematic diagram is shown in Figure 5.50.

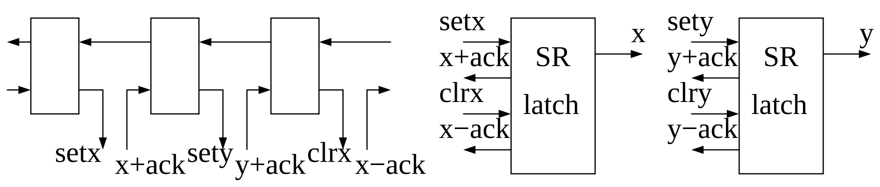


Figure 5.50 The direct solution from the techniques 2-3.

In fact the DC between x+ and y+ is not necessary, because x+ and y+ are independent from the implementation point of view. After *x* is set, setting *y* can be implemented immediately rather than by first withdrawing the *x* setting signals and then set *y* sequentially. It does not affect the functionality of the fragment. Ideally, if several not related transitions (events) are connected, more DC reduction is possible. Only between complementary events a DC is used to isolate them. Using this method we can make the circuit simpler and faster than its original one obtained from the NDT method. The following circuit shown in Figure 5.51 results after optimization using this method.

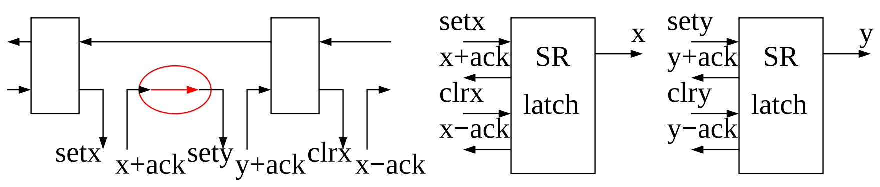


Figure 5.51 The optimization circuit.

In Figure 5.51, the signal *sety* comes directly from signal *x+ack*. In this circuit, one DC is removed. More memory elements mean more delays, so the optimization is advantageous. Most of this kind of DC can be deleted in optimization.

5.6.2 Introduce VDC to further reduce memory elements

During the specification refinement step, each transition is mapped onto an event which is implemented by using an SR latch (normally used to keep an output variable). A large number of this kind of memory element should exist in a circuit obtained by direct mapping.

In fact, the DCs used so far act purely as clock signals. In other words, their only function is to control events. However, since they are memory elements anyway, it may be possible to use some of them to store the values of output variables in addition to performing their event controlling duties. This is because in some cases, the internal pair of latched signals in a DC may be made to coincide with the value of an output variable. This may lead to a reduction in the number of SR latches needed for the storage of output variables.

Definition 5.5 Variable-holding DCs (VDCs) are DCs in which the normal state signals, *q* and *qb*, are used as output variables to the DC's environment rather than just the internal state variables. In other words, the state variables can be supplied to the environment and connected to the DC's environment via direct wires.

VDCs can be used to perform both output variable holding and clock functions. In order to make effective use of VDCs to store output variables, complementary output events, such as a+ and a- in this example, need to be grouped together so that a single VDC can be used to hold variable *a*. So the first step of this method is the identification of likely output variables which might be held in VDCs instead of dedicated SR latches.

We again use the same example shown in Figure 5.7 to explain the optimization. Having identified likely candidate variables, in this case the variable *a*, a VDC must be configured to hold each of them. This, however, requires a DC with non-simple set and/or reset functions, unlike the DCs used so far in NDT method for this example. This is because a DC, if used purely to control events, does not have to maintain its *q* and *qb* based on the value of any output variable. Therefore the set and reset functions can be made simple so that *q* and *qb* have meaningful values only used during the token propagation. For a VDC that needs to hold the value of an output variable, however, the set and reset functions must include additional information so that *q* and *qb* are changed for the dual purpose of token propagation and variable holding.

In this example, a+ should happen after b-. So the set function of the VDC holding variable *a* (VDCa) should include both a term from its token propagation duty and a term from variable *b*, i.e. *b*=0 activates. b+ should happen after a+, but this event is controlled by the environment and therefore does not affect the configuration of VDCa. a- should happen after b+. This may be implemented by configuring the reset function of VDCa appropriately. However, because *b* is controlled by the environment, it is safer to implement this handshake by modifying the set function of the DC following VDCa. This is because in a DC circuit, each DC contributes to the SI of the overall circuit by maintaining a correct relationship between signals *s* and *bk* provided its subsequent stages manage to maintain SI between its *fw* and *r* signals. Introducing signals controlled by the environment to the reset function of any DC may therefore introduce errors if the environment fails to respond at the right speed.

Against this background, it is better to modify the set function for the DC(s) to follow the VDC to achieve the same handshake functionality and preserve SI without assumptions about the environment. In this case, the set function of the DC following VDCa can be modified so that it includes b+. This guarantees that VDCa will not be reset until b+ has happened.

The next step is to plug the resulting VDC and any other DCs modified during the previous step into an overall DC circuit implementing the specification. This produces the implementation shown in Figure 5.52 (a) for the current example. There are only three memory elements in this new implementation.

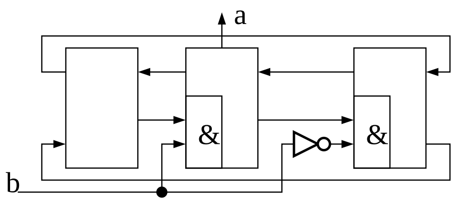


Figure 5.52 An implementation based on the special optimization idea.

In this example there is only one pair of handshake variables. This makes VDC easy to apply. With more handshake variables, they need to be partitioned based on their relationships. The optimization method in its current form is not mature and cannot be automated. Because of the obvious advantages in cost and performance improvements it may bring, it is a worthwhile subject for future investigation.

In addition, VDCs open up potentials for further optimization. Based on this method we can obtain a fast relative timing (RT) circuit. RT techniques have become popular recently [Stevens 1999].

In the resulting circuits obtained using the VDC optimization method, for example the circuit shown in Figure 5.52 (a), we have made clear the set and reset functions of each output signal. With this information, a circuit based on traditional SR latches and not DCs can be constructed. Each output signal is implemented by using a traditional SR latch. The set and reset function of this SR latch consist of the state variables and appropriate input signals. The schematic diagram of the method is illustrated in Figure 5.53 (a).

The output signal *a* is set when the input signal *b* is 0, and a is reset when b is 1. Because this is a very simple example, state variables are not needed. So a simple circuit results, shown in Figure 5.53 (b).

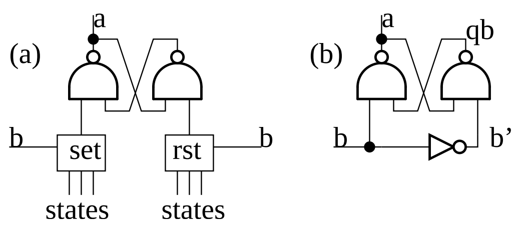


Figure 5.53 An example.

This circuit is not SI. It is correct under certain timing assumptions. Here the assumptions are that a+ should be earlier than qb- and qb- should be earlier than b+. Furthermore, qb- should come earlier than b-. Signals *a* and *qb* are internal ones. Signal *b* is from the environment and it is fired by *a*. These assumptions are reasonable if the environment does not take zero time to respond.

5.6.3 Special optimization method

Because there is only one pair of handshake variables in the example shown in Figure 5.7, a pair of DCs can simply control this event sequence without additional circuitry. Figure 5.54 presents an implementation solution for this example using this method. Since this method does not have a wide application, it is just mentioned here to show that special circumstances in the specification may allow clever tricks to produce minimal results which a generalized tool based on DCs may never be able to find.

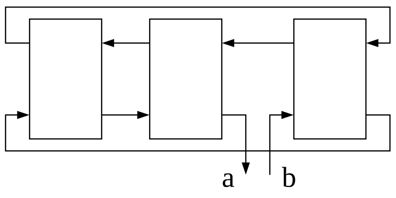


Figure 5.54 An implementation of this special case.

5.7 Conclusions

In this Chapter, a new direct translation synthesis method was reported, and based on this method an automatic synthesis tool (PN2DCs) was developed.

This new synthesis method is based on Varshavsky's method and uses DCs as building blocks to implement asynchronous circuits. In this chapter, we presented a formal definition for DCs and developed a DC library. In addition, we also mentioned that DCs can function as general clock signals, which can make asynchronous design less difficult.

Compared with the other synthesis methods which depend on encoding of the state space using abstract variables, this new direct translation method based on DCs has many advantages. It can be used to translate high (abstract) level to low (signal) level specifications to DC circuits, overcoming the disadvantages in Varshavsky's method. In addition, it provides SI without resorting to assumptions about the speed of devices such as the inverter, or about the existence of a large complex gate library.

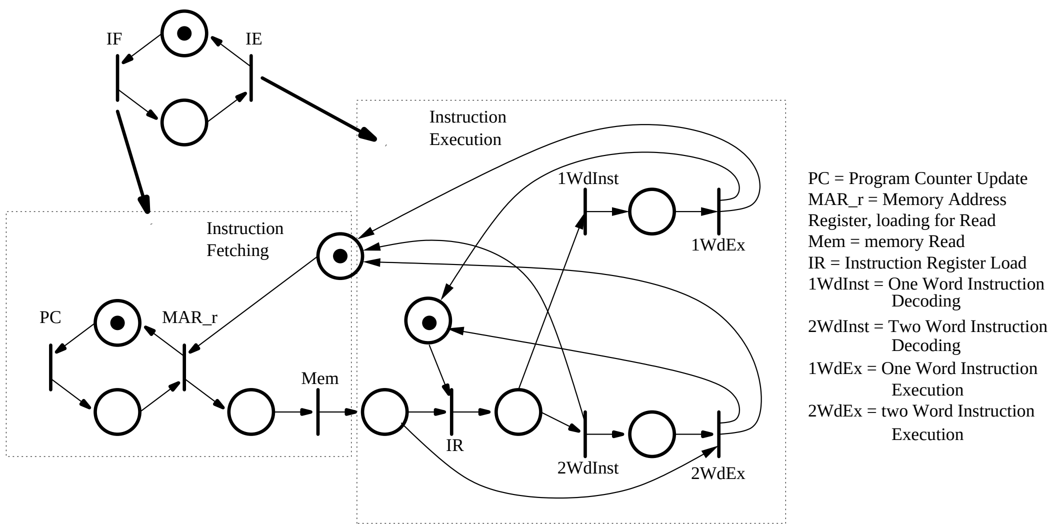
In order to generate a good performance and low cost circuit, systematic decomposition, refinement and optimization techniques were also introduced in this chapter.

All of these build up a sound foundation of a completely automated asynchronous synthesis tool.

Chapter 6: Case Studies

Asynchronous event control systems can be specified at various levels from abstract level to gate level. From the design point of view, the lower the more difficult. As mentioned in Chapters 2 and 5, synthesis tools can be used to translate high level specifications to gate level circuits. In this Chapter, we will demonstrate how to design asynchronous systems using the PN2DCs tool.

6.1 Asynchronous processor



IF
IE
Instruction Execution
Instruction Fetching
PC
MAR_r
Mem
IR
1WdInst
1WdEx
2WdInst
2WdEx
PC = Program Counter Update
MAR_r = Memory Address Register, loading for Read
Mem = memory Read
IR = Instruction Register Load
1WdInst = One Word Instruction Decoding
2WdInst = Two Word Instruction Decoding
1WdEx = One Word Instruction Execution
2WdEx = two Word Instruction Execution

Figure 6.1 An asynchronous processor specified at high level.

PN2DCs can be used to cope with various level specifications from high abstract level to low signal level. In this section, we will show how to design an asynchronous processor using our PN2DCs tool. This example is from [Semenov 1997c] and it is specified at an abstract level which is shown in Figure 6.1.

At the high design level the behaviour is defined in terms of an asynchronous process that can be represented by a labelled PN (LPN). The transitions of such a net can be labelled with the names of relatively abstract operations in the datapath and/or control components. In this example, at the top abstraction level, the behaviour of a processor consists of two actions, Instruction Fetching (IF) and Instruction Execution (IE), which alternate and are therefore performed sequentially.

The processor is very simple. However, we cannot obtain other useful meanings from this top high level specification, and so, we cannot derive an implementation.

In order to make circuit design possible, we can refine these actions into sub actions according to our ideas about the processor architecture. Thus, the IF action can be seen as a process, i.e. a PN fragment, consisting of the following sub actions: incrementing a Program Counter (PC), loading a Memory reading register (MAR_r), and reading the new instruction word from Memory (Mem). The IE action can be refined into a process (another PN fragment) involving other sub actions: loading an Instruction Register (IR), and decoding, activating and executing the fetched instruction for two possible instruction formats, a one word instruction (1WdInst and 1WdEx) and a two word instruction (2Wdinst and 2WdEx). The part of the process concerned with the two word instruction execution requires two memory cycles. As can be observed from the analysis of this PN, the initial sequential operation between IF and IE has been refined into a model which allows concurrency between actions with smaller granularity. For example, the PC action can be executed concurrently with instruction reading, decoding and execution. Another paradigm appearing at this level is that of choice between two types of instruction execution. The process of refining the design can be continued until the designer realizes that the abstract behavioural model satisfies the desired functional and quantitative requirements. The result of this design stage is a specification of the control flow in such a form that its actions, i.e. transitions in the labelled PN model, can be easily mapped onto the primitive operations of the datapath units. More details of this asynchronous processor can be found in [Semenov 1997c].

This refined asynchronous processor is implemented by using DCs and it is based on the following model shown in Figure 6.2,

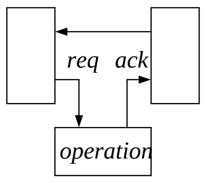


Figure 6.2 The model of DC circuits.

in which an operation is inserted between two DCs. This operation fires by the *req* signal from the DC control circuits. After it is finished, an *ack* signal will be given to the DC control circuits. Using the direct translation method introduced in Chapter 5, each place was translated to a DC and each transition was mapped to an operation which is inserted between two relevant DCs, in the middle of the request/acknowledgement handshake.

We generate an input file for the tool from the above LPN specification shown in Figure 6.1. In Chapter 5, we have introduced the input format, which is similar to the input file of the Petrify tool. As for this example, the input is shown as follows:

```
.model I_execution
# Declaration of signals
.places p1 p2 p3 p4 p5 p6 p7 p8 p9
.transitions  pc mar_r mem ir t1wdinst t2wdinst t1wdex t2wdex
.marking p1 p4 p5
# Petri net
.graph
p1 pc
pc p2
p2 mar_r
mar_r p1 p3
p3 mem
mem p6
```

```
p6 ir t2wdex
ir p7
p7 t1wdinst t2wdinst
t1wdinst p8
t2wdinst p4 p9
p8 t1wdex
t1wdex p4 p5
p4 mar_r
p5 ir
p9 t2wdex
t2wdex p4 p5
# initial marking
#.marking {<p1,pc>}
.end
```

The meaning of each part of this file has been explained in Chapter 5. After feeding this input file to PN2DCs tool, we can obtain a result, specified in VHDL, which is as follows:

```
U1_1:    DCsetreset_1 ( U1_1:b%Udummy1_10:r1,
                        Udummy1_10:f%U1_1:s1,
                        U1_1:f%U2_2:s1,
                        U2_2:b%U1_1:r1 );
U2_2:    DCsetresetAND2_0 ( U2_2:b%U1_1:r1,
                            U1_1:f%U2_2:s1,
                            U2_2:f%U1_11:in1,
                            U3_3:b%U2_2:r1%U4_4:r1,
                            Udummy1_10:b%U2_2:r2%U4_4:r2 );
U3_3:    DCsetreset_0 ( U3_3:b%U2_2:r1%U4_4:r1,
                        U1_11:out%Udummy1_10:s1%U3_3:s1,
                        U3_3:f%U6_6:s1,
                        U6_6:b%U3_3:r1 );
U4_4:    DCsetOR3resetAND2_1 ( U4_4:b%U7_7:r1.1%U8_8:r1%U6_6:r1.1%U9_9:r1,
                               U7_7:f%U4_4:s1%U9_9:s1%U8_8:s1,
                               U8_8:f%U4_4:s2%U5_5:s1,
                               U3_13:out%U4_4:s3%U5_5:s2,
                               U4_4:f%U1_11:in2,
                               U3_3:b%U2_2:r1%U4_4:r1,
                                    Udummy1_10:b%U2_2:r2%U4_4:r2 );
U5_5:      DCsetOR2reset_1 ( U5_5:b%U8_8:r2%U6_6:r1.2%U9_9:r2,
                             U8_8:f%U4_4:s2%U5_5:s1,
                             U3_13:out%U4_4:s3%U5_5:s2,
                             U5_5:f%U2_12:in1,
                             U7_7:b%U6_6:r2.1%U5_5:r1 );
U6_6:      DCsetresetOA21_0 ( U6_6:b%U3_3:r1,
                              U3_3:f%U6_6:s1,
                              U6_6:f%U3_13:in1%U2_12:in2,
                              U4_4:b%U7_7:r1.1%U8_8:r1%U6_6:r1.1%U9_9:r1,
                              U5_5:b%U8_8:r2%U6_6:r1.2%U9_9:r2,
                              U7_7:b%U6_6:r2.1%U5_5:r1 );
U7_7:      DCsetresetOA21_0 ( U7_7:b%U6_6:r2.1%U5_5:r1,
                              U2_12:out%U7_7:s1,
                              U7_7:f%U4_4:s1%U9_9:s1%U8_8:s1,
                              U4_4:b%U7_7:r1.1%U8_8:r1%U6_6:r1.1%U9_9:r1,
                              U9_9:b%U7_7:r1.2,
                              U8_8:b%U7_7:r2.1 );
U8_8:      DCsetresetAND2_0 ( U8_8:b%U7_7:r2.1,
                              U7_7:f%U4_4:s1%U9_9:s1%U8_8:s1,
                              U8_8:f%U4_4:s2%U5_5:s1,
                              U4_4:b%U7_7:r1.1%U8_8:r1%U6_6:r1.1%U9_9:r1,
                              U5_5:b%U8_8:r2%U6_6:r1.2%U9_9:r2 );
U9_9:      DCsetresetAND2_0 ( U9_9:b%U7_7:r1.2,
                              U7_7:f%U4_4:s1%U9_9:s1%U8_8:s1,
                              U9_9:f%U3_13:in2,
                              U4_4:b%U7_7:r1.1%U8_8:r1%U6_6:r1.1%U9_9:r1,
                              U5_5:b%U8_8:r2%U6_6:r1.2%U9_9:r2 );
Udummy1_10:  DCsetreset_0 ( Udummy1_10:b%U2_2:r2%U4_4:r2,
                            U1_11:out%Udummy1_10:s1%U3_3:s1,
                            Udummy1_10:f%U1_1:s1,
                            U1_1:b%Udummy1_10:r1 );
U1_11:      CeleIn2 ( U2_2:f%U1_11:in1,
                      U4_4:f%U1_11:in2,
                      U1_11:out%Udummy1_10:s1%U3_3:s1 );
U2_12:      CeleIn2 ( U5_5:f%U2_12:in1,
                      U6_6:f%U3_13:in1%U2_12:in2,
                      U2_12:out%U7_7:s1 );
U3_13:      CeleIn2 ( U6_6:f%U3_13:in1%U2_12:in2,
                      U9_9:f%U3_13:in2,
                      U3_13:out%U4_4:s3%U5_5:s2 );
```

Because the specification is at the abstract level, we obtain only a main control circuit. As for the operations, these need further consideration. From this output result, the circuit implementing the asynchronous processor is shown in Figure 6.3, in which all the events have been inserted. For example, the event labelled PC in Figure 6.3 is inserted between DC1 and DC2.

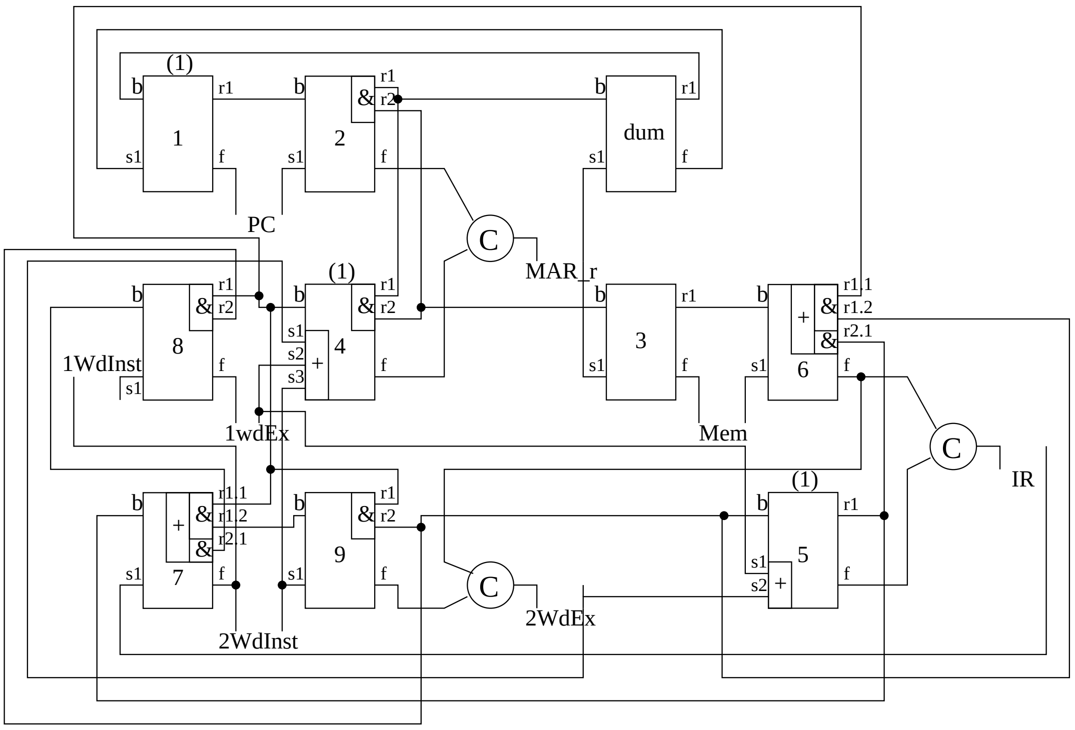


Figure 6.3 The DC implementation of the asynchronous processor.

From the implementation point of view, the next step for the designer is to implement all the events (i.e. operations). These operations are then inserted between the request and acknowledgement signals in the control circuit. If all inserted events are SI, the entire system will be SI.

This almost mechanical method yields a working result for any specification. Safety is guaranteed because of the SI of the result. However, the cost in silicon tends to be high and the performance tends not to be the best possible.

6.2 Fully-decoupled latch controller

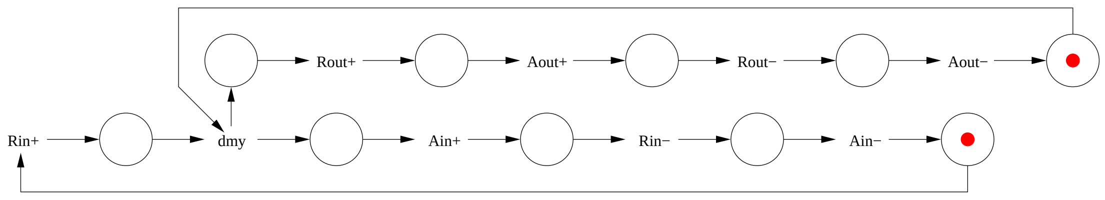


Figure 6.4 The STG specification of the fully-decoupled latch controller.

Figure 6.4 shows the original specification of a fully-decoupled latch controller in STG format, in which there are two parallel paths in order to fully decouple the input and output handshakes. This example is found in [Sotiriou 2001]. Here signals *Rin* and *Aout* are inputs, signals *Ain* and *Rout* are outputs, and signals *Rin* and *Ain*, *Rout* and *Aout* are two pairs of handshakes respectively.

The function of this controller is as follows: when *Rin* is asserted (Rin+), if the handshake pair *Rout*/*Aout* is idle (Aout-), the flow is parallel and both output signals *Ain* and *Rout* are active (Ain+, Rout+). Two handshakes are synchronized here. After the environment senses them, Rin- and Aout+ are passed to the controller to respond to them. Furthermore, the controller will withdraw *Ain* and *Rout* (Ain-, Rout-), and then the environment withdraws *Aout* (Aout-). After that, the controller is idle and waiting for the new request (Rin+).

As introduced in Chapter 5, the current PN2DCs tool is not suitable for dealing with low level (signal level) specifications directly, because it may introduce a large number of memory elements which affects the performance of the resulting circuits.

In order to synthesize this controller, firstly we convert the original specification shown in Figure 6.4 into an intermediate format (we called it RS (refined specification) format) based on the methods introduced in Chapter 5. The RS specification is shown in Figure 6.5,

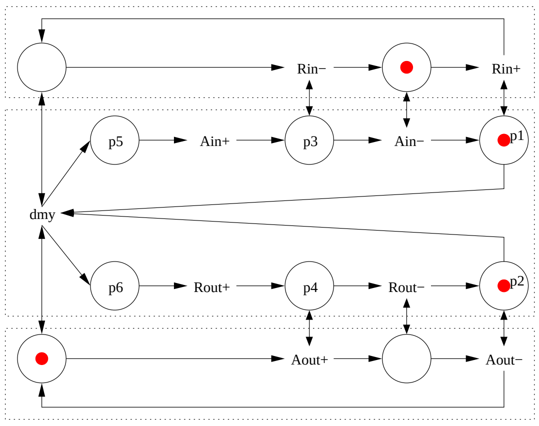


Figure 6.5 The RS format specification of the controller.

in which we separate the input and output signals based on the ideas of the refinement methods introduced in Chapter 5. Here the middle parts consist of output signals and the top and bottom parts consists of input signals. Based on this new specification we generate an input file as follows:

```
.model A_fully_decoupled_latch
# Declaration of signals
.inputs p1 p2 p3 p4 p5 p6
.outputs  Ain- Rout- Ain+ Rout+ dmy
.marking p1 p2
# Petri net
.graph
p1 dmy
p2 dmy
dmy p5 p6
p5 Ain+
Ain+ p3
p3 Ain-
Ain- p1
p6 Rout+
Rout+ p4
p4 Rout-
Rout- p2
# initial marking
.end
```

By employing the PN2DCs tool, we obtain an output result file as follows:

```
U1_1:    DCsetresetAND2_1 ( U1_1:b%U3_3:r1,
                     U3_3:f%U1_1:s1,
                     U1_1:f%U1_7:in1,
                     U5_5:b%U1_1:r1%U2_2:r1,
                     U6_6:b%U1_1:r2%U2_2:r2 );
U2_2:    DCsetresetAND2_1 ( U2_2:b%U4_4:r1,
                     U4_4:f%U2_2:s1,
                     U2_2:f%U1_7:in2,
                     U5_5:b%U1_1:r1%U2_2:r1,
                     U6_6:b%U1_1:r2%U2_2:r2 );
U3_3:    DCsetreset_0 ( U3_3:b%U5_5:r1,
                 U5_5:f%U3_3:s1,
                 U3_3:f%U1_1:s1,
                 U1_1:b%U3_3:r1 );
U4_4:    DCsetreset_0 ( U4_4:b%U6_6:r1,
                 U6_6:f%U4_4:s1,
                 U4_4:f%U2_2:s1,
                 U2_2:b%U4_4:r1 );
U5_5:    DCsetreset_0 ( U5_5:b%U1_1:r1%U2_2:r1,
                 U1_7:out%U5_5:s1%U6_6:s1,
                 U5_5:f%U3_3:s1,
                 U3_3:b%U5_5:r1 );
U6_6:    DCsetreset_0 ( U6_6:b%U1_1:r2%U2_2:r2,
                 U1_7:out%U5_5:s1%U6_6:s1,
                 U6_6:f%U4_4:s1,
                 U4_4:b%U6_6:r1 );
U1_7:    CeleIn2 ( U1_1:f%U1_7:in1,
            U2_2:f%U1_7:in2,
            U1_7:out%U5_5:s1%U6_6:s1 );
```

This output file introduces six DCs and one C-element. According to the input specification, the events Ain-, Rout-, Ain+, and Rout+ are inserted in DCs based on the specification. Here we define the event Ain- as resetting the signal *Ain*, Ain+ as setting the signal *Ain*. Similar for Rout- and Rout+. The resulting DC control circuit is based on the DC model introduced in section 6.1 and shown in Figure 6.6.

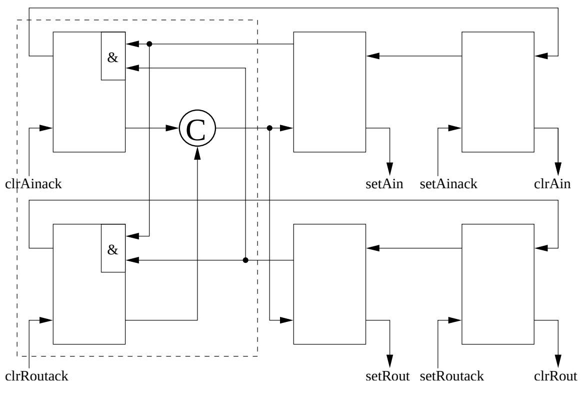


Figure 6.6 The DC control circuit of the controller.

However, this is not a complete circuit. The signals *Ain* and *Rout* are outputs, and they should be included inside the controller. The circuit shown in Figure 6.6 does not have this feature; only setting and resetting signals are presented. In order to implement the *Ain* and *Rout* signals, we employ the SR latch with completion-detection introduced in Chapter 5. Because *Rin*/*Ain* and *Rout*/*Aout* are two handshakes, the following SR latch (Figure 6.7) is used.

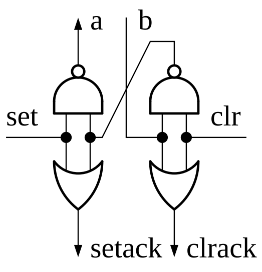


Figure 6.7 The SR latch with completion-detection logic for handshake signals.

In Figure 6.7, *a* is an output signal and *b* is an input signal, and form a pair of handshake signals. So *Ain* and *Rout* can be implemented as shown in Figure 6.8, and the whole circuit is shown Figure 6.9.

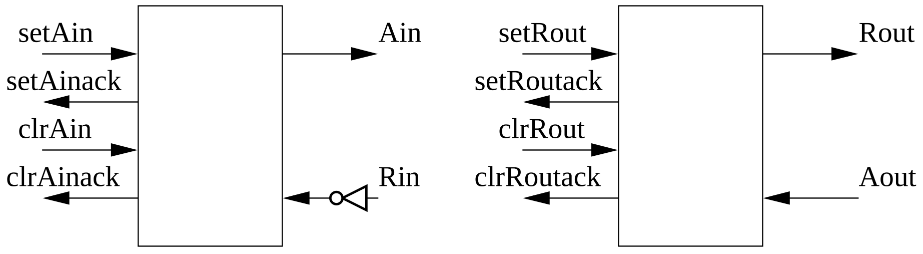


Figure 6.8 The output signal implementation in the controller.

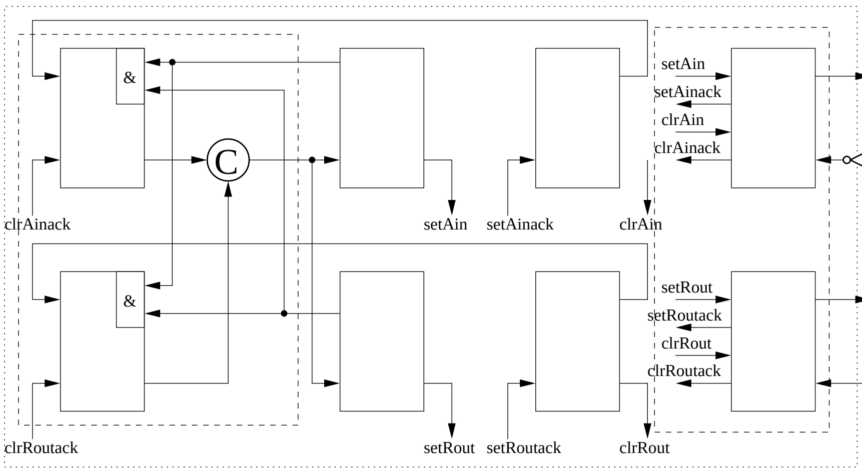


Figure 6.9 The whole circuit.

However, the circuit of the controller is not optimal. For instance, by combining two of the DCs dotted in Figure 6.9, a simple circuit with the same functionality can be obtained. This is shown in Figure 6.10.

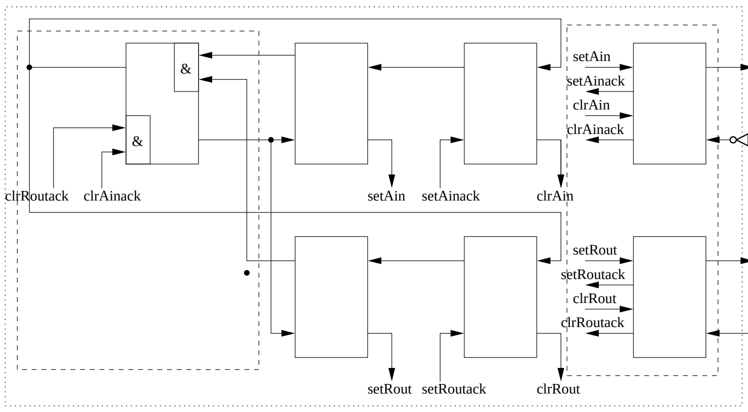


Figure 6.10 The optimized implementation.

Although this solution is better than the previous one, we can still optimize it. In this solution, there are total 7 memory elements, five DCs and two SR latches. As discussed in Chapter 5, more memory elements should affect the performance. In Chapter 5, we have proposed an optimization method which is based on the VDCs. Using this method, a simpler solution can be obtained. This circuit is shown in Figure 6.11.

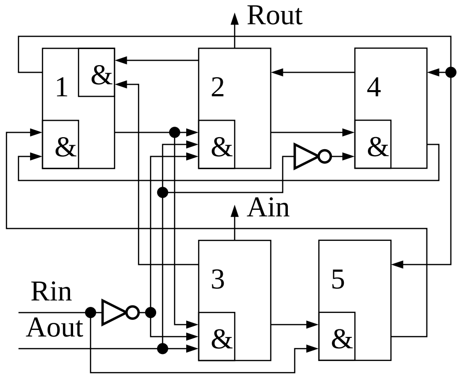


Figure 6.11 The optimized implementation.

In this solution, only five memory elements are used, so it is better than the two previous ones. Apart from this, there is another advantage in this solution: we can optimize it to RT solution as shown in Chapter 5, which improves the performance.

From the solution shown in Figure 6.11, DCs 1, 4 and 5 are used only to maintain the integrity of the token flow in this small circuit. This provides an opportunity for post-translation optimization if timing assumptions can be introduced (i.e., allowing the circuit to become not completely SI). The minimal solution should probably contain no more memory elements than are needed to maintain the output variables (the VDCs). If this is true, DCs 1, 4 and 5 may be removed and only VDCs 2 and 3 need to remain. The main memory elements within a (V)DC is a normal SR latch. It may be possible to use two SR latches with appropriate set and reset functions to replace the VDCs in the circuit. So the "length three" rule no longer applies and the other DCs can be removed.

Based on the implementation in Figure 6.11, the inputs of the set function for both SR latches which replace the VDCs should consist of signals *Rin* and *Aout*. The inputs of the reset function should be from DC4 and DC5 respectively. The operations of DC4 and DC5 are that DC5 is set after Aout+ happens, and DC5 is set after Rin- happens. Then VDC2 (Rout-) and VDC3 (Ain-) are reset respectively. This makes it possible to use signals *Aout* and *Rin* to replace the reset signals from DC4 and DC5 respectively in the reset functions, which in addition should also contain token propagation signals. Thus, we obtain a block diagram of an optimized circuit (RT solution) shown as in Figure 6.12.

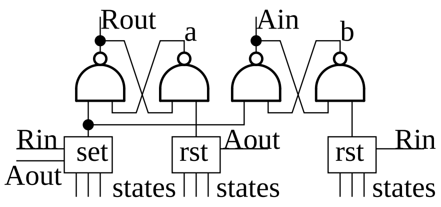


Figure 6.12 The block diagram of RT solution.

In Figure 6.11, DC1 is used only to maintain correct token propagation by setting VDCs 2 and 3 under appropriate conditions. The initial values of signals *Rout*, *Ain* can be used for this purpose instead. Following this path, and taking care of the polarity of the signals involved, we build the simplified circuit shown in Figure 6.13.

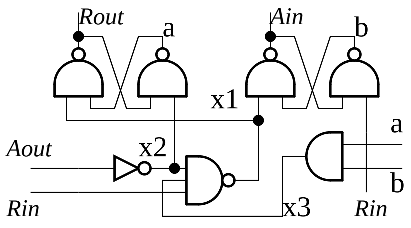


Figure 6.13 The RT solution of the controller.

This circuit is not SI but rather works only under certain RT assumptions, namely that Rout+ should be earlier than x1+, b- and Rin-, and Ain+ should be earlier than x1+, a- and Aout-. There assumptions are reasonable for this circuit, if both SR latches have the same speed and the environment has finite speed. This is true if the entire circuit is implemented with the same technology and the environment consists of real circuits.

Using the PN2DCs tool, we generate a basic asynchronous circuit (SI) for this example. Based on this basic solution, using the optimization methods introduced in Chapter 5, we easily generate a VDC solution and an RT solution for this example.

We would like to compare the solution obtained based on this PN2DCs tool with other ones. As a result, simulations have been used to compare the circuit from the original article [Sotiriou 2001], a circuit designed using Hollaar's method [Hollaar 1982], a circuit constructed from simple gates found in standard libraries based on the Petrify solution [Petrify], the circuit shown in Figure 6.10 and the circuit shown in Figure 6.13.

Table 6.1 Comparison results

	Sotiriou's Method (Non SI)	Hollaar's (Non SI)	Petrify (Non SI)	VDC (SI)	RT (Non SI)
Rin+ −> Rin+	1.87ns	2.10ns	2.94ns	5.33ns	1.45ns
Rin+ −> Ain+	0.52ns	0.45ns	2.03ns	2.57ns	0.32ns
Rin+ −> Rout+	0.57ns	0.49ns	2.38ns	2.59ns	0.35ns
Rin+ −> Ain−	1.55ns	1.79ns	2.67ns	4.94ns	1.13ns
Rin+ −> Aout−	1.66ns	1.82ns	3.67ns	5.4ns	1.28ns
Num. of Gates	Trans. Level	20	11	17	7

Table 6.1 lists the result of this comparison. The simulation was done under the Spice3 tool [Spice3] on the Unix system, which was run on the SUN ULTRA 30.

From this Table, the VDC solution is not fast compared with the others. However it is SI. So it is safer than the others. Especially, we can easily obtain a RT solution from VDC one. The RT solution is the best one from the performance point of view.

6.3 Two-slot *Signal* ACM controller

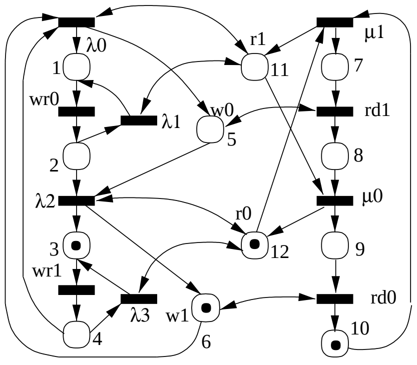


Figure 6.14 The PN specification of the two-sot *Signal*.

The control circuit of the two-slot *Signal*, which we now introduce, is designed by using our direct translation method. The PN specification of the two-slot *Signal* is shown in Figure 6.14, which is obtained by using PN specification synthesis [Yakovlev 2001]

In Chapter 3, we introduced hardware implementation from the optimization solution shown in Figure 3.58. Here we explain the process from PNs to hardware circuits. In Figure 6.14, the left side is the specification for the writer process and the right side is the one for the reader process. Because of the three important properties of the ACM, asynchrony, data coherence and data freshness, the writer and reader processes should be running independently. The communication is controlled through the control variables *w* and *r*. In other words, the control variables are sensed by the writer and reader respectively to guarantee that the communication keeps data coherence and data freshness properties. For example, the writer senses the control variable *r* to decide whether to set/reset the control variable *w*.

From the hardware implementation point of view, we should design the writer and reader separately because they are two independent processes. On the writer side, there are four events, wr0, wr1, clrw and serw. The control variable *r* is an input signal. It is sensed by the writer and then the corresponding event is started. For example, if *r*=1 and place 4 has a token, clrw fires and then the token moves to place 1. Otherwise, nothing happens and the token moves to place 3. The control variable *w* is an output signal which is set and reset by the events setw and clrw respectively. We can initially ignore this output signal when we construct the control circuit for the writer.

The specification for the writer is modified as shown in Figure 6.15.

In this specification, places 1 and 3 have the same functional proprieties. Although there is no token in place 1, we should assume a token there. Based on this assumption, there are four streams in this specification. They are:

1. place1, place2, and place1;
2. place1, place2, and place3 (assuming place1);

3. place3, place4, and place3;
4. place3, place4, and place1 (assuming place3)

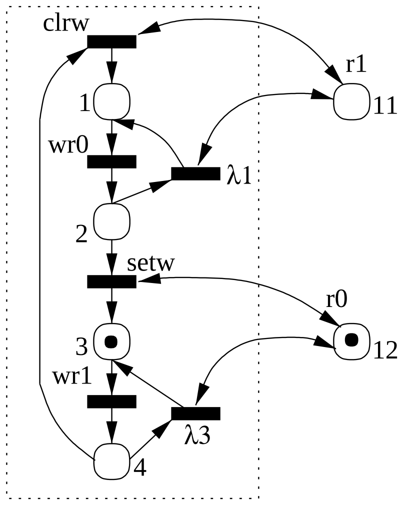


Figure 6.15 The specification for the writer.

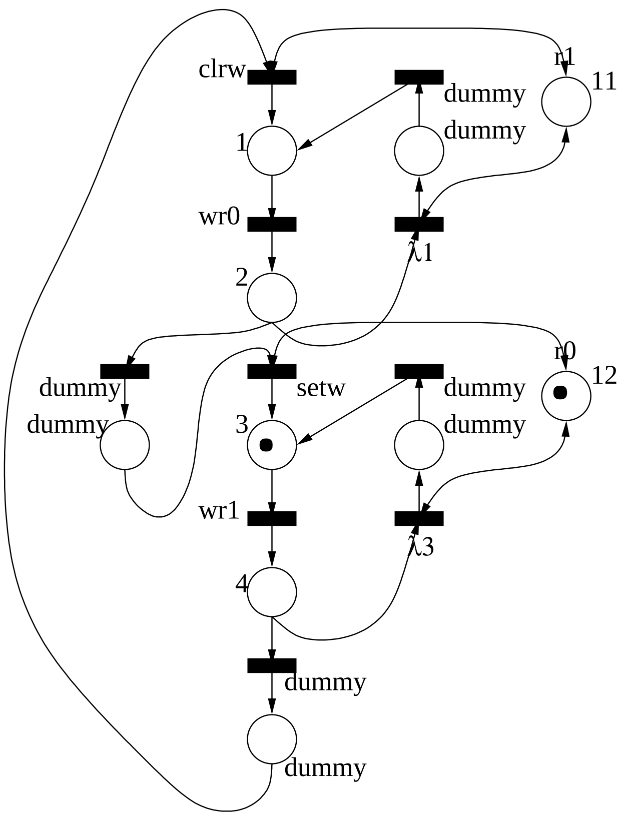


Figure 6.16 A refined specification.

Because at least three DCs are needed in each stream, PN2DCs can generate the specification to an implementable one automatically, which is shown Figure 6.16.

Finally PN2DCs produces a VHDL description of a circuit. We construct this control circuit schematically. It is shown in Figure 6.17.

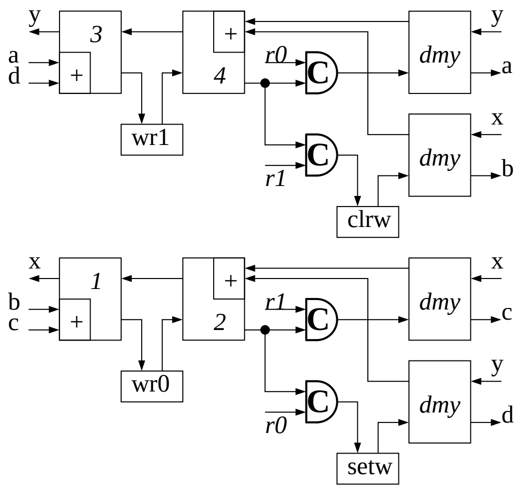


Figure 6.17 The circuit obtained based on the direct-translation method.

The obtained circuit does not yet fully correspond to the specification. The problem is caused by the input signal *r*, because *r* is not managed directly by the writer. The circuit samples only the value of *r* (*r0* standing for *r*=0 and *r1* *r*=1) and then decides which stream to start. In fact, the actual value is irrelevant. So we can use sync (see Figure 3.57) to replace the C-elements. However, if we simply replace them, the circuit will not be safe. This is because *r* may be changed immediately after *w* changes. To solve this problem, the solution is to separate the sampling of *r* and the changing of *w*.

On the other hand, the obtained circuit from the direct translation method is not an optimal circuit. In this example, DC1 and DC3 can be combined. After optimizing the circuit by hand, the final version of the circuit is shown in Figure 6.18. This was discussed in more detail in Chapter 3.

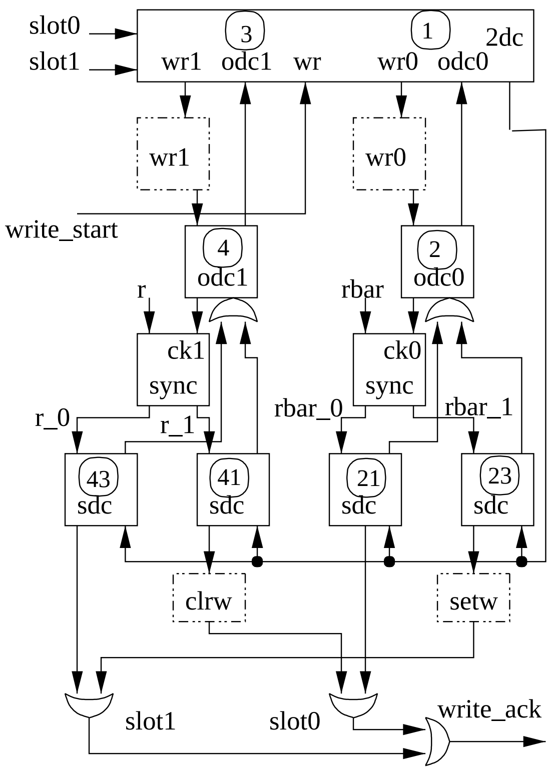


Figure 6.18 The implementation of the writer.

Using the same method, we can easily generate the circuit for the reader. The intermediate solution is shown in Figure 6.19.

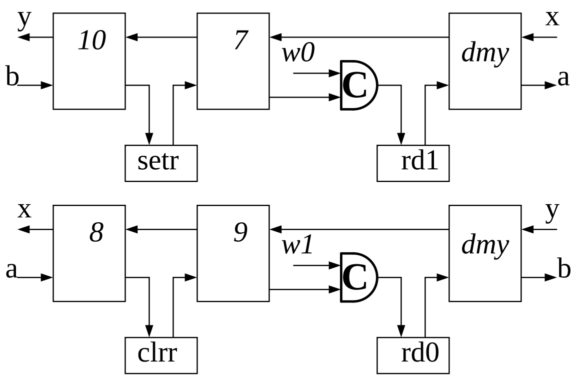


Figure 6.19 The circuit obtained from the direct-translation method.

The same method is subsequently used in the reader part. Finally we generate the circuit for the reader.

6.4 Master-read benchmark

This example is from a standard set of STG benchmarks. The original STG specification is shown in Figure 6.20, which is a low level one.

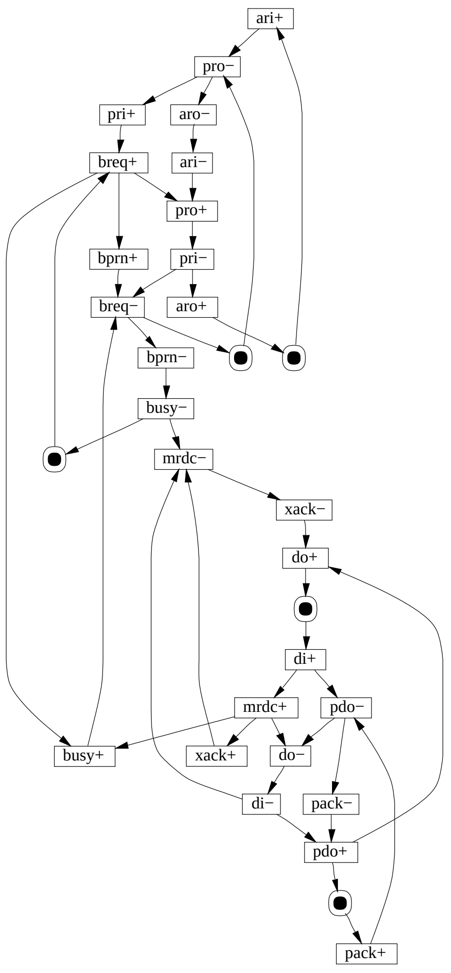


Figure 6.20 The STG specification.

In this specification, signals *ari*, *bprn*, *xack*, *di*, and *pack* are inputs and signals *aro*, *pro*, *breq*, *mrdc*, *do*, *pdo*, and *busy* are outputs. Because it is a low level specification, the operations are defined as the up-going and/or down-going of these signals.

We first convert the original specification to an intermediate format (RS) specification in order to remove redundant memory elements. We then obtain a pure control part and an input part, which are connected by read-arcs. The RS specification is shown in Figure 6.21.

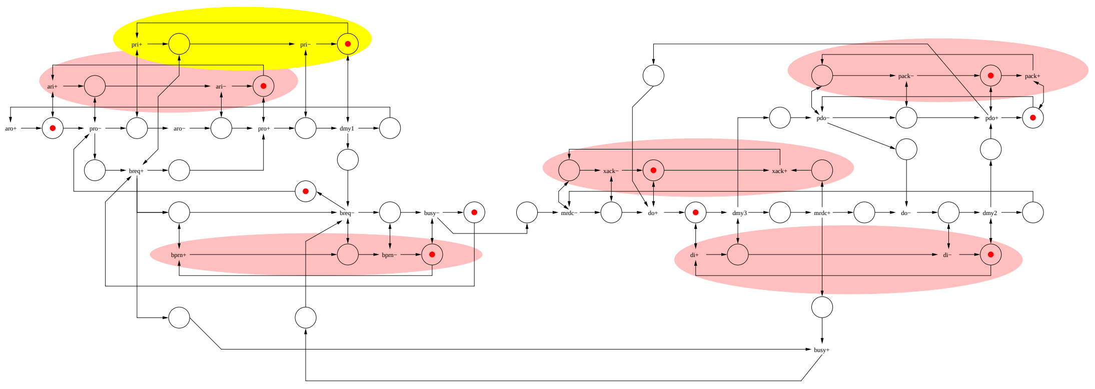


Figure 6.21 The RS specification of the benchmark.

The PN2DCs tool can deal with this RS specification and generate a DC control circuit. However, because there are a large number of places in the control part, many memory elements will be generated, as well as many extra SR latches for the output signals.

Because the resulting circuit contains so many memory elements, it is too large to reproduce here. Its performance will be very poor.

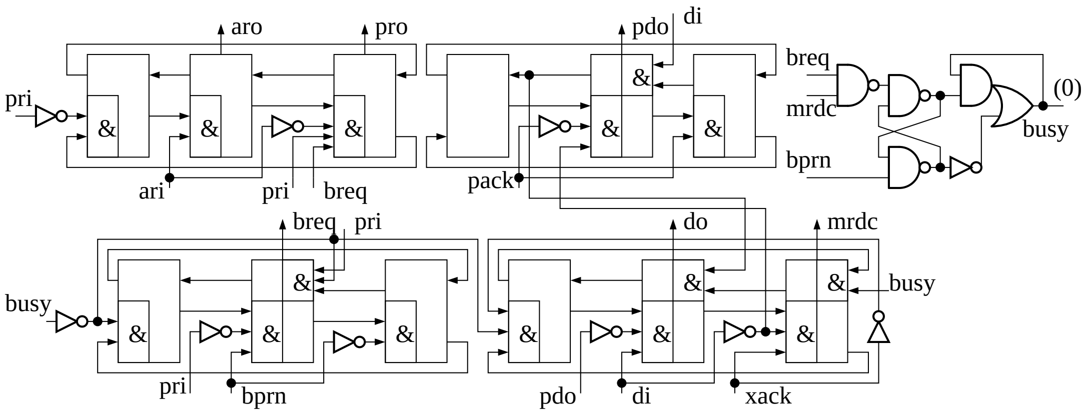


Figure 6.22 The refined implementation.

In order to generate a circuit of reasonable size, we can produce a new solution using VDCs which are generated using the methods introduced in Chapter 5. In this way, a better solution has been obtained, which is shown in Figure 6.22.

We also would like to compare our solutions with the other solutions, such as Hollaar’s and the solution obtained by using the Petrify tool.

Table 6.2 Comparison results

	Hollaar’s Method NOT SI	NDC SI	Petrify NOT SI	VDC SI
aro+ −> aro+	5.06ns	21.67ns	5.95ns	6.93ns
ari− −> aro+	1.96ns	7.96ns	2.99ns	4.17ns
ari+ −> aro−	2.53ns	13.15ns	2.60ns	2.22ns
mrdc+ −> mrdc+	5.06ns	21.67ns	5.95ns	6.93ns
xack− −> mrdc+	1.65ns	4.91ns	1.77ns	2.82ns
xack+ −> mrdc−	2.80ns	16.11ns	3.79ns	3.70ns

Table 6.2 gives a comparison result which is obtained by comparing the circuit obtained from the tool directly and the circuit after optimizing using VDCs, and circuits generated by using Hollaar’s method and using the Petrify tool.

This example is quite big. From the above Table, we can find the speed of the VDC solution is similar to Hollaar’s and Petrify’s. In addition, the circuit is still SI. Here we did not show the RT solution. It should be the fastest one.

Chapter 7: Conclusions and Future Work Discussion

Summary

This chapter presents the conclusions for the contributions offered by this work. This work is done in the following areas: 1) self-timed ACM design; 2) implementation; 3) simulation and manufacture; 4) testing; 5) development of a synthesis tool based on self-timed ACM designs.

The first self-timed ACM design, four-slot *Pool*, was done by hand. Not using any synthesis tools caused us many problems. The second self-timed ACM design, three-slot *Pool*, was done using an existing synthesis tool, Petrify. However, the results obtained from Petrify are not guaranteed to be SI using a standard library. In order to generate SI circuits, some manual work was needed. The third ACM design, two-slot *Signal*, was done based on the following design flow:

1. conceptual definition (using PNs);
2. construction of the basic protocol specification (using a state graph);
3. refining the state graph with silent actions;
4. synthesis of a PN specification (using regions);
5. translation of the PN model into a circuits using DCs;
6. entering the design into the Cadence tool and simulation.

Steps 1 to 4 are outside the scope of this thesis. Only step 5 and step 6 are introduced here.

After simulation and fabrication, in order to test the ACM circuit, a new testing technique was proposed. Using this new method we tested the four-slot *Pool* successfully.

As a result of our work, all basic ACM type of circuit can be implemented in hardware, especially in the form of self-timed circuits.

While we investigated the *Pool*, *Signal*, and *Message* kind of ACM, we also studied asynchronous design methodology. Based on our experience of self-timed design, we developed a synthesis tool and defined a useful library for this tool. In addition, in order to obtain a better circuit, some optimization methods were also proposed.

7.1 Introduction

As claimed in Chapter 1 and Chapter 2, future chips will consist of several independent systems, and future digital circuits will be asynchronous. It is necessary to study some related topics now, such as self-timed implementation and automatic design (synthesis) methodologies, in order to adapt future requirements.

One interesting and useful communication mechanism between systems is ACM. The ACM is divided into four types. They are *Channel*, *Pool*, *Signal* and *Message*. Except for the *Channel* type of ACM, however, the mechanisms of the other kinds of ACMs have not been studied and implemented using self-timed circuits.

The ACM type of circuit has two independent processes which communicate by using the ACM. It has three important properties. Self-timed implementations and these properties introduce many difficulties with regard to testing.

7.2 Conclusions

7.2.1 Self-timed ACM implementations

A new system of classifying asynchronous data communication mechanism has been proposed, which is based on the blocking and waiting properties of the reader and writer actions in ACMs. This new classification is much more meaningful than the original one [Simpson 1994] from the viewpoint of the timing requirements to the system. For example, communications between two independent processes are allowed in the *Message* type of ACM unlike in the *Constant* type of ACM in the original classification.

In this thesis, the mechanisms of the four-slot *Pool*, three-slot *Pool*, two-slot *Signal*, and its dual type, the *Message* type of ACM, have been studied and implemented using self-timed circuits. The simulation results show that they work as expected.

These ACM designs, within their local boundary, are not fully asynchronous (in the sense of non-blocking) by virtue of the unpredictable waiting introduced by the arbiters. However, unlike the original fundamental mode slot systems proposed in [Simpson 1990b], which may forcibly ensure full asynchronism by relaxing the requirement on data coherence when metastability occurs, the four-slot *Pool* implementation can give the client (the designer of the writer and reader processes) the choice of either sacrificing timing independence or data coherence by defining the overall protocol between the reader and the read statement buffer and the writer and the write statement buffer.

If a choice is made to give temporal independence priority over data coherence, then the new ACMs would perform similarly to the original algorithms in terms of data coherence violation rates. This is because the statistical profile of metastability is unchanged, and the arguments of settling metastability in repeated copying inside processors do not apply when both ACM and then client processors are not on the same chip.

In addition, the client designer may choose not to lose data coherence when the *done* signal is not forthcoming, but still be able to preserve timing integrity for the access process. Since at this point the access process has the information that metastability has occurred within the ACM, it may be specified not to carry out the current item of data (for the writer) or use the item of data acquired during the last cycle (for the reader). In this case data freshness is sacrificed. This is not a real sacrifice because when data coherence is not maintained, data freshness becomes automatically meaningless.

The three-slot *Pool* and two-slot *Signal* do not have this possibility. They only work when sacrificing timing independence. This means that the ACM can run as fast as possible.

The new option of letting the ACM run as fast as it can should produce significant speed gains simply because metastability is such a rare event.

In essence, these designs eliminate critical sections on the data slots by using the slot ACMs as basic components. They shift critical sections to small control variables implemented by means of arbiters and SI statement circuits, and give clients the choice of whether to make full use of these minimised critical sections.

In order that ACMs can run as fast as possible, as introduced in Chapter 3, handshake interfaces are introduced for each statement in our self-timed implementations. This means that metastability has been moved from the control variables in the original mechanisms to the handshake interfaces in our self-timed mechanisms. Simulation results under the Cadence tool show that no errors happened on the control variables. From this point of view, these self-timed ACMs are safe.

Although small timing interference exists in self-timed ACM implementations, simulation results illustrate that metastability does not propagate through the Mutexes. It was settled down inside the Mutexes. So metastability does not affect the functionality of the mechanisms.

After the four-slot *Pool* was built, the three-slot *Pool* was implemented too. This illustrates that self-timed implementation can reduce slot memory in implementations. This is one aim of studying self-timed ACM implementations.

In addition, in order to get fast ACM circuits, RT technology is introduced into the implementations. Except for SI implementations, RT implementations were also implemented. However, they should work under some reasonable timing assumptions. The simulation results show RT implementations are as expected.

We want to mention that the two-slot *Signal* type of ACM was designed using a proposed synthesis method, rather than manually and/or using existing CAD tools.

Techniques for the synthesis and implementation of ACM mechanisms, only partially formalised and to a large extent unautomated, have been presented. The overall design proceeded along the steps introduced above.

The model transformation steps (2 – 4), which are shown in the summary section in this chapter, are currently only supported to a very limited extent by the Petrify synthesis software. The refinement with silent events, to make the state graph synthesizable into a PN of a given class, is a very challenging theoretical problem and more research is needed here. The direct translation of PNs into DCs and subsequent optimisation with relative timing is another problem to be tackled in the future.

Our synthesis tool, PN2DCs, is based on the above self-timed practices and is employed in step 6.

7.2.2 Testing ACM

After implementation, we investigate another hot topic, testing asynchronous circuits. Although a lot of research has been done recently, testing techniques are still not adequate. Especially, no testing methods are available for testing ACM circuits.

ACM is a fully asynchronous kind of system. It has three important properties, asynchrony, data coherence and data freshness. It is much more difficult to test this kind of circuit.

In this thesis, we proposed a method which is based on checking the sequence of data input items, rather than individual items, whilst varying the rates of data communication.

The testing results show that the slot ACM implemented by using self-timed circuits performs as expected. The mini-interlock introduced by using the Mutex does not affect the operation of the ACM appreciably. No obvious violations of the three important properties have been observed. In addition, the testing results correspond with the simulation results obtained by using the Cadence tool, which indicates that the implementation did not introduce errors.

Though the testing was successful, and the on-chip testing circuits performed as expected, there is still potential for improvement. The testing circuits should be made nimbler and provided with more observation points. We plan to continue research on developing "more asynchronous" techniques for testing inherently asynchronous properties of circuits that are free from global clock.

However, owing to the limit on the area and the pin number of the chip, the scope of the testing was necessarily limited.

7.2.3 Synthesis

An extension to the direct translation method based on Varshavsky's direct translation method [Varshavsky 1996] has been presented in this thesis.

This method can be used to synthesize PN specifications which consist of input and output events at both the high level (behaviour level) and the low level (signal level). An important property is that the solution obtained using this method is guaranteed SI.

As mentioned in Chapter 5, compared with logic synthesis methods which depend on the encoding of the state space using abstract variables, direct translation method based on DCs has many advantages. It can deal with big specifications and provides SI without resorting to assumptions about the speed of devices such as inverters, or about the existence of large complex gate libraries. The results are easier to analyse or debug because of the topographical similarity between a DC-based solution and its PN specification. Because of the use of one-hot encoding, DC-based solutions are also potentially faster, especially for large systems.

As Varshavsky's direct translation methods claimed, however, the cost and performance of its resulting circuits are not entirely satisfactory and tend to compare poorly with those obtained with tools such as Petrify.

A number of techniques with which Varshavsky's direct translation method can be improved has been developed and shown to be effective with a number of demonstrative and real-life examples and case studies. These techniques, when employed sensibly, bring direct translation to within the same quantitative level in cost and performance to Petrify on examples of small size.

A prototype automatic tool employing this direct translation technique has been developed and tested. It incorporates additional refinement features compared with Varshavsky's method, but does not yet contain all the improvement techniques we have developed for direct translation.

It is not possible to compare the direct translation method with such methods as Petrify for systems of large size because Petrify has difficulty handling specifications of large size because of the state explosion problem. In comparison, direct translation in general and PN2DCs in particular demand a computational complexity which is linearly related to the specification.

7.3 Areas of future research

7.3.1 Summary

The potential for future research in the areas related to the work presented above is enormous. In the future, we would like to concentrate on developing an asynchronous CAD tool and to look for fast and safe asynchronous circuit (SI and/or RT) implementations for this tool. We expect this asynchronous CAD tool to include the direct-translation synthesis tool based on DCs and an asynchronous testing tool.

7.3.2 Direct-translation method based on DCs

Although the direct-translation method presented in Chapter 5 can deal with input and/or output events in PN specifications, it is not a mature synthesis tool. Mostly it is good at coping with control circuits. From the hardware design point of view, a system should consist of both datapath and control circuits.

We would like to develop a synthesis methodology which can synthesise both control circuits and datapath. In this work we will introduce Colour Petri nets (CPNs) for datapath and still use LPNs for control circuits [Burns 2002].

Although LPNs and CPNs are very popular in the asynchronous community, we prefer VHDL as our input specification language. This is because most circuit designers are familiar with it.

In short, in our CAD tool, VHDL specifications can be translated into LPNs and CPNs for control circuits and datapath circuits respectively, after which the LPNs will be mapped to asynchronous DC circuits and CPNs will be mapped to asynchronous datapath circuits.

7.3.3 Asynchronous design for testing method

In order to meet time-to-market requirements, while designing asynchronous circuits, some testing circuits should be considered. We would like to investigate asynchronous design for testing methods and to develop a tool to support it.

Although the testing for ACM circuits is successful and on-chip testing circuits performed as expected, there are still potentials for improvement. In addition, the testing circuits should be made more reactive and provided with more observation points. We plan to continue research on developing “more asynchronous” techniques for testing inherently asynchronous properties of circuits that are free from global clock.

7.3.4 Fast and reliable asynchronous circuits

After using our direct translation synthesis method, a DC circuit can be obtained. However, as mentioned in Chapter 4, although it is safe, it is not fast enough. In order to generate a fast circuit, NDC is introduced in Chapter 5. But it is not good enough. We would like to look for new kinds of asynchronous circuits for our synthesis tool.

RT circuits can work safely under some reasonable assumptions. We also would like to introduce RT circuits into our synthesis tool.

7.3.5 Theory work

After studying the above synthesis method and testing method, we would like to develop a theoretical basis for them.

Much work is needed to further formalize and codify the refinement techniques so that they can all be incorporated into PN2DCs. Other refinement and optimization methods will also be investigated in the future.

References

Alur 1994	Alur, R., and Dill, D. L., A Theory of Timed Automata, The Oretical Computer Science, 126(2), pp.18-235, 1994.
Anderson 1996	Anserson, J., and Gouda, M., A Criterion for Atomicity, Formal Aspects of Computing Vol. 4, pp. 273-298, 1996.
Amulet	http://www.cs.man.ac.uk/amulet.
Armstrong 1993	Armstrong, James R., and Gray, F. Gail, Structured Logic Design with VHDL, ISBN 0-13-855206-1, Prentice-Hall, Inc., 1993.
Ashenden 1996	Ashenden, Peter J., The Designer's Guide to VHDL, ISBN 1-55860-270-4, Morgan Kaufmann Publishers, Inc., 1996.
Badouel 1998	Badouel, E, and Darondeau, Ph., Theory of Regions, Lecture Notes in Computer Science, Vol. 1491, pp. 529-586, Springer-Verlag, 1998.
Bainbridge 2000	Bainbridge, W. J., Asynchronous System-on-Chip Interconnect, Ph.D thesis, Department of Computer Science, University of Manchester, 2000.
Beerel 1991	Beerel, Peter A., and Meng Teresa H.-Y., Semi-Modularity and Self-Diagnostic Asynchronous Control Circuits. In Carlo H. Séquin, Editor, Advanced Research in VLSI, pp. 103-117. MIT Press, March 1991.
Beerel 1992a	Beerel, Peter A., and Meng, Teresa H.-Y., Semi-Modularity and Testability of Speed-Independent Circuits, Integration, the VLSI Journal, 13(3), pp. 301-322, September 1992.
Beerel 1992b	Beerel, P., and Meng, T. H.-Y., Automatic Gate-Level Synthesis of Speed-Independent Circuits, In Proceedings of International Conference on Computer Aided Design (ICCAD), pp. 581-586, Santa Clara, California, USA, Novermber 1992.

Beerel 1994	Beerel, P. A., Myers, C. J., and Meng, T. H.-Y., Automatic Synthesis of Gate-Level Speed-Independent Circuits, Technical Report CSL-TR-94-648, Stanford University, Novermber 1994.
Berkel 1991	Berkel, K. van, Kessels, J., Roncken, M., Sawijs, R., and Schalij, F., The VLSI-Programming Language Tangram and its Translation into Handshake Circuits, In Proceedings European Conference on Design Automation (EDAC), pp. 384-389, 1991.
Berkel 1992	Berkel, K. van, Handshake Circuits: An Intermediary between Communicating Processes and VLSI, Ph.D thesis, Eindhoven University of Technology, 1992.
Berkel 1994a	Berkel, K. van, Burgess, R., Kessels, J., Peeters, Ad, Roncken, M., and Schalij, F., A Fully Asynchronous Low Power Error Corrector for the DCC Player, In International Solid State Circuits Conference, pp 88-89, February 1994.
Berkel 1994b	Berkel, K. van, Burgess, R., Kessels, J., Peeters, Ad, Roncken, M., and Schalij, F., Asynchronous Circuits for Low Power: A DCC Error Corrector, IEEE Design and Test of Computer, 11(2) pp. 88-89, February 1994.
Blunno 2000a	Blunno, I., Bystrov, A., Carmona, J., Cortadella, J., Lavagno, L., and Yakovlev, A., Direct Synthesis of Large-Scale Asynchronous Controllers Using a Petri net Based Approach, Handouts of Fourth ACiD WG Workshop, Grenoble, January 2000. http://time-cmp.imag.fr/tima/cis/cis.html
Blunno 2000b	Blunno, I., and Lavagno, L., Automated Synthesis of Micro-Pipelines From Behavioural Verilog HDL, Proceedings of IEEE Symposium on Advanced Research In Asynchronous Circuits and Systems (ASYNC'2000), pp. 84-92, Eilat, Israel, April 2000.
Borriello 1987	Borriiello, G., and Katz, R. H., Synthesis and Optimization of Interface Transducer Logic, In Proceedings IEEE 1987 ICCAD Digest of Papers, pp. 274-277, 1987.

Bredeson 1972	Bredeson, J. G., and Hulina, P. T., Elimination of Static and Dynamic Hazards for Multiple Input Changes in Combinational Switching Circuits, Information and Control, Vol. 20, pp. 114-224, 1972.
Brunvand 1989	Brunvand, E., and Sproull, R. F., Translating Concurrent Programs into Delay-Insensitive Circuits, In Proceeding of ICCAD, pp. 262-265, IEEE computer society press, November 1989.
Brunvand 1991	Brunvand, E., Translating Concurrent Communicating Programs into Asynchronous Circuits, Ph.D thesis, Carnegie Mellon University, 1991.
Bryant 1992	Bryant, R., Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Computing Surveys, 24(3), pp. 292-318, September 1992.
Brzozowski 1989	Brzozowski, J. A., and Ebergen, J. C., Recent Developments in The Design of Asynchronous Circuits, Technical Report CS-89-18, University of Waterloo, Computer science department, 1989.
Burns 2002	Burns, F., Shang, D., Koelmans, A., and Yakovlev, A., Synthesis of Asynchronous Data Path and Controllers using PNs, in Proceeding of 12th UK Asynchronous Forum, South Bank University, London, June 17-18, 2002.
Bystrov 1999	Bystrov, A., Shang, D., Xia, F., Yakovlev, A., Self-Timed and Speed Independent Latch Circuits, 6th UK Asynchronous Forum, University of Manchester, 12-13th July 1999.
Bystrov 2001	Bystrov, A. and Yakovlev, A., Asynchronous Circuit Synthesis by Direct Mapping: Interface to Environment, Tech. Report, Dept. of CS, University of Newcastle, CT-TR-743, Oct. 2001.
Carmona 2001	Carmona, J., Cortadella, J., and Pastor, E., A Structural Encoding Technique for The Synthesis of Asynchronous Circuits, Proceedings of ICACSD'01, pp. 157-166, IEEE

	computer society press (ISBN 0-7695-1071-X), June 2001, Newcastle upon Tyne, U.K.
Chaney 1973	Chaney, T. J., and Molnar, C. E., Anomalous Behaviour of Synchronizer and Arbiter Circuits, IEEE Transactions on Computers, C-22(4), pp.42-425, April 1973.
Chen 1998a	Chen, J. and Burns, A., Asynchronous Data Sharing in Multiprocessor Real-Time Systems Using Process Consensus, Proceedings of 10th Euromicro Workshop on Real-Time Systems, pp. 2-9, Berlin, Germany, IEEE Comp. Soc., June 17-19, 1998.
Chen 1998b	Chen, J., and Burns, A., Asynchronous Data Sharing in Multiprocessor Real-Time Systems Using Process Consensus, Tech. Report YCS 295 (1998), Department of Computer Science, University of York.
Chu 1985	Chu, T.-A., Leung, C. K. C., and Wanuga, T. S., A Design Methodology for Concurrent VLSI Systems, In Proc. International Conference Computer Design (ICCD), pp. 407-410, IEEE Computer Society Press, Nov. 1985.
Chu 1987	Chu, Tam-Anh, Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications, Ph.D thesis, MIT Laboratory for Computer Science, June 1987.
Clark 1998	Clark, I., Xia, F., Yakovlev, A. and Davies, A. C., Petri net Models of Latch Metastability, Electronics Letters, Vol. 34, No. 7, pp. 635-636, April, 1998.
Coelho 1989	Coelho, David R., The VHDL Handbook, ISBN 0-7923-9031-8, Kluwer Academic Publishers, 1989.
Cortadella 1997	Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L. and Yakovlev, A., Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers, IEICE Trans. Information and Systems, Vol. E80-D, No. 3, pp. 315-325, March 1997.
Cortadella 1998	Cortadella, J., Kishinevsky, M., Lavagno, L. and Yakovlev, A., Deriving Petri nets from Finite Transition Systems,

	IEEE Trans. on Computers, Vol. 47, No. 8, pp. 859-882, Aug. 1998.
Cortadella 2002	Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., and Yakovlev, A., Logic Synthesis for Asynchronous Controllers and Interfaces, ISBN 3-540-43152-7 Springer-Verlag Berlin Heidelberg New York, 2002.
Couvreur 1994	Ykman-Couvreur, C., Lin, B. and Man, H. D., Assassin: A Synthesis System for Asynchronous Control Circuits, Technical Report – User and Tutorial Manual, IMEC, September 1994.
David 1977	David, Rene, Modular Design of Asynchronous Circuits Defined by Graphs, IEEE Transactions on Computers, 26(8), pp. 727-737, August 1977.
Davis 1993	Davis, A., Coates, B., and Stevens, K., The Post-Office Experience: Designing a Large Asynchronous Chip, In Proceedings of the 26th Annual Hawaii International Conference on Systems Sciences, Vol. I, pp. 409-418, 1993.
Davis 1998	Davis, AI and Nowick, Steven M., An Introduction to Asynchronous Circuit Design, In A. Kent and J. G. Williams, Editors, The Encyclopaedia of Computer Science and Technology, Vol. 38, Marcel Dekker, New York, February 1998.
Dean 1992	Dean, M. E., STRiP: A Self-Timed RISC Processor Architecture, Ph.D thesis, Stanford University, 1992.
Dill 1988	Dill, D. L., Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits, The MIT press, Cambridge, Mass., An ACM Distinguished Dissertation, 1988.
Ebergen 1989	Ebergen, J. C., Translating Programs into Delay Insensitive Circuits, Vol. 56 of CWI tract. CWI, Amsterdam, 1989.
Eichelberger 1965	Eichelberger, E. B., Hazard Detection in Combinational and Sequential Switching Circuits, IBM Journal of Research and Development, Vol. 9(2), pp. 90-99, 1965.

Eles 1998	Eles, P., Kuchcinski, K., and Peng, Z., System Synthesis with VHDL, Kluwer Academic Publishers, P.O. Box 17,3300 AA Dordrecht, The Netherlands, 1998.
Ferranti 1952	Ferranti Sales Literature, Universal High-Speed Digital Computers: A Small Scale Experimental Machine, August 1952. http://www.computer50.org/kgill/mark1/sale/html.
Furber 1999	Furber, S. B., Garside, J. D., Riocreux, P. A., Temple, S., Day, P., Liu, J., and Paver, N. C., AMULET2e: An Asynchronous Embedded Controller, Proceedings of the IEEE 88(2), pp. 243-256, February 1999.
Gageldonk 1998	Gageldonk, H. van, Baumann, D., Berkel, K. van, Gloor, D., Peeters, Ad., and Stegmann, G., An Asynchronous Low-Power 80c51 Microcontroller. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 96-107, 1998.
Hauck 1995	Hauck, S., Asynchronous Design Methodologies: An Overview, Proceedings of the IEEE, Vol. 83, No. 1, pp. 69-93, January 1995.
Hayes 1993	Hayes, J. P., Introduction to Digital Logic Design, Addison-Wesley Publisher, 1993.
Hoare 1985	Hoare, C. A. R., Communicating Sequential Processes, Prentice-Hall, 1985.
Hoke 1999	Hoke, J. M., Bond, P. W., Lo, T., Pidala, F. S., and Steinbrueck, G., Self-Timed Interface for S/390 I/O Subsystem Interconnection. IBM Journal of Research and Development, 43(5/6), pp. 829-846, 1999.
Hollaar 1982	Hollaar, Lee A., Direct Implementation of Asynchronous Control Units, IEEE Transactions on Computers, C-31(12), pp. 1133-1141, December 1982.
Huffman 1954	Huffman, D. A., The Synthesis of Sequential Switching Circuits, In Moor, E. F. Editor, Sequential Machines: Selected Papers, pp. 3-62, Addison-Wesley, 1964. Reprinted from Franklin, F. Institute, Vol. 257, No. 3, pp. 161-190, Mar. 1954, and No. 4 pp. 275-303, Apr. 1954.

ITRS 2001	http://public.itrs.net/Files/2001ITRS.
Jackson 1977	Jackson, K., Language Design for Modular Software Construction, IFIP Congress Proceedings, pp. 577-581, 1977
Josephs 1990	Josephs, M. B., and Udding, J. T., An Algebra for Delay Insensitive Circuits, In R. P. Kurshan and E. M. Clarke, Editors, Proc. International Workshop on Computer Aided Verification, Vol. 531 of Lecture Notes in Computing Science, pp. 343-352, Springer-Verlag, 1990.
Jung 1994	Jung, S. T. and John, C. S., Direct Synthesis of Efficient Speed Independent Circuits from Deterministic Signal Transition Graphs, Proc. of International Symposium on Circuits and Systems, pp. 307-310, June 1994.
Jung 1999	Jung, Sung Tae and Myers, Chris J., Direct Synthesis of Timed Asynchronous Circuits. In Proc. International Conf. Computer-Aided Design (ICCAD), pp. 332-337, November 1999.
Keister 1951	Keister, W., Ritchie, A. E., and Washburn, S. H., The Design of Switching Circuits, Van Nostrand, Princeton, New Jersey, 1951.
Kessels 2001	Kessels, Joep, and Peeters, Ad, The Tangram Framework: Asynchronous Circuits for Low Power. In Proc. of Asia and South Pacific Design Automation Conference, pp. 255-260, February 2001.
Kinniment 1972	Kinniment, D. J., and Edwards, D. G. B., Circuit Technology in a Large Computer System, Based on a Paper Presented at the Joint 1ERE-IEE-BCS Conference on Computers Systems and Technology held in London, October 1972, Subsequently Published in The Radio and Electronic Engineer, Vol 43, No. 7, pp. 435-441, 1973.
Kinniment 1976	Kinniment, D. J., Woods, J. V., Synchronisation and Arbitration Circuits in Digital Systems, Proc. Of IEE, Vol. 123, No. 10, pp. 961-966, October 1976.

Kinniment 1998	Kinniment, D. J., Gao, B., Yakovlev, A. and Xia, F., Towards Asynchronous A-D Conversion, Proc. 4th International Symp. on Advanced Research in Asynchronous Circuits and Systems, San Diego, CA, pp. 206-215, IEEE computer society press, 1998.
Kinniment 1999	Kinniment, D. J., Measurements on a High Speed Arbiter, Technical Report Series, TR 677, Department of Computing Science, University of Newcastle, 1999.
Kirosis 1987	Kirosis, L. M., Atomic Multiread Register, Proc. 2nd Int. workshop on Distributed Computing, Amsterdam, LNCS-312, pp. 278-296, Springer Verlag, 1987.
Kishinevsky 1993	Kishinevsky, M., Kondratyev, A., Taubin, A., and Varshavsky, V., Concurrent Hardware: The Theory and Practice of Self-Times Design, John Wiley and Sons, London, 1993.
Kol 1997	Kol, R., Ginosar, R., Future Processors will be Asynchronous (sub-title: KIN*:* A High Performance Asynchronous Processor Architecture), Technical Report CC PUB#202 (EE PUB#1099), Department of Electrical Engineering, Technion - Israel Institute of Technology, Jul. 1997.
Kolks 1996	Kolks, Tilman, Vercauteren, Steven and Lin, Bill, Control Re-Synthesis for Control Dominated Asynchronous Design, Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems, March 1996.
Kondratyev 1998	Kondratyev, A., Cortadella, J., Lavagno, L., Taubin, A., and Yakovlev, A., Lazy Transition Systems: Application to Timing Optimization of Asynchronous Circuits, Proc. IEEE/ACM Int. conference on CAD (ICCAD'98), pp. 324-331, San Jose, IEEE Comp Soc. Press, Nov. 1998.
Kondratyev 1999	Kondratyev, A., Cortadella, J., Kishinevsky, M., Lavagno, L., and Yakovlev, A., Automatic Synthesis and Optimization of Partially Specified Asynchronous Systems.

	In Proc. ACM/IEEE Design Automation Conference, pp. 110-115, 1999.
Kondratyev 2002	Kondratyev, A., Sorensen, Lief, and Streich, Amy, Testing of Asynchronous Designs by "Inappropriate" Means. Synchronous Approach, In Proceedings of the 8th International Symposium on Asynchronous Circuits and Systems, IEEE Computer Society Press, April 2002.
Kopetz 1993	Kopetz, H., and Reisinger, J., The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem, In Proc. of the 14th IEEE Real-Time Systems Symposium, pp. 131-137, 1993.
Lamport 1977	Lamport, L., Concurrent Reading and Writing, Communication of the ACM, Vol. 20(11), pp. 806-811, 1977.
Lavagno 1991	Lavagno, L., Keutzer, K. and Sangiovanni-Vincentelli, A., Algorithms for Synthesis of Hazard Free Asynchronous Circuits, Proc. of the 28th Design Automation Conference, 1991.
Lavagno 1995	Lavagno, L., Keutzer, K., and Sangiovanni-Vincentelli, A., Synthesis of Hazard-Free Asynchronous Circuits With Bounded Wire Delays, IEEE Transaction on Computer-Aided Design, Vol. 14(1), pp. 61-86, January 1995.
Liljeberg 2001	Liljeberg, P., Plosila, J., and Isoaho, J., Asynchronous Interface for Locally Clocked Modules in ULSI Systems. In Proc. International Symposium on Circuits and Systems, volume 4, pp. 170-173, 2001.
Lin 1997	Lin, K. J., Kuo, C. W. and Lin, C. S., Synthesis of Hazard-Free Asynchronous Circuits Based on Characteristic Graph, IEEE Transactions on Computers, Vol. 46, No. 11, pp. 1246-1263, Nov. 1997.
Liu 1997	Liu, J., Arithmetic and Control Components for An Asynchronous System, Ph.D thesis, Department of Computer Science, University of Manchester, 1997.

Marino 1981	Marino, L. R., General Theory of Metastable Operation, IEEE Trans. Comput., C-30(2) pp. 107-115, February 1981.
Martin 1989a	Martin, A. J., Burns, S. M., Lee, T. K., Borkovic, D. and Hazewindus, P. J., The Design of An Asynchronous Microprocessor, In Decennial Caltech Conference on VLSI, pp. 226-234, 1989.
Martin 1989b	Martin, A. J., Programming in VLSI: From Communicating Processes to Delay Insensitive Circuits, In UT Year of Programming Institute on Concurrent Programming, Hoare, C. A. R., Ed. MA: Addison-Wesley, pp. 1-64, 1989.
Martin 1990a	Martin, Alain J., The Limitations to Delay-Insensitivity in Asynchronous Circuits, In Willian J. Dally Editor, Advanced Research in VLSI, pp. 263-278, MIT press, 1990.
Martin 1990b	Martin, Alain J., Programming in VLSI: From Communicating Processes to Delay-Insensitive VLSI Circuits, In C. A. R. Hoare, Editor, UT Year of Programming Institute on Concurrent Programming, Addison-Wesley, 1990.
Martin 1990c	Martin, A. J., Collected Papers on Asynchronous VLSI Design, Technical Report Caltech-CS-TR-90-09, Department of Computer Science, California Institute of Technology, 1990.
McCluskey 1986	McCluskey, E. J., Logic Design Principles: With Emphasis on Testable Semicustom Circuits, Prentice-Hall, Englewood Cliffs, NI, 1986.
McMillan 1993	McMillan, K. L., Symbolic Model Checking, Kluwer Academic Publishers, Boston, 1993.
Mealy 1955	Mealy, G. H., A Method for Synthesizing Sequential Circuits, Bell System Technical J., 34(5) pp. 1045-1079, 1955.
Meng 1989	Meng, T. H.-Y., Brodersen, R. W. and Messerschmit, D. G., Automatic Synthesis of Asynchronous Circuits from High-Level Specifications, IEEE Transactions on Computer

	Aided Design, Vol. 8, No. 11, pp. 1185-1205, November 1989.
Miller 1965	Miller, R. E., Switching Theory, Vol. II: Sequential Circuits and Machines, John Wiley and Sons, New York, NY, 1965.
Molnar 1985	Molnar, C. E., Fang, T.-P., and Rosenberger, F. U., Synthesis of Delay-Insensitive Modules, In Henry Fuchs, Editor, 1985 Chapel Hill Conference of VLSI, pp. 67-86, Computer Science press, 1985.
Moore 1956	Moore, E. F., Gedanlen Experiments on Sequential Machines, Automata Studies, pp. 129-153, 1956.
Mukai 1974	Mukai, Yuzo and Tohma, Yoshihiro, A Method for the Realization of Fail-Safe Asynchronous Sequential Circuits, IEEE Transactions on Computers, C-23(7), pp. 736-739, July 1974.
Muller 1956	Muller, D. E., and Bartky, W. S., A Theory of Asynchronous Circuits I, Digital Computer Laboratory 75, University of Illinois, Nov. 1956.
Muller 1957	Muller, D. E., and Bartky, W. S., A Theory of Asynchronous Circuits II, Digital Computer Laboratory 78, University of Illinois, Nov. 1957.
Muller 1959	Muller, D. E., and Bartky, W. S., A Theory of Asynchronous Circuits, Proc. of International Symposium on the Theory of Switching, Vol. 29 of the Annals of the Computation Laboratory of Harvard University, pp. 204-243, Harvard University press, 1959.
Murata 1989	Murata, T., Petri nets: Properties, Analysis and Applications, Proceedings of IEEE, Vol. 77(4), pp. 541-580, April 1989.
Myers 1993	Myers, C. J. and Meng, T. H.-Y., Synthesis of Timed Asynchronous Circuits, IEEE Transitions on VLSI Systems, pp. 106-119, June 1993.
Myers 1995a	Myers, C. J., Rokicki, T. G., and Meng, T. H.-Y., Automatic Synthesis and Verification of Gate-Level Timed Circuits,

	Technical Report CSL-TR-94-652, Stanford University, January 1995.
Myers 1995b	Myers, Chris J., Computer Aided Synthesis and Verification of Gate-Level Timed Circuits, Ph.D Thesis, Stanford University, October, 1995.
Nanya 1995	Nanya, T., A Quasi-Delay-Insensitive Microprocessor: Titac-I, Proceedings of 1995 Israel Workshop on Asynchronous VLSI, March 1995.
Nielsen 1992	Neilsen, M., Rozenberg, G. and Thiagarajan, P. S., Elementary Transition Systems, Theoretical Computer Science, Vol. 96, pp. 3-33, 1992.
Nowick 1991	Nowick, Steven M., Dill, David L., Synthesis of Asynchronous State Machines Using a Local Clock, In International Conference on Computer Design, ICCD 1991, pp. 192-197, IEEE Computer Society press, 1991.
Nowick 1993	Nowick, Steven M., Automatic Synthesis of Burst-Mode Asynchronous Controllers, Ph.D thesis, Stanford University, Department of Computer Science, 1993.
Pastor 1998	Pastor, E., Cortadella, J., Kondratyev, A. and Roig, O., Structural Methods for the Synthesis of Speed Independent Circuits, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 17, No. 11, pp. 1108-1129, Nov. 1998.
Patil 1974	Patil, S. S., Cellular Arrays for Asynchronous Control, In Proceedings of the ACM 7th Annual Workshop on Microprogramming 1974.
Peeters 1996	Peeters, A. M. G., Single Rail Handshake Circuits, Ph.D thesis, Technische Universiteit Eindhoven, Netherlands, 1996.
Pena 1996	Pena, M. A. and Cortadella, J., Combining Process Algebras and Petri nets for the Specification and Synthesis of Asynchronous Circuits. Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, IEEE Computer Society press, March 1996.

Peterson 1981	Peterson, James L., Petri net Theory and the Modelling of Systems, Prentice-Hall, 1981.
Peterson 1983	Peterson, G., Concurrent Reading While Writing, ACM Transactions on Programming Language and Systems, Vol. 5(1) pp. 46-55, 1983.
Petrify	http://www.lsi.upc.es/~jordic/petrify/
Phillips 1967	Phillips, C. S. E., Networks for Real Time Programming, Computer J., No. 10 (1), pp. 46-52, 1967.
Raynal 1986	Raynal, M., Algorithms for Mutual Exclusion, North Oxford Academic Publishers Ltd., 1986.
Rosenblum 1985	Rosenblum, L. Y., and Yakovlev, A. V., Signal Graphs: From Self-Timed to Timed Ones. In Proceedings of International Workshop on Timed Petri Nets, pp. 199-207, Torino, Italy, IEEE Computer Society Press, July 1985.
Sawin 1974	Sawin, D. H. and Maki, G. K., Asynchronous Sequential Machines Designed for Fault Detection, IEEE Transaction on Computers, C-32(3), pp. 239-249, March 1974.
Seitz 1980	Seitz, Ch., Ideas About Arbiter, Lambda, Vol. 1, pp. 10-14, First Quarter 1980.
Semenov 1997a	Semenov, A., Yakovlev, A., Pastor, E., Pena, M. A. and Cortadella, J., Synthesis of Speed Independent Circuits from STG Unfolding Segment, Proc. 34th ACM/IEEE Design Automation Conference, pp. 16-21, June 1997.
Semenov 1997b	Semenov, A., Verification and Synthesis of Asynchronous Control Circuits Using Petri net Unfoldings, Ph.D thesis, Department of Computing Science, University of Newcastle upon Tyne, 1997.
Semenov 1997c	Semenov, A., Koelmans, A. M., Lloyd, L., and Yakovlev, A., Design an Asynchronous Processor Using Petri nets, IEEE Micro, Vol. 17(2), pp. 54-64, 1997.
Sentovich 1992	Sentovich, E. M., et. al., SIS: A System for Sequential Circuit Synthesis, Memorandum No. UCB/ERL M92/41, University of California, Berkeley, 1992.

Sgroi 2000	Sgroi, M., Lavagno, L. and Sangiovanni-Vincentelli, A., Formal Model for Embedded Systems Design, IEEE Design and Test, Vol. 17(2), pp. 14-27, April-June 2000.
Shang 2000a	Shang, D., Xia, F., and Yakovlev, A., A Self-Timed Asynchronous Data Communication Mechanism, Proc. 1st Annual Postgrad Symp. on Convergence of Telecommunications, Networking and Broadcasting (PGNET2000), Liverpool, John Moores University, EPSRC, pp. 170-176.
Shang 2000b	Shang, D., Xia, F., and Yakovlev, A., An Implementation of A Three-Slot Asynchronous Communication Mechanism Using Self-Timed Circuits, In Alex Yakovlev and Reinder Nouta, Editors, Asynchronous Interfaces: Tools, Techniques and Implementations, pp. 37-44, July 2000.
Shang 2000c	Shang, D., Xia, F., and Yakovlev, A., Testing a Self-Timed Asynchronous Communication Mechanism (ACM) VLSI Chip, Proc. 9th Asynchronous UK Forum, Cambridge University, 18-19th December 2000.
Shang 2001a	Shang, D., Xia, F., and Yakovlev, A., Testing a Self-Timed Asynchronous Communication Mechanism (ACM) VLSI Chip, IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS) 2001, pp. 53-56, Gyor, Hungary, 18-20 April 2001.
Shang 2001b	Shang, D., Xia, F., and Yakovlev, A., Asynchronous Circuit Synthesis via Direct Translation, 11th UK Asynchronous Forum, University of Cambridge Computer Laboratory, 17-18 December 2001.
Shang 2002a	Shang, D., Xia, F., and Yakovlev, A., Asynchronous Circuit Synthesis via Direct Translation, ISCAS 2002, IEEE International Symposium on Circuits and Systems, Scottsdale, Arizona, Volume 3, pp. 369-372, May 2002.
Simpson 1979	Simpson, H. R., and Jackson, K., Process Synchronisation in MASCOT, Computer Journal, 1979, 22 (4), pp. 332-345.

Simpson 1986	Simpson, H. R., The MASCOT Method, Software Engineering Journal, 1986, 1 (3), pp. 103-120.
Simpson 1990	Simpson, H. R., Four-slot Fully Asynchronous Communication Mechanism, IEE Proceedings, Vol. 137, Pt. E, No.1, pp. 17-30, January 1990.
Simpson 1994	Simpson, H. R., Methodological and Notational Conventions in DORIS Real Time Networks, Dynamics Division, Abe, 11 February 1994.
Simpson 2000	Simpson, H. R., Campbell, E., Real Time Network Architecture: Principles and Practices, Proc. AINT'2000, Asynchronous Interfaces: Tools, Techniques and Implementations, pp. 5 and Handouts, TU Delft, The Netherlands, July 19-20, 2000.
Sotiriou 2001	Sotiriou, Christos Panagiotis, Design of an Asynchronous Processor, Ph.D thesis, University of Edinburgh, 2001.
Sparsø 2001	Sparsø, Jens and Furber, Steve Editors, Principles of Asynchronous Circuit Design: A Systems Perspective, Kluwer Academic Publishers, 2001.
Spice3	http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE.
Sproull 1986	Sproull, R. F., and Sutherland, I. E., Asynchronous Systems, Sutherland, Sproull and Associates, Palo Alto, 1986, Vol. I: Introduction, Vol. II: Logic Effort and Asynchronous Modules, Vol. III: Case Studies.
Sutherland 1989	Sutherland, Ivan E., Micropipelines, Communications of the ACM, 32(6):720-738, June 1989.
Sutherland 2002	Sutherland, I. E., and Ebergen, J., Computer With Clocks, Scientific American, July 2002.
Stevens 1999	Stevens, Ken, Ginosar, Ran and Rotem, Shai, Relative Timing. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 208-218, April 1999.
Tromp 1989	Tromp, J., How to Construct an Atomic Variable, Proc. 3rd Int. Workshop on Distributed Algorithms, Nice, LNCS, Springer Verlag, pp. 292-302, 1989.

Unger 1969	Unger, S. H., Asynchronous Sequential Switching Circuits, Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.
Varshavsky 1990	Varshavsky, V. I., Kishinevsky, M. A., Marakhovsky, V. B., Peschansky, V. A., Rosenblum, L. Y., Taubin, A. R. and Tzirlin, B. S., Self-Timed Control of Concurrent Processes, Kluwer Academic Publisher, 1990. (Russian edition: 1986).
Varshavsky 1996	Varshavsky, V. and Marakhowsky, V., Hardware Support for Discrete Event Coordination, Proc. of International Workshop on Discrete Event Systems (WODES'96), pp. 332-340, August 1996, Edinburgh, U.K.
Verhoeff 1995	Verhoeff, T., Encyclopedia of Delay-Insensitive Systems, Eindhoven University of Technology, The Netherlands, 1995-1998. http://edis.win.tue.nl/edis.html.
Versify	University of Politecnica de Catalunya, http://www.ac.upc.es/vlsi/versify/.
Vidyasankar 1990	Vidyasankar, K., Concurrent Reading While Writing Revisited, Distributed Computing, Vol. 4(2), pp. 81-85, 1990.
Wang 1991	Wang, Francis C., Digital Circuit Testing: A Guide to DFT and Other Techniques, Academic Press, Inc., San Diego, California 92101.
Williams 1948	Williams, F. C., Kilburn, T., Electronic Digital Computers, Nature 162, pp. 487, September 1948. URL: http://www.computer50.org.
Williams 1951	Williams, F. C., Kilburn, T., and Toothill, G. C., Universal High-Speed Digital Computer: A Small Scale Experimental Machine, In Proceedings of the IEE, pp. 487, Febryary 1951. URL: http://www.computer50.org.
Williams 1982	Williams, Thomas W., Design for Testability – A Survey, IEEE Transactions on Computers, Vol. C-31, No. 1, January 1982.

Williams 1991	Williams, T. E., Self-Timed Rings and Their Application to Division, Ph.D thesis, Computer Systems Laboratory, Stanford University, 1991.
Xia 1997	Xia, F. Clark, I. G., and Davies, A. C., Petri-net Based Investigation of Synchronisation Free Interprocess Communication in Shared-Memory Real-Time System, Proceedings of 2nd UK Asynchronous Forum, Newcastle upon Tyne, UK, July 1-2, 1997.
Xia 1999a	Xia, F., Shang, D., Yakovlev, A., and Koelmans, A., An Asynchronous Communication Mechanism Using Self-Timed Circuits, 6th UK Asynchronous Forum, University of Manchester, 12-13th July 1999.
Xia 1999b	Xia, F., Yakovlev, A. and Clark, I. G., Testing the Data Freshness Properties of Asynchronous Communication Mechanism, Proc. of the 7th UK Asynchronous Forum, Newcastle upon Tyne, U.K., Dec. 20-21 1999.
Xia 2000a	Xia, F., Supporting the MASCOT Method with Petri net Techniques for Real-Time Systems Development, Ph.D. thesis, London University, King’s College, January 2000. (Downloadable from http://www.eee.kcl.ac.uk/~comfort/)
Xia 2000b	Xia, F., Yakovlev, A., Shang, D., Bystrov, A., Koelmans, A., and Kinniment, D. J., Asynchronous Communication Mechanisms Using Self-Timed Circuits. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 150-159. IEEE Computer Society Press, April 2000.
Xia 2002	Xia, F., Yakovlev, A., Clark, I. G., and Shang, D., Asynchronous Communication Mechanisms: Classification and Hardware Implementations, MPCS'02, Fourth International Conference on Massively Parallel Computer Systems, Sponsored by Euromicro, 10-12 April 2002, Ischia, Italy.

Xia 2002b	Xia, F., Yakovlev, A., Clark, Ian G., and Shang, D., Data Communication in System with Heterogeneous Timing, IEEE Micro, Vol 22, Part 6, pp. 48-69, 2002.
Yakovlev 1995	Yakovlev, A., Varshavsky, V., Marakhovsky, V., and Semenov, A., Designing an Asynchronous Pipeline Token Ring Interface, Proc. of 2nd Working Conference on Asynchronous Design Methodologies, pp. 32-41, IEEE Comp. Society Press, London, May 1995.
Yakovlev 1996a	Yakovlev, A., Lavagno, L., and Sangiovanni-Vincentelli, A., A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis, Formal Methods in System Design (Kluwer), Vol. 9, No. 3, pp. 139-188, Nov. 1996.
Yakovlev 1996b	Yakovlev, A., Koelmans, A. M., Semenov, A., and Kinnement, D. J., Modelling, Analysis and Synthesis of Asynchronous Control Circuits Using Petri nets, Integration, the VLSI Journal, Vol. 21(3), pp. 143-170, December 1996.
Yakovlev 1998	Yakovlev, A. V. and Koelmans, A. M., Petri nets and Digital Hardware Design, In Lectures on Petri nets II: Applications, Advances in Petri nets, Vol. 1492, pp. 154-236, 1998.
Yakovlev 2000	Yakovlev, A., Gomes, L. and Lavagno, L. Editors, Hardware Design and Petri Nets, Kluwer Academic Publishers, March 2000
Yakovlev 2001	Yakovlev, A., Xia, F., and Shang, D., Synthesis and Implementation of a Signal-Type Asynchronous Data Communication Mechanism. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 127-136. IEEE Computer Society Press, March 2001.
Yakovlev 2002a	Yakovlev, A., Burns, F., Bystrov, A., Koelmans, A., Krenz, R., Shang, D., Behavioural Synthesis of Asynchronous Controllers: A Case Study With a Self-Timed

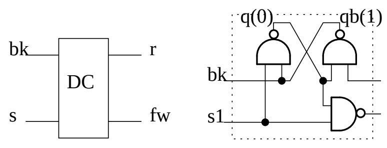
	Communication Channel, Second ACiD-WG Workshop of the European Commission's Fifth Framework Programme, Munich, Germany, 28-29 January 2002.
Young 1999	Young, F. C. D., Stevens, K. S., and Graham, R. P., Timed Logic Conformance and its Application, in 1999 International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU 99), ACM/IEEE, March 1999.
Yun 1992a	Yun, K., and Dill, D., Automatic Synthesis of 3D Asynchronous State Machines, In Proceedings of ICCD, pp. 576-580, 1992.
Yun 1992b	Yun, K., Dill, D., and Nowick, S. M., Synthesis of 3D Asynchronous State Machines, In Proceedings of ICCD, pp. 346-350, 1992.

Appendix *A*

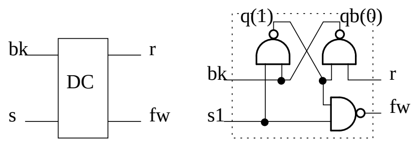
Some useful components for asynchronous circuit designs are listed, which can be used as an extension to a standard library.

Specially, we present all DCs which are used in our PN2DCs tool. As for the VDCs, they are basically the same as the DCs. The difference is that in DCs, the *q* and *qb* are internal state signals. However, in VDCs, they are not only the internal state signals but also the output signals. But only one, either *q* or *qb*, is used as an output signal. We do not list them in this Appendix.

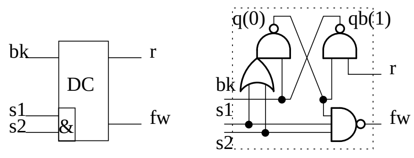
Apart form the DCs (VDCs), some other useful components are also presented, such as Mutex, MSLatch, SRLatch with completion diction, and son on.



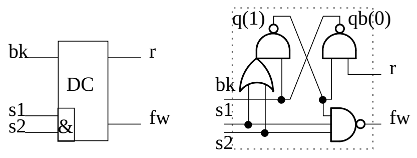
DCsetreset_0



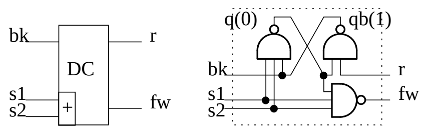
DCsetreset_1



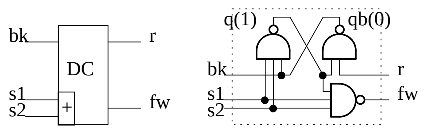
DCsetAND2reset_0



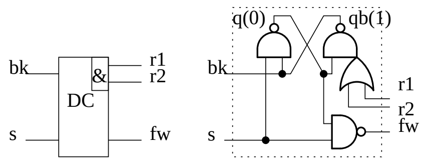
DCsetAND2reset_1



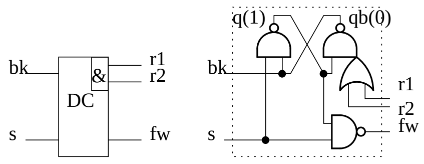
DCsetOR2reset_0



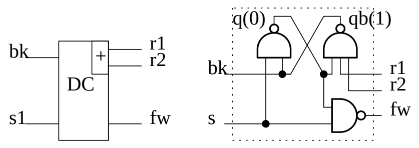
DCsetOR2reset_1



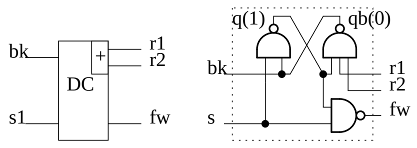
DCsetresetAND2_0



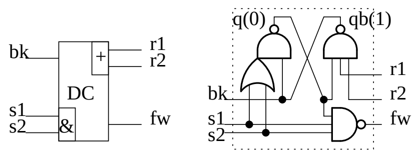
DCsetresetAND2_1



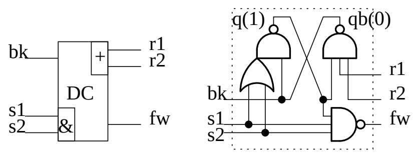
DCsetresetOR2_0



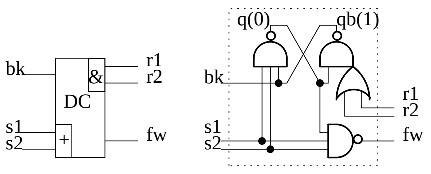
DCsetresetOR2_1



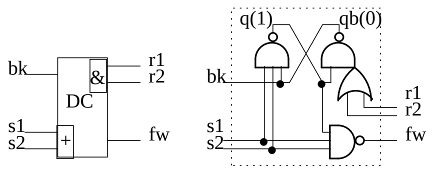
DCsetAND2resetOR2_0



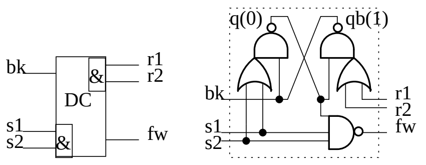
DCsetAND2resetOR2_1



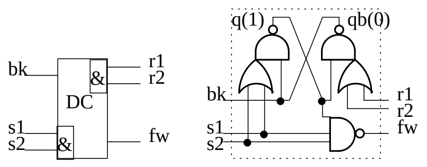
DCsetOR2resetAND2_0



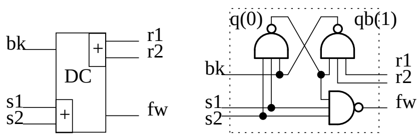
DCsetOR2resetAND2_1



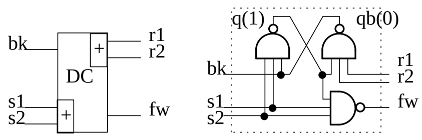
DCsetAND2resetAND2_0



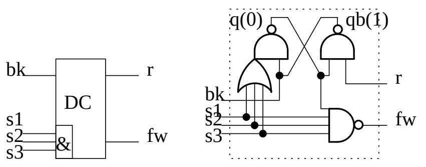
DCsetAND2resetAND2_1



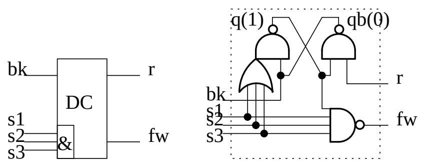
DCsetOR2resetOR2_0



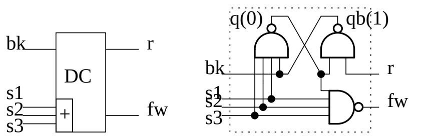
DCsetOR2resetOR2_1



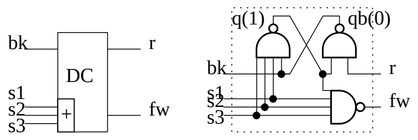
DCsetAND3reset_0



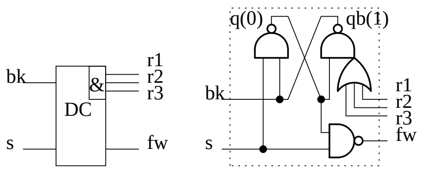
DCsetAND3reset_1



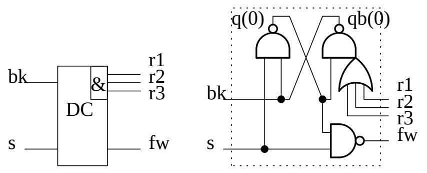
DCsetOR3reset_0



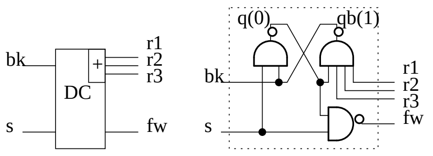
DCsetOR3reset_1



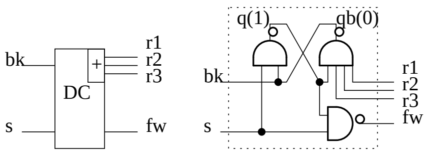
DCsetresetAND3_0



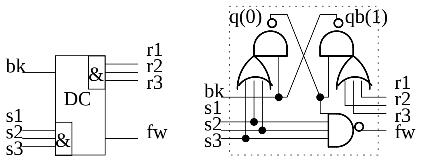
DCsetresetAND3_1



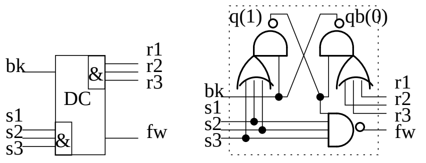
DCsetresetOR3_0



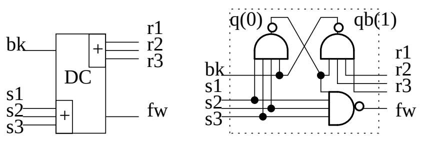
DCsetresetOR3_1



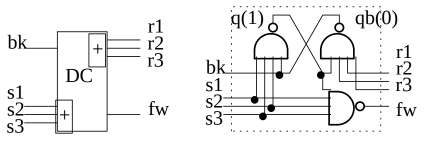
DCsetAND3resetAND3_0



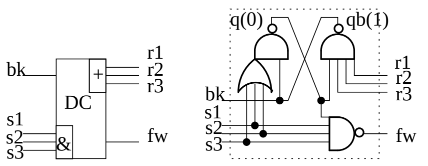
DCsetAND3resetAND3_1



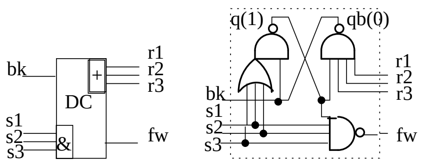
DCsetOR3resetOR3_0



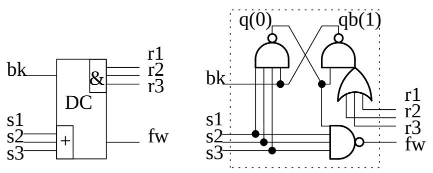
DCsetOR3resetOR3_1



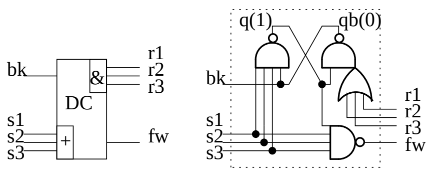
DCsetAND3resetOR3_0



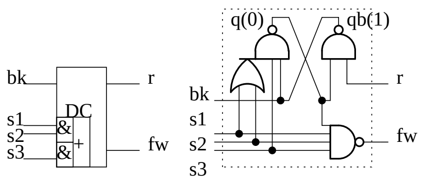
DCsetAND3resetOR3_1



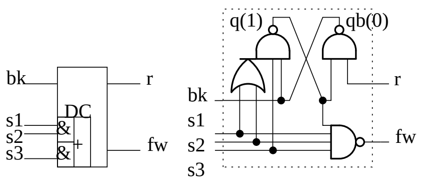
DCsetOR3resetAND3_0



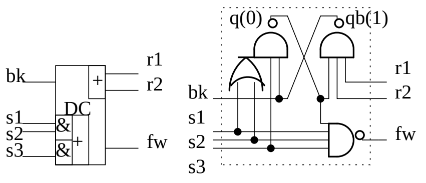
DCsetOR3resetAND3_1



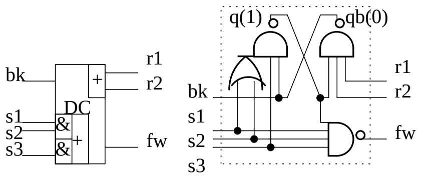
DCsetAO21reset_0



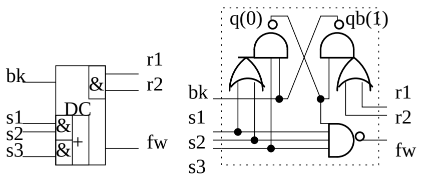
DCsetAO21reset_1



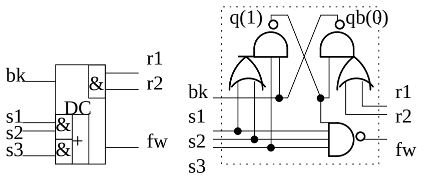
DCsetAO21resetOR2_0



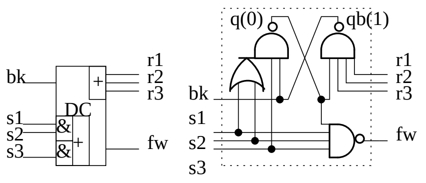
DCsetAO21resetOR2_1



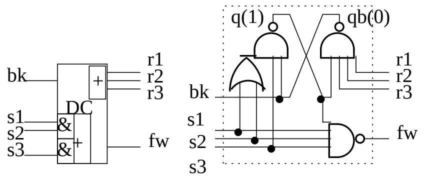
DCsetAO21resetAND2_0



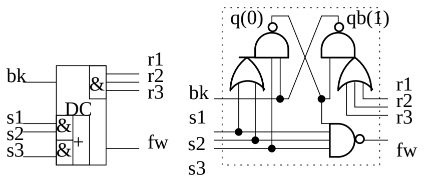
DCsetAO21resetAND2_1



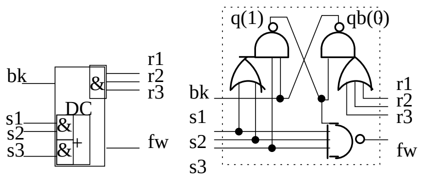
DCsetAO21resetOR3_0



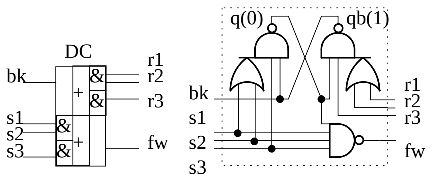
DCsetAO21resetOR3_1



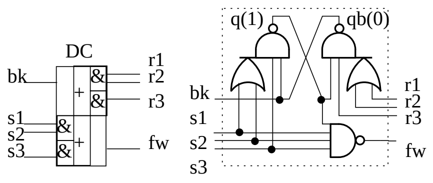
DCsetAO21resetAND3_0



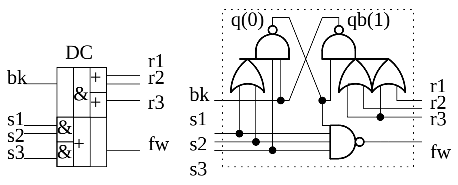
DCsetAO21resetAND3_1



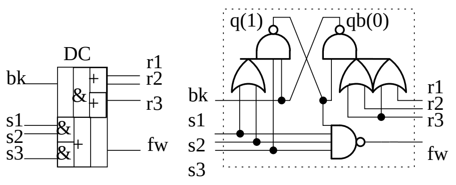
DCsetAO21resetAO21_0



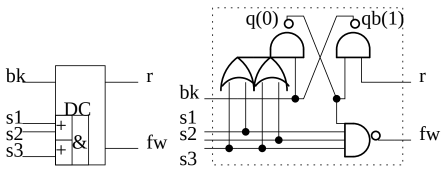
DCsetAO21resetAO21_1



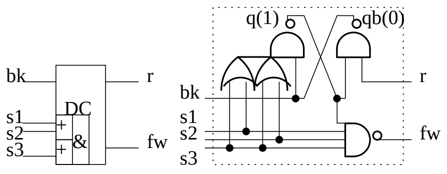
DCsetAO21resetOA21_0



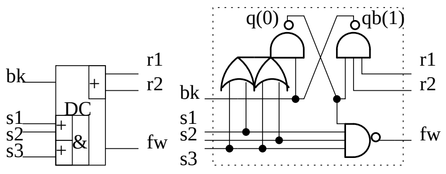
DCsetAO21resetOA21_1



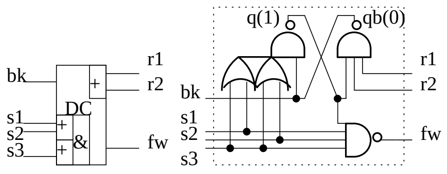
DCsetOA21reset_0



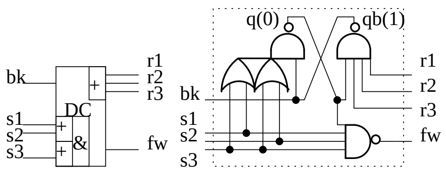
DCsetOA21reset_1



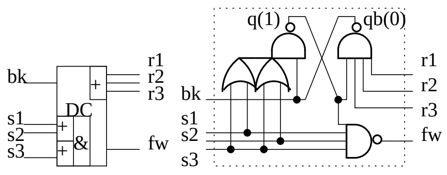
DCsetOA21resetOR2_0



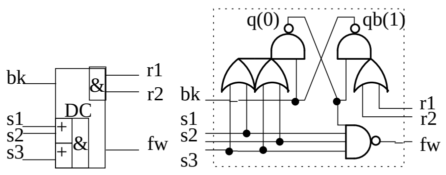
DCsetOA21resetOR2_1



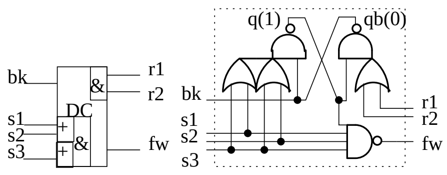
DCsetOA21resetOR3_0



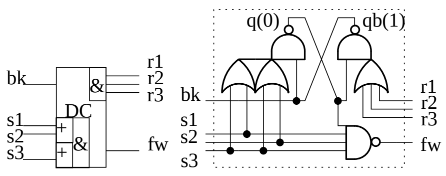
DCsetOA21resetOR3_1



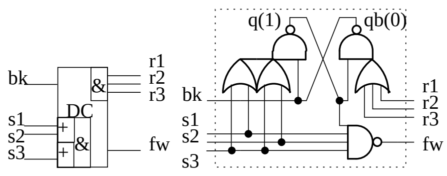
DCsetOA21resetAND2_0



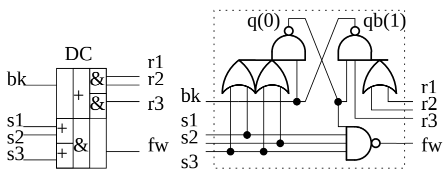
DCsetOA21resetAND2_1



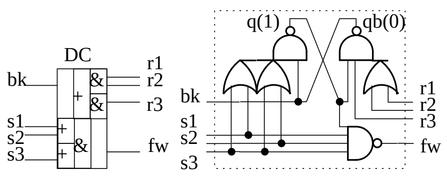
DCsetOA21resetAND3_0



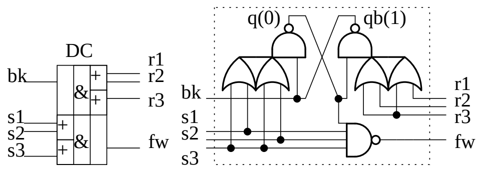
DCsetOA21resetAND3_1



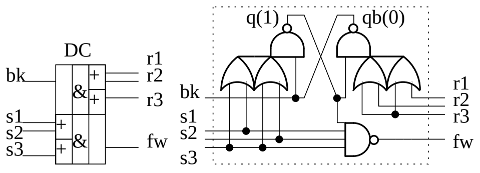
DCsetOA21resetAO21_0



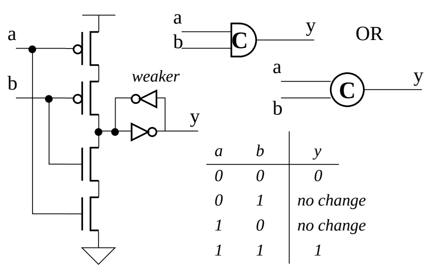
DCsetOA21resetAO21_1



DCsetOA21resetOA21_0

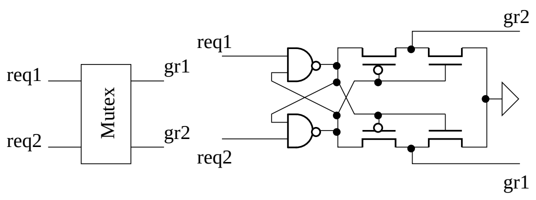


DCsetOA21resetOA21_1

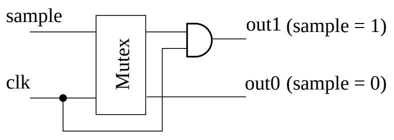


a	b	y
0	0	0
0	1	no change
1	0	no change
1	1	1

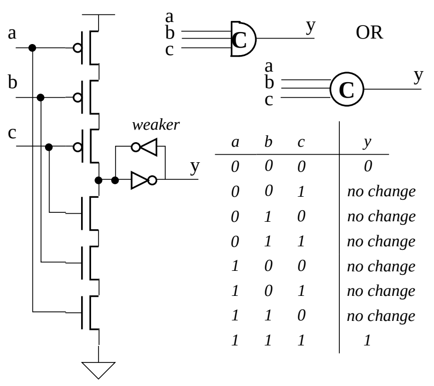
Cele2



Mux

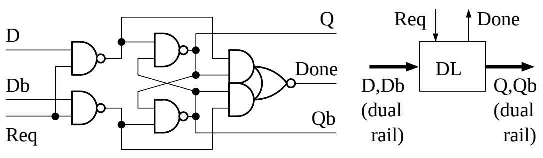


Sync

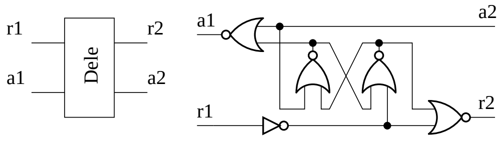


a	b	c	y
0	0	0	0
0	0	1	no change
0	1	0	no change
0	1	1	no change
1	0	0	no change
1	0	1	no change
1	1	0	no change
1	1	1	1

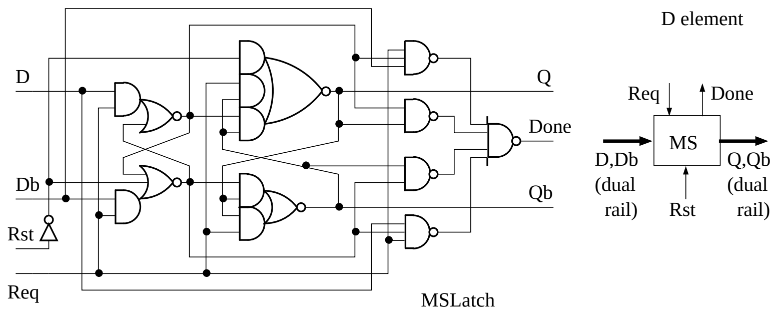
Cele3



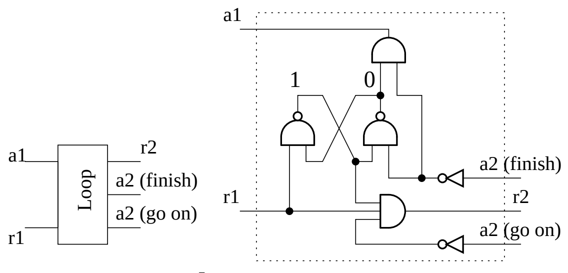
DLatch



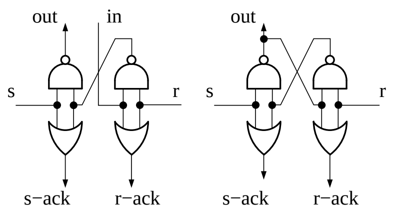
D element



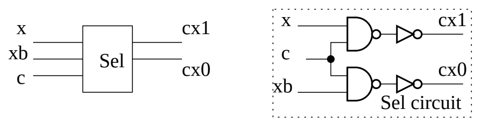
MSLatch



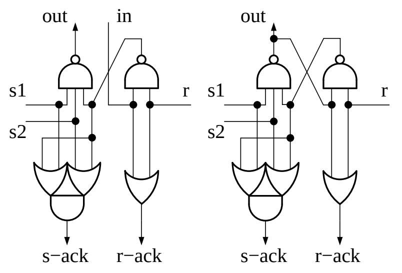
Loop



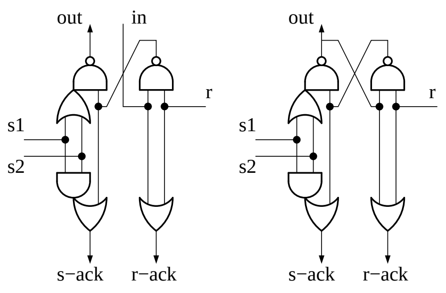
Simple SR latches with completion detection



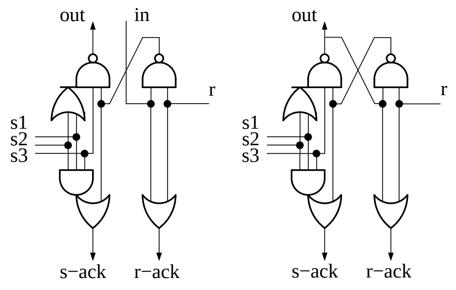
Sel



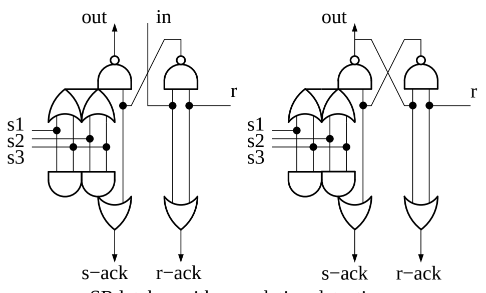
SR latches with completion detection



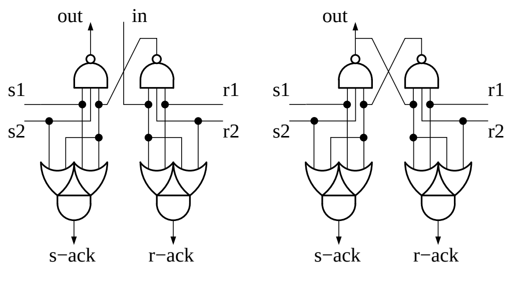
SR latches with completion detection



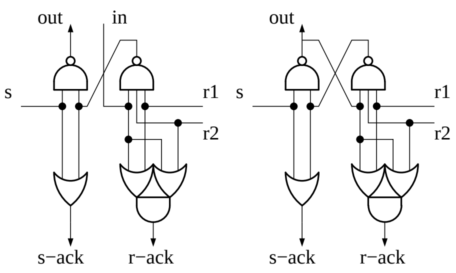
SR latches with completion detection



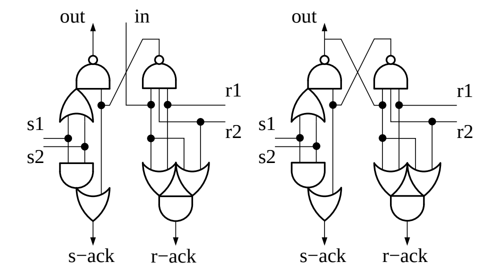
SR latches with completion detection



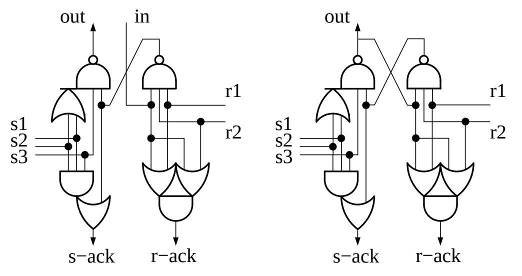
SR latches with completion detection



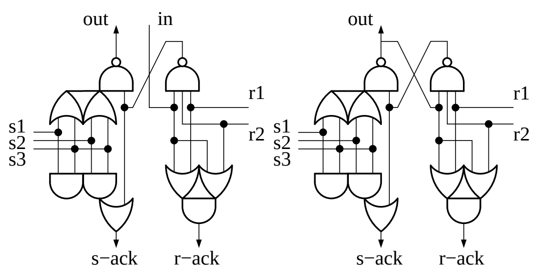
SR latches with completion detection



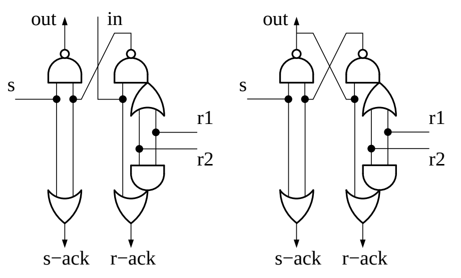
SR latches with completion detection



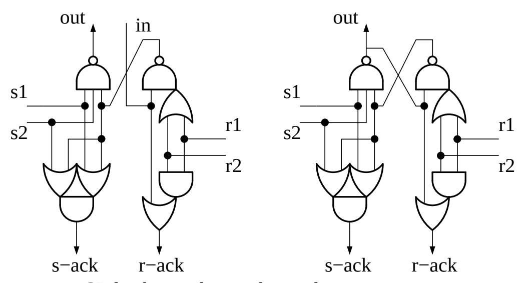
SR latches with completion detection



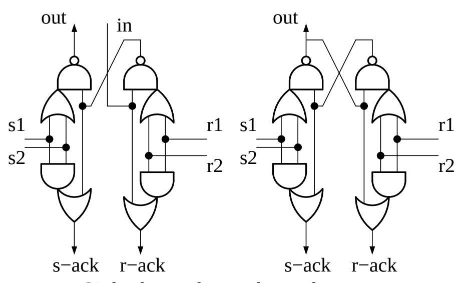
SR latches with completion detection



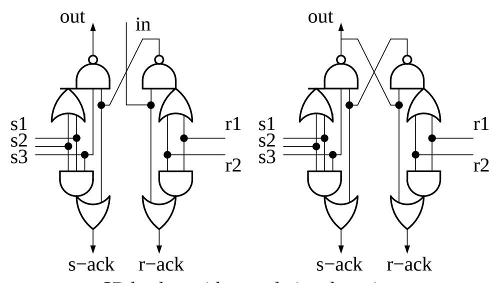
SR latches with completion detection



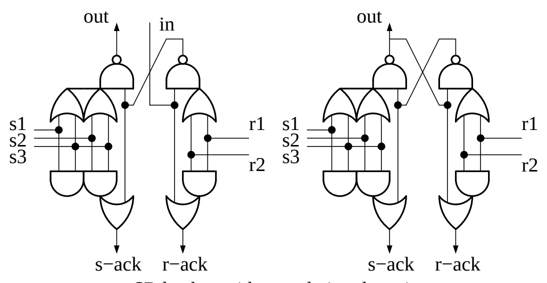
SR latches with completion detection



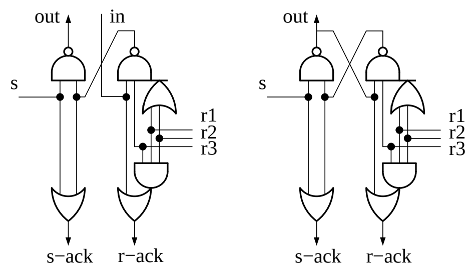
SR latches with completion detection



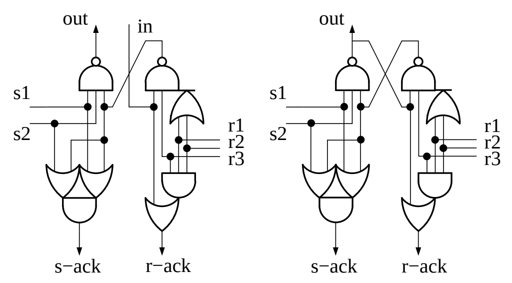
SR latches with completion detection



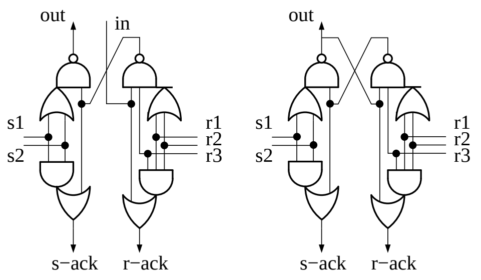
SR latches with completion detection



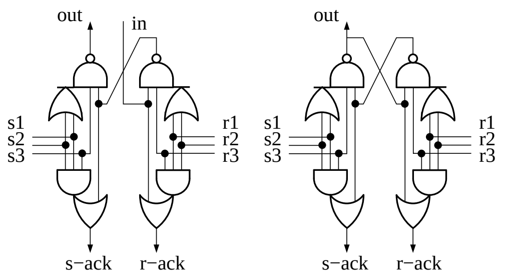
SR latches with completion detection



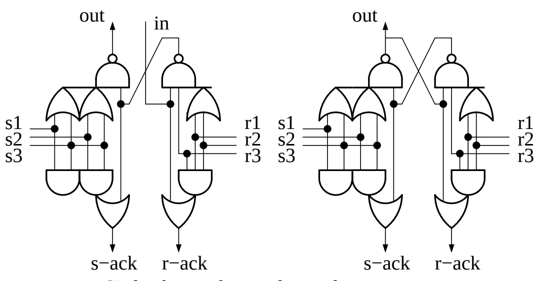
SR latches with completion detection



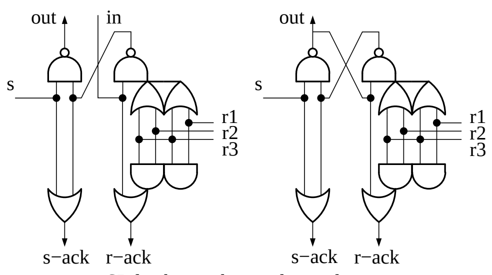
SR latches with completion detection



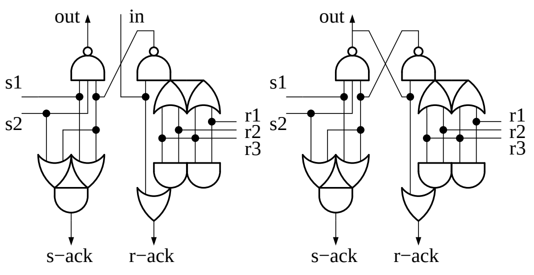
SR latches with completion detection



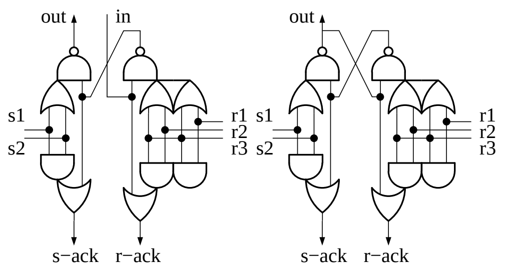
SR latches with completion detection



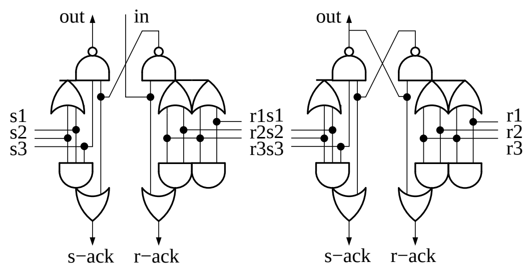
SR latches with completion detection



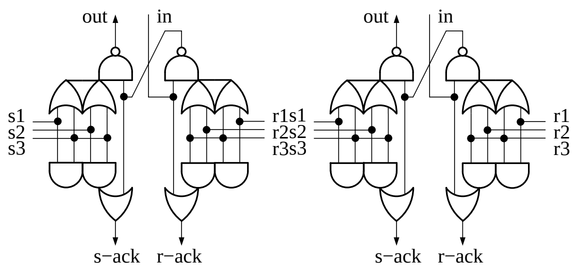
SR latches with completion detection



SR latches with completion detection



SR latches with completion detection



SR latches with completion detection