Technical Report

UCAM-CL-TR-151
ISSN 1476-2986

Number 151

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Formalising an integrated circuit design style in higher order logic

Inderpreet-Singh Dhingra

November 1988

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

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

This technical report is based on a dissertation submitted March 1988 by the author for the degree of Doctor of Philosophy to the University of Cambridge, King's College.

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

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

ISSN 1476-2986

Abstract

If the activities of an integrated circuit designer are examined, we find that rather than keeping track of all the details, he uses simple rules of thumb which have been refined with experience. These rules of thumb are guidelines for deciding which building blocks to use and how they are to be connected. This thesis gives a formal foundation, in higher order logic, to the design rules of a dynamic CMOS integrated circuit design style.

Correctness statements for the library of basic elements are formulated. These statements are based on a small number of definitions which define the behaviour of transistors and capacitors and the necessary axiomatisation of the four-valued algebra for signals. The correctness statements of large and complex circuits are then derived from the library of previously proved correctness statements, using logical inference rules instead of the rules of thumb. For example, one gate from the library can drive another only if its output constraints are satisfied by the input constraints of the gate that it drives. In formalising the design rules, these constraints are captured as predicates and are part of the correctness statements of these gates. So when two gates are to be connected, it is only necessary to check that the predicates match. These ideas are fairly general and widely applicable for formalising the rules of many systems.

A number of worked examples are presented based on these formal techniques. Proofs are presented at various stages of development to show how the correctness statement for a device evolves and how the proof is constructed. In particular it is demonstrated how such formal techniques can help improve and sharpen the final specifications.

As a major case study to test all these techniques, a new design for a digital phase-locked loop is presented. This has been designed down to the gate level using the above dynamic design style, and has been described and simulated using ELLA. Some of the subcomponents have been formally verified down to the detailed circuit level while others have merely been specified without formal proofs of correctness. An informal proof of correctness of this device is also presented based on the formal specifications of the various submodules.

Acknowledgments

During the time of this project, it is both pleasing and reassuring to note the number of people who have freely given their help and support. My greatest debt is to my supervisor Mike Gordon. This work could never have taken shape without his continued guidance and support. His patience during my earlier days and his ever optimistic manner has brought me through some of the most difficult times. He made valuable suggestions on earlier drafts of this thesis, and was the source of many excellent discussions.

In addition, I would like to express my sincerest thanks to Graham Birtwistle and Tom Melham. They diligently read through various drafts of this thesis and made valuable written comments. Tom's insistence on clear writing has set a standard which I will always try to aim for.

This work could not have started without the financial support of Racal Research and the British Science and Engineering Council. Racal Research has provided an excellent start to my career. The technical advice and individual guidance received while working there before the start of this project, and during the many visits thereafter, greatly helped in this project. Of the innumerable people who helped, I must explicitly acknowledge Bob Chapman and Dave Orton, who were the source of many stimulating discussions. I would also like to thank Racal Research for the funds to attend conferences.

Thanks also to Professor Roger Needham of the Computer Laboratory for providing such a stimulating and cheerful work environment. The hardware verification research group has proved an excellent platform to test ideas before making them public. Special thanks go to the following members of this group: Albert Camilleri, Avra Cohn, Thomas Forster, Don Gaubatz, Mike Gordon, Roger Hale, John Herbert, Jeff Joyce, Miriam Leeser, Tom Melham, Ben Moszkowski, and Glynn Winskel. Indeed, many thanks are due to the all the members of the Computer Laboratory for making this a friendly place to work.

During my year at the University of Calgary, I was looked after extremely well. For this I must acknowledge the kind generousity of Graham Birtwistle, and the

help of the VLSI research group: Brian, Han, Konrad, Mark, Mike, Todd, and others. I would also like to express my gratitude to the Alberta Microelectronics Center for allowing me access to their machines. In particular, Brian, Earl, Wallace, and Dale made my life there very enjoyable. Thanks also for all the free coffee and doughnuts guys.

Thanks are also due to the many people at the various sites for putting up with my persistent questioning about the systems etc.. These include Graham, Martin, Mike, Piete, and Steve at Cambridge, Dave, Keith, and Terry at the University of Calgary, and Earl and Wallace at the Alberta Microelectronics Center.

Finally, I would like to thank Humphry, with whom I shared a house for nearly three years. Thanks for all the cooked breakfasts, and for allowing me to test some of my crazy ideas. I can now confidently state that tomato skin is not a superconductor!

Contents

Chapter 1

Introduction

1.1 Background

On 23*rd* December 1947 at the Bell Laboratories in Murry Hill, the first semiconductor transistor was invented by Bardeen, Brattain and Shockley. It was kept a closely guarded secret until just before the first publication in 1948 [Bardeen 48]. It took a further five years before transistors were manufactured on a large scale. In 1953 the best transistors cost around $8US [Braun 78]. Today it is possible to buy memory chips containing over a million transistors on a single chip for around $10US, or about 0.00002 cents per transistor if converted to 1953 values!

But the development of the transistor was not without its problems. Manufacturing was difficult, and even more difficult was getting two transistor characteristics to match. The early transistors, i.e. point contact transistors, were very noisy and highly unreliable; and they tended to deteriorate rapidly with not too extreme temperature and humidity conditions. A story has it that a factory in England, upon retiring one of its senior manufacturing workers, found the yield of its point contact transistors dropped dramatically. This was due to the fact that the retired worker knew just how hard to tap the electrode to make the point contact. Too hard and the electrode went too deep into the semiconductor, and too light meant no contact; either way the transistor would not function.

It was fortunate that industry did not have to endure this for too long. By April 1952 the junction transistor was in manufacture, even though it was only at a rate of less than a hundred per month. Over the next few years, with the advent of zone refining and improved manufacturing techniques, the industry grew rapidly. The first integrated circuit was made in October 1958 by Jack Kilby of

Texas Instruments, and the first full scale microprocessor on a single chip, the Intel 4004, appeared in 1972.

By comparison, the developments of digital hardware design techniques have been slow. The complexity of the hardware has increased over the years, but the design tools have far from kept pace with this "silicon chip" revolution. The basic design techniques used today differ little from those used to design some of the early digital systems, which used valves and relays as switches instead of transistors. Today, Boolean logic [Boole 54] is used to design combinational circuits, and the methods of Mealy [Mealy 55] and Moore [Moore 64] are used to design sequential circuits.

With the advent of computers, designers have been able to design more complex systems, since they can be simulated on computers and so debugged to some extent before manufacture. The basic models used for the primitive devices have improved and hence the accuracy of simulators [Bryant 81]. However, for large systems, exhaustive simulation is prohibitively expensive, and requires exponentially large simulation time even with today's fastest computers. At IBM Research, scientists use the YSE machine, which is designed to simulate circuitry at the primitive component level rather than at the architectural level. This is a highly parallel processor, able to simulate several hundred times faster than any uni-processor machine. But even with this sort of specialised computing power it may still be not viable to simulate very large systems.

Due to these seemingly unsurmountable problems, designers have been forced to look in other directions to verify designs before building them. Even though the cost of manufacturing is coming down, it is still important to have confidence in the design before it is committed to silicon. This is essential since it is not always possible to test every aspect of the design in the time available. Whilst many applications can tolerate some errors, there are others where even minor flaws could be dangerous or too expensive to tolerate. These include:

- *Safety critical applications:*
  Increasingly, integrated circuits of enormous complexity are being used in areas where errors in these circuits could lead to loss of life. Examples include flight control systems, railway signaling systems, medical life-support systems such as pacemakers, military applications, nuclear plant controllers, and anti-lock braking system in cars (see [Cullyer 88]).

- *Remotely sited applications:*
  Applications where access to the systems is difficult would benefit from having the design 100% correct. This is simply because the cost of repairing or

replacing faulty components would be too high. Examples include systems installed in arctic regions, satellites, and systems installed on oil and gas pipelines.

- *Volume production applications:*
  Many industries manufacture systems for the mass market. If the design of a device in such systems is faulty then the cost of recalling and repairing it would be very high. Examples include circuits in automobiles, in telecommunication systems, and in fact any domestic appliance which uses integrated circuits.

With such important applications as these, it is necessary to ensure that a design is correct. To achieve this, attention has turned to formal methods for specifying and verifying the correctness of circuit designs.

1.2 Hardware Specification and Verification using Formal Methods

Work in this area has progressed considerably over the last two decades, and several sites are now actively involved in research in this field. Some of this work is briefly described in this section. An overview is first given of what is meant by "hardware specification and verification using formal methods," and what constitutes a proof of correctness.

1.2.1 What is a Proof of Correctness?

The idea of specifying the behaviour of hardware devices using boolean algebra is in fact quite old. Indeed, designers use boolean algebra all the time to communicate the behaviour of small circuits to other designers. A trivial example of this is the behaviour of a multiplexor circuit, which can be stated as follows:

$$op \;=\; (ctl \wedge a) \vee (\sim ctl \wedge b) \tag{1.1}$$

where the operators $\sim$, $\wedge$, and $\vee$, represent logical negation, conjunction and disjunction respectively.

Equation 1.1 above can be referred to as the *specification* for the multiplexor circuit, because it contains all the information necessary to understand its top level

behaviour. Boolean algebra is also used for deriving the top level specification from the specifications of the lower level components that constitute the circuit. The steps of this derivation together constitute a proof of the correctness of the circuit (see [Gordon 85c]). Each line in this proof is either a hypothesis, or is derived from the previous lines. The final result is the derived theorem stating the top level behaviour of the circuit. As an example consider the proof for the multiplexor device. A circuit that implements the multiplexor device is shown in figure 1.1.

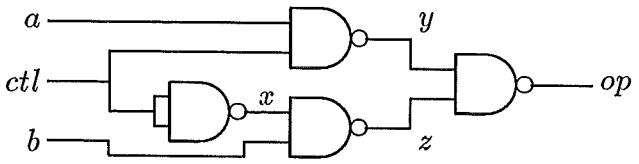


Figure 1.1: A simple multiplexor circuit

The proof of this multiplexor device is given below. The final result in this proof exactly matches the specification of this device.

1.	$x = \sim(ctl \wedge ctl)$	[by definition of Nand]
2.	$y = \sim(ctl \wedge a)$	[by definition of Nand]
3.	$z = \sim(x \wedge a)$	[by definition of Nand]
4.	$op = \sim(y \wedge z)$	[by definition of Nand]
5.	$x = \sim ctl$	[by 1 and the law $p \wedge p = p$]
6.	$z = \sim(\sim ctl \wedge b)$	[substituting 5 into 3]
7.	$op = \sim(\sim(ctl \wedge a) \wedge \sim(\sim ctl \wedge b))$	[substituting 2 & 6 into 4]
8.	$op = \sim\sim(ctl \wedge a) \vee \sim\sim(\sim ctl \wedge b)$	[by law $\sim(p \wedge q) = \sim p \vee \sim q$]
9.	$op = (ctl \wedge a) \vee (\sim ctl \wedge b)$	[by law $\sim\sim p = p$]

The important difference between this style of verifying the correctness of a circuit and techniques involving hardware description languages and simulation, is that the first supports *formal* reasoning. A validation technique which involves formal methods can be summarised as follows:

1. Write the top level specification in some formal language.

2. Design the circuit and describe it in the same language as that used for the specification.

3. By using the inference rules of this language, mathematically prove that the circuit description meets its specification.

In the above example the specification is given in equation 1.1, the implementation is given in figure 1.1, and the proof is given by the 9 steps shown above. This proof is an extremely simple one and could be done on the back of an envelope. But the proofs of real devices can be thousands of lines long and very complex, so different techniques are required to manage them. Several different approaches in this direction are briefly outlined below.

1.2.2 Related Work

Techniques used for the complete verification of hardware without use of simulation are briefly summarised in this section. Summaries of only the most relevant works are given, together with pointers to others.

The early work of Milner and others on the LCF project [Gordon 79] inspired much effort in the area of mechanised theorem proving. A specialised language was developed for specifying and verifying hardware [Gordon 82] which led to the development of the LCF-LSM theorem proving system [Gordon 83a]. Many examples were done using this system including the verification of a simple computer [Gordon 83b]. Many improvements to the LCF system have been made over the years, including an improved rewriting package [Paulson 83a], and a new tactics package [Paulson 83b]. Hanna and Daeche then independently developed the VERITAS theorem prover based on higher-order logic [Hanna 86b,Hanna 86a]. With the improved expressive power of higher-order logic becoming increasingly attractive, and with the experience gained from the LCF family of theorem provers, Gordon then developed the HOL theorem proving system [Gordon 85b,Gordon 88], which forms the basis of the work done in this thesis. A number of examples have been successfully completed which demonstrate the use of higher-order logic as a vehicle for specification and verification. These include the verification at the detailed timing level of a D-flip-flop implemented in logic gates (see [Hanna 86b] and [Herbert 86] for proofs done in the VERITAS and the HOL systems respectively), the verification of a ring interface chip [Gordon 85a], re-proof of the computer in HOL [Joyce 88], and the first level proof of correctness of VIPER, the first industrial microprocessor [Cohn 88].

The systems mentioned above are all based on general theorem provers, where the proof is done manually and the system merely does the housekeeping. In the direction of automated theorem provers is the work of Boyer and Moore [Boyer 79]. This has been used by Hunt to verify the correctness of a 16-bit microprocessor [Hunt 87]. The underlying logic of this theorem prover is first order predicate logic without quantifiers. In principle proofs are done automatically by this

system, but in practice the theorem prover needs to be guided considerably by carefully requesting simpler theorems to be proved first. Barrow's VERIFY system [Barrow 84] is another example of an automatic theorem prover. The underlying model used by this system is that of finite state machines, based on Gordon's LSM language [Gordon 83a]. In both of these systems, logic gates form the set of primitive devices on the basis of which hardware verification is done.

Another general formalism used for hardware verification is temporal logic. An interesting variant is that developed by Moszkowski known as Interval Temporal Logic (ITL) [Moszkowski 83a]. ITL has been used to formulate the specifications of a bit-sliced microprocessor [Moszkowski 83b]. Proofs using ITL were initially done by hand. Recent work by Hale [Hale 88b] shows that the HOL system can be used to mechanise proofs in ITL. A subset of ITL has also been developed as an executable language known as Tempura [Moszkowski 86]. More recently, Leeser has used a variant of this formalism together with Prolog to reason about circuits down to the detailed transistor level [Leeser 87].

On the side of more specialised formalisms are CIRCAL and μFP.

— The early work of Milne and Milner on Concurrent Processes [Milne 79] inspired a number of calculi. Milne went on to develop CIRCAL [Milne 83a,Milne 83b], while Milner went in a slightly different direction and developed the Calculus of Communicating Processes (CCS) [Milner 80,Milner 82]. In the CIRCAL framework, the structure of hardware devices is represented hierarchically, with communication between components being done through commonly named ports. Behaviour in this framework is described as a sequence of events on the external ports of devices. Operators are provided for the composition and the hiding of ports. Several examples have been completed using this framework, including a simple CRT controller by Traub in [Traub 87]. Here Traub also presents the various temporal concepts needed to model different granularities of time and means of moving between them. A Lisp based environment for doing proofs in this formalism has also been developed [Traub 83].

— Sheeran uses μFP [Sheeran 83], which is an extension of the programming language FP developed by Backus. Sheeran introduces the μ operator into the basic language FP to model memory in circuits. Both the behaviour (functional part) and the implementation (geometric part) of a circuit can be represented in this language. With the aid of the μ operator, the inputs and outputs are modeled as streams. Behavioural descriptions of circuits can be transformed into their geometric forms thus leading to correctness by construction. Only synchronous systems can be modeled by this formalism. Examples done using this framework include the design for a systolic correlator [Sheeran 83].

The above works have concentrated at the gate level and higher. Winskel in [Winskel 88] describes a compositional model for the more primitive components of a VLSI technology, namely transistors, capacitors, etc.. This work is based on the simulation models developed by Bryant [Bryant 81]. Though this work is at a more detailed level than what has been described so far, it is not clear how it can be related to the system level.

1.3 Motivation

From the above summary of work in the formal methods area, it is clear that effort has concentrated on getting concise models of devices (transistors, gates, circuits, etc.); but little effort is placed in formalising the rules of thumb used by designers of integrated circuits. These rules are generally obtained with experience, but in most cases their justifications are rooted in the detailed models of the lower level circuit devices. These rules of thumb are much simpler and more abstract than the formal reasoning behind what is and what is not a correct thing to do to get a correctly functioning circuit. The research work presented in this thesis addresses this problem.

1.4 Research Summary

There are a number of topics covered in this thesis in addressing the question of formalising an integrated circuit design style. These can be separated into the following areas:

- Develop simple models for the primitive components of a VLSI technology such as CMOS.
- Based on these simple models show the correctness of the rules of an integrated circuit design style.
- Demonstrate the viability of these ideas on a major case study.

1.4.1 Outline

If the activities of an integrated circuit designer are examined, we find that rather than keeping track of all the details, he uses simple rules of thumb which have

been refined with experience. These rules of thumb are guidelines for deciding which building blocks to use and how they are to be connected. Usually the more systematic the design style, the more structured these design rules are.

To show that these rules have a logical foundation, a design style known as NORA [Goncalves 83] has been analysed. In this design style there are two clock lines, inverses of each other, and two sorts of gates; n-type and p-type. The design rules govern how gates may be connected and which clock lines may drive them if the result on the output is to be guaranteed. To make the design style synchronous, a C^2MOS latch is used as a dynamic register. This further complicates the rules, but gives rise to a design style that generates smaller and faster circuits than standard CMOS.

The NORA design style has been demonstrated to fail for large circuits [Orton 84], but a refinement of this design style using a two phase non-overlapping clocking scheme solves this problem. This design style is known as CLIC, and was developed at Racal Research by Orton and his team [Orton 84]. This has a more complex set of rules than its predecessor due to the extra pair of clock lines. The research work presented here gives a formal foundation for the design rules of the CLIC design style. This gives a higher degree of confidence in the correct functionality of devices designed using this design style.

1.4.2 Simple Models for VLSI Primitives

The work presented here in developing simple models of primitive devices has mainly concentrated on the CMOS technology. In particular, it has been guided by the needs for a simple and tractable model for the verification of the CLIC design style. The model originally used was based on Bryant's simulation work [Bryant 81], where three signal strengths were used to allow modelling of charge decay on capacitors. This results in a lattice of seven signals. This was further simplified by modelling the decay of charge on the capacitors in a different way resulting in the four valued model of Hi, Lo, Er and Zz as used at present.

The model used for the transistor is also simple. A simple uni-directional model was found to be adequate for modelling the CLIC design style. This simple model greatly reduces the complexity of the correctness proofs of CLIC gates. The model of hardware used at present and as presented in this thesis is a simple non-delay, uni-directional model of transistors, together with a model of capacitors with charge decay. More complex models could have been used, but they complicate the proof considerably. Also, since the CLIC design style uses transistors in a strictly

defined way, the simple model of hardware as used here could be derivable from other more complex models.

1.4.3 Formalising the CLIC Design Style

The formalisation of a design style [Dhingra 88] entails giving a formal foundation to any rules of thumb that are used. This means that all correctness statements of CLIC gates are derived from a small number of primitive device definitions, rather than merely being stated as axioms. These definitions cover devices such as transistors and capacitors, and the necessary axiomatisation of the four valued algebra for signals. Instead of rules of thumb, we now rely on being able to show the correctness of complex structures by logical inference, based on the correctness statement for the various CLIC gates.

For example, in order that a CLIC gate may drive another, it is important that the input constraints of one be satisfied by the output constraints of the other. In formalising the design rules, these constraints are captured as predicates and incorporated in the correctness statement of the CLIC gate. So when two gates are to be connected, it is only necessary to check that the predicates match. This checking is done using the HOL system [Gordon 85b].

1.4.4 A New Digital Phase-Locked Loop Design

As a major case study to test some of the specification and verification techniques, the design of a Digital Phase-Locked Loop (DPLL) system was undertaken. The function of this DPLL is to re-generate the clock signal from the serial data being received. The basic principle is to measure the phase difference between incoming data transitions and the sampling edge of the clock signal. When on average this phase difference becomes too small, or too large, then the clock period is appropriately adjusted.

Fully digital phase-locked loop designs exist using binary quantised phase detectors and discrete phase adjustments, e.g. [Yamamoto 78] and [Yukawa 73]. The characteristics of such systems vary, depending on the techniques used for averaging the phase difference; but they all use binary quantised digital phase detectors. The design presented in this thesis uses a new kind of a Digital Phase Detector (DPD). The heart of this DPD is a digital window comparator. The proof of correctness of this and other parts of the system are outlined in this thesis.

This new DPLL system has been designed down to the CLIC gate level and has been described and simulated using ELLA [Morison 84]. Some of the subcomponents have been formally verified down to the detailed circuit level while others have merely been specified without formal proofs of correctness. In this thesis a discussion is presented on constructing the correctness statement and the proof of correctness for such real time systems.

1.5 Hardware Verification using Higher-Order Logic

The formalism used in this thesis is that of typed higher-order logic [Church 40]. To make this thesis self-contained, a brief overview is given in this section of the logical notation that will be used, followed by a review of the techniques for specifying and verifying hardware using higher-order logic. The material presented in this section is not new, and has already been covered in considerably more detail in the literature. For a more complete description of this logic and its machine formulation see [Gordon 85b], and for a more thorough introduction to the general techniques for specifying and verifying hardware using higher order logic see [Gordon 86, Camilleri 87, Gordon 88, Melham 88a]. In particular, Melham's paper [Melham 88a] is an excellent source; most of what follows in the remainder of this section is taken, with permission, from this paper.

1.5.1 Logical Notation

This logic uses standard predicate calculus notation. So for example the term "$\mathsf{P}(x)$" is interpreted as saying that x has property P, and the term "$\mathsf{R}(x, y)$" means that the relation R holds between x and y. It has the usual logical operators $\sim$, $\wedge$, $\vee$, $\supset$ and $=$ denoting negation, conjunction, disjunction, implication and equivalence respectively. Also provided are the two quantifiers $\forall$ and $\exists$ which express the concepts of *all* and *some*, e.g. "$\forall x.\ \mathsf{P}(x)$" means that the property P holds for every value of x, and "$\exists x.\ \mathsf{P}(x)$" means that the property P holds for some value of x. As some syntactic sugar to the logic, nested quantifiers of the form "$\exists x_1.\ \exists x_2.\ \ldots\ \exists x_n.$ tm" can be written "$\exists x_1\ x_2\ \ldots\ x_n.$ tm." Finally conditionals of the form "If b Then t_1 Else t_2" are expressed as "$(b \Rightarrow t_1 \mid t_2)$."

What makes this logic higher-order is that quantification is allowed over functions and predicates. Furthermore functions can take functions as arguments and

return functions as results. So for example the induction axiom for natural numbers can be expressed as the following theorem where the variable P ranges over predicates.

$$\vdash\ \forall \mathsf{P}.\ \ \mathsf{P}(0) \wedge (\forall n.\ \mathsf{P}(n) \supset \mathsf{P}(n{+}1)) \ \supset\ \forall n.\ \mathsf{P}(n)$$

Note that a theorem is denoted here by the turnstile symbol ($\vdash$) at start of the equation. This is to indicate that this is a derived fact in the system. Additional facts can be added to the basic system by either stating them as axioms or as definitions. Definitions are a conservative means of extending the basic system, i.e. they only add an additional constant to the system as a means for abbreviation. No additional facts can be derived with the addition of this definition that could not be derived before. An axiom however could lead to inconsistencies being introduced into the system if care is not taken. These are not used in the work that follows. As an example of a definition, consider the function Rise. This is defined in the logic as follows:

$$\mathsf{Rise}\ sig\ t \quad =_{def} \quad \sim sig(t) \wedge sig(t{+}1)$$

This predicate captures the notion that the signal *sig* rises at time t. The variable *sig* is a function from natural numbers to booleans, and is an example of a higher order variable being passed as an argument to a higher order function. The predicate Rise upon taking its first argument (*sig*) returns a function from numbers to booleans. This function when given an additional argument time (t) returns a boolean answer indicating whether the signal *sig* rises at time t or not.

In expressing complex terms in the logic, use is often made of an additional piece of syntax to simplify the readability of terms. This involves making abbreviations local to the term by making use of the "**let** ... **in** ..." statement. This is really syntactic sugar for a term stated in λ-calculus. The transformation of such terms into equivalent terms not involving this syntax is as follows:

$$\begin{pmatrix} \textbf{let}\ x = a \\ \textbf{in} \\ f(x) \end{pmatrix} \quad \equiv \quad (\lambda x.\ f(x))\, a \quad \equiv \quad f(a)$$

One more primitive constant that needs to be described is the ε-operator. Terms of the form "$\varepsilon x.$ tm[x]" denote the value v chosen such that "tm[v]" is

true. For example, the term "$\varepsilon x.\ x = 7$" exactly denotes the number 7, and the term "$\varepsilon x.\ x^2 + 3x + 2 = 0$" denotes one of two possible values, namely -1 or -2. However if this ε-operator is used over a term which is false for all values then an arbitrary but a fixed value is returned. For example, the term "$\varepsilon x.\ x < x$" denotes an arbitrary natural number. For a more in depth discussion of the ε-operator see [Leisenring 69].

1.5.2 Types in Higher-Order Logic

Every correctly formed term in higher-order logic has a type. The type of each term must be consistent with the type of its subterms. Informally, types can be thought of as sets and terms of that type can be thought of as elements of that set. The basic HOL system has the type of natural numbers(: *num*) and the type of boolean truth values(: *bool*) built in. Types can be built from other types by using type operators. So for example the type of signals on a wire represented as functions from natural numbers to booleans can be denoted "$: num \rightarrow bool$," where "$\rightarrow$" is an example of an infixed type operator.

Writing "tm : *ty*" explicitly states that the term tm has type *ty*. By using this notation the type of the predicate Rise defined above can be stated as follows:

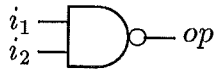
$$\mathsf{Rise} : (num \rightarrow bool) \rightarrow (num \rightarrow bool)$$

Such explicit type information is usually not given. In practice it is only necessary to give such detailed type information when it is not clear from the context what the type of the term should be.

New types can also be declared in the HOL system. This is a fairly tedious task. A package which allows new types to be declared automatically in the HOL system has recently been developed by Melham [Melham 88b]. The actual procedure for declaring new types in the HOL system is not described here, but for more information on this see [Gordon 85b,Melham 88b].

1.5.3 Specifying the Behaviour of Hardware Devices

The behaviour of hardware can be captured in higher-order logic by predicates. The labels corresponding to the external ports of a device are passed as arguments to the predicate. The predicate is defined as a relation indicating which combinations of values can appear on these ports. As an example, consider the definition for the behaviour of a two input nand gate as shown below:



The behaviour of this device can be captured in logic by a predicate Nand_2 with three arguments corresponding to the three external ports. This predicate is then defined to be true for all combinations of values that can occur on these variables which correspond exactly to the ports of a hardware nand device. The definition for this device can be formally stated as follows:

$$\mathsf{Nand}_2(i_1, i_2, op) \quad =_{def} \quad (op = \sim(i_1 \wedge i_2))$$

As another example, the behaviour of the multiplexor device as used earlier can be expressed formally in the same way. The top level view of this device is shown below in figure 1.2. The external ports are labeled here in exactly the same way

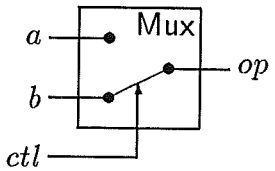


Figure 1.2: Top level view of a multiplexor device

they were labeled in figure 1.1 on page 4. The behaviour of this device is captured as a predicate Mux with four arguments corresponding to the four external ports. This is defined in logic as follows:

$$\mathsf{Mux}(ctl, a, b, op) \quad =_{def} \quad (op = (ctl \;\Rightarrow\; a \mid b))$$

1.5.4 Specifying the Structure of Hardware Devices

The structure of hardware devices is captured in higher-order logic by conjoining together the predicates for each of the subcomponents. In this way the constraints imposed by each of the subcomponents are pooled. These constraints together constitute the behaviour of the top level device. The interconnect between the parts is captured in logic by commonly named lines. So, for example, the structure of the multiplexor device shown earlier in figure 1.1 can be formalised as follows:

$$\mathsf{Nand}_2(ctl, ctl, x) \wedge \mathsf{Nand}_2(a, ctl, y) \wedge \mathsf{Nand}_2(b, x, z) \wedge \mathsf{Nand}_2(x, y, op)$$

The internal lines of this circuit are hidden from the environment, but in the above equation this "hiding" is not captured. Hiding of lines is done in logic by existentially quantifying them [Camilleri 87]. Now a new predicate MuxImp can be defined which captures the structure of the multiplexor circuit complete with the information that the lines x, y and z are hidden (or internal) as follows:

$$\begin{aligned}\mathsf{MuxImp}(ctl, a, b, op) \quad =_{def} \quad & \exists x\ y\ z. \\ & \mathsf{Nand}_2(ctl, ctl, x) \wedge \\ & \mathsf{Nand}_2(a, ctl, y) \quad \wedge \\ & \mathsf{Nand}_2(b, x, z) \quad \wedge \\ & \mathsf{Nand}_2(x, y, op)\end{aligned}$$

This definition states that the values that can appear on the external ports are precisely those which satisfy the constraints imposed by the four predicates, where these four predicates model the four nand gates used in the implementation of this multiplexor device.

1.5.5 Deriving the Correctness Statement

The two predicates Mux and MuxImp as defined above, capture the specification and the implementation of a simple multiplexor device in higher-order logic. Now by simple logical manipulation in the logic, the following theorem can be derived:

$$\vdash \mathsf{MuxImp}(ctl, a, b, op) \quad = \quad \mathsf{Mux}(ctl, a, b, op)$$

This is the correctness statement for the multiplexor device. It states that the values that can appear on the external ports of the implementation are exactly those that are allowed by the specification, i.e. the implementation meets the specification.

In this example, the implementation predicate MuxImp is proved to be equivalent to the specification predicate Mux. For more complex devices, formulating the correctness statement as an equivalence relation may not be appropriate. The behavioural specification of large and complex systems could be different in many respects as compared to the implementation which implements them. For example, specification may be partial, or be stated at different granularities of time, or be stated using abstract data types. In such cases the correctness statement will generally be in the form:

$$
\begin{array}{l}
\vdash \mathsf{Dev\_Imp}(i_1, \ldots i_n, o_1, \ldots o_n) \supset \\
\quad \textbf{let } ip \ = \ \mathsf{Abs}_i(i_1, \ldots i_n) \\
\quad \textbf{in} \\
\quad \textbf{let } op \ = \ \mathsf{Abs}_o(o_1, \ldots o_n) \\
\quad \textbf{in} \\
\quad \mathsf{Dev\_Spec}(ip, op)
\end{array}
$$

where the abstraction functions Abs_i and Abs_o map the input and the output signals respectively of the implementation Dev_Imp, to those of the specification Dev_Spec.

1.6 Thesis Outline

Chapter 2: In this chapter a brief introduction is given to the various design techniques. In particular, the term "hardware design style" is clearly defined. Then an extended summary is given of two such design styles known as DOMINO and NORA, together with a discussion on their advantages and problems.

Chapter 3: In this chapter the CLIC design style is presented. This design style overcomes some of the problems associated with NORA. First a detailed but informal description of this design style is given. Then a full account is given of the formalisation of this design style in HOL, beginning with simple models of the primitives such as transistors and capacitors. The last part of this chapter is devoted to simple examples illustrating how this work can be used to derive the correctness statements for circuits designed using this formal framework.

The next two chapters give proof outlines of two components, with varying amounts of detail. Both of these components are used in the design of the digital phase locked loop.

Chapter 4: In this chapter a formal proof of correctness of the Random Walk Filter (RWF) is presented. The RWF is designed in the CLIC design style, and the correctness statement for it is derived using the formal techniques of chapter 3. Integers are used in the specification of this device since they help represent the functionality in a more natural way. This does however complicate the proof considerably at the higher level. This chapter focuses on deriving the correctness statement from a cluster of CLIC gates rather than dealing with the proof details at the higher levels.

Chapter 5: In this chapter a plan for the proof of correctness of the Window Comparator is presented. The proof uses integers and modular arithmetic, but it does not go all the way down to the transistor level as in the previous chapter. However the primitive devices used in constructing this device can be trivially proved down to the transistor level. An informal specification is first given which is then improved as the proof develops.

Chapter 6: This chapter first gives a brief overview of control systems with particular emphasis on Digital Phase-Locked Loops (DPLLs). Then a novel design for a new class of phase-locked loops is presented which uses the devices of the previous two chapters. A formal specification for this device is presented together with a sketch for an informal proof of correctness. Some of the difficulties involved in arriving at the correctness statement and constructing the proof are discussed.

Chapter 7: Conclusions, discussion and future work.

Chapter 2

Hardware Design Styles

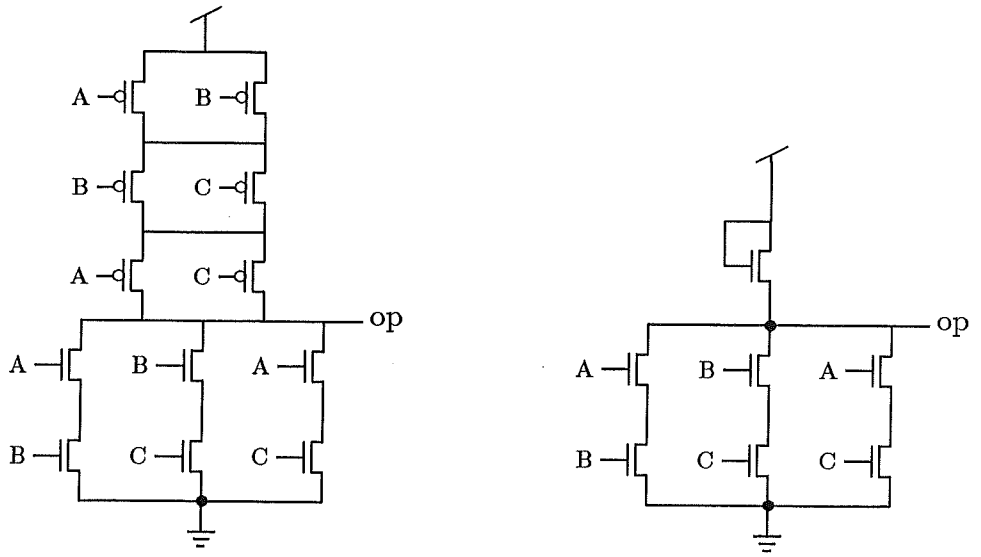
In this chapter a brief introduction is given to various design techniques. In particular the term "hardware design style" is clearly defined. An extended summary is given of two such design styles known as DOMINO *and* NORA*, together with a discussion of their advantages and problems.*

2.1 Introduction

Circuits built using fully complementary CMOS techniques have an inherent redundancy of information. Consider, for example, the circuit for the carry out stage of a full adder, as illustrated in figure 2.1a. The six n-channel transistors contain all the information needed to implement the logic function of this gate as do the six p-channel transistors. The advantage of duplicating the functionality is that there is virtually no power consumed by the gate, except for the short periods of time when the inputs or the outputs are making transitions.

The problem with this approach is that for complex gates, such as the one in figure 2.1a, a considerable amount of silicon area is wasted by duplicating the functionality of the gate. Also, the capacitive loads of such devices are fairly high, since the output of these gates have to drive both the n-channel and the p-channel devices. To obtain the same functionality in NMOS would require only the n-channel devices together with a load transistor as shown in figure 2.1b.

The other major problem with fully complementary CMOS technique has to do with clock races. As an example consider the design of a simple master-slave D-flip-flop. Illustrated in figure 2.2 is the D-flip-flop, which is amongst the most commonly-used elements in VLSI. Note that transmission gates are used to clock



(a) CMOS (b) NMOS

Figure 2.1: The carry out circuit of a full adder

data in. These are generally implemented using a p-channel and an n-channel transistor in parallel. Single transistors are avoided due to low noise margins. For data to flow through the pn-transmission gate, two clocks, ϕ and $\overline{\phi}$, are required. For any reasonably sized circuit, the clock will be degraded due to distribution and loadings, and this will lead to clock skew and the potential for some overlapping of the two phases. During the phase overlaps, several successive transmission gates may be switched on, which would lead to illegal data flow depending on the ratio between gate delay and clock skew. This has become a serious problem with the reduction in gate delays due to improvements in technology, and has resulted in designers paying considerable attention to clock distribution and load.

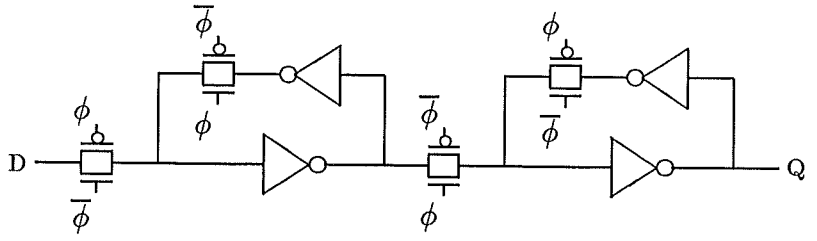


Figure 2.2: Master slave D-flip-flop designed in fully complementary CMOS

Because of these two drawbacks of fully complementary CMOS, designers have developed techniques which allow one to use stripped down versions of logic gates provided that certain rules are followed. It is these rules, either self imposed or motivated by the way the hardware is being used, which constitute "Hardware

Design Styles". In the remainder of this chapter a brief overview is given of two design styles which are the predecessors to the design style used as the subject of this thesis. But first a brief summary of *Synchronous Circuits, Clocked CMOS Circuits* (C^2MOS), and *Dynamic Circuits* is presented, which are the generic techniques used in many design styles. All of these techniques use some form of a clocking scheme which is an important part of systems design. Weste summarises the use of clocking in systems design as follows:

> Clocks are used in digital systems to hold up a signal until it is time for it to begin to move through the next stages of logic. Registers are used in conjunction with the clocks so that a signal can be stored at a location until it is needed.
>
> [Weste 85, page 332]

2.2 Synchronous Circuits

The basic principle behind any synchronous design philosophy is that the system is separated into blocks of purely combinational logic with no data storage facility, interleaved by register latches which hold the data between clock pulses. There is a global system clock which is used to clock the register latches, the period of which is such as to allow all combinational logic blocks to finish evaluation of results. So, on the tick of the clock, new data appears on the inputs of the combinational logic blocks, and the old results are passed as inputs to the next stage by use of the register latches. By definition there is no feedback within the purely combinational logic blocks. This principle is illustrated in figure 2.3.

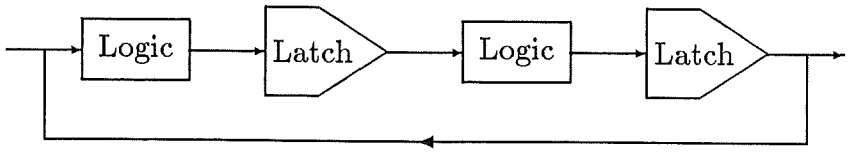
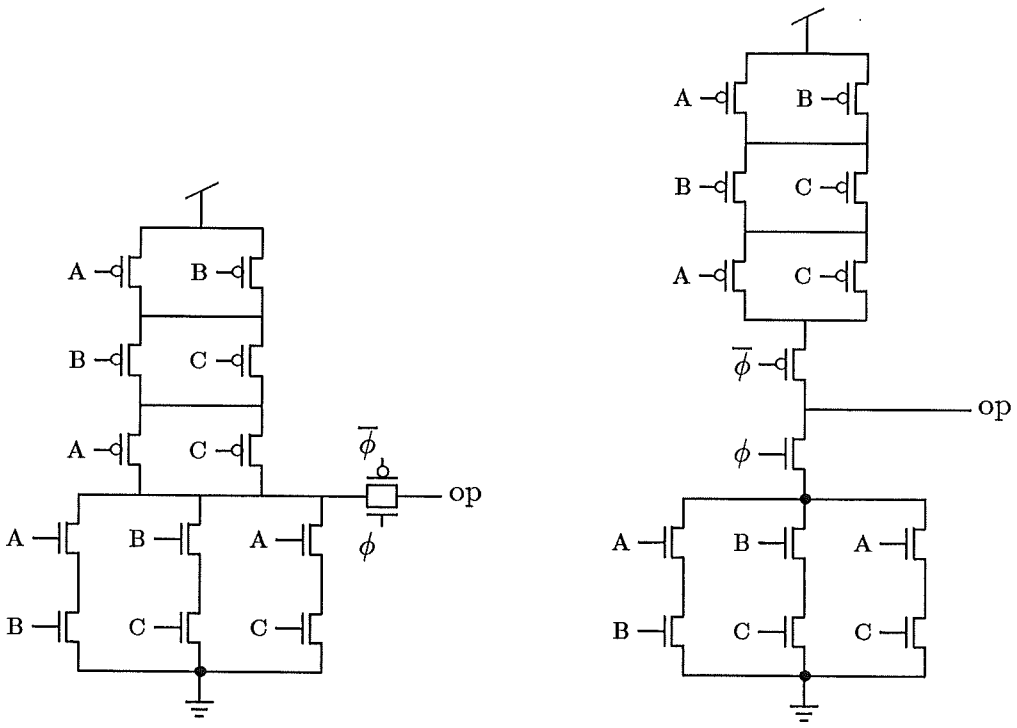


Figure 2.3: Synchronous Logic Concept

The advantage of such a design technique is that all the timing problems are localised to the areas between the register latches. Thus the problem of identifying difficult timing paths and race hazards is considerably simplified. However, it does require care to ensure that the latency through the various paths are matched. Furthermore, in fully complementary CMOS, the register latches used are essentially variations of the basic D-flip-flop as discussed above. So problems due to clock skew and phase overlap are still present.

2.3 Clocked CMOS Circuits (C^2MOS)

A C^2MOS gate is essentially the net result of combining a fully complementary CMOS gate with its output passed through a clocked transmission gate. An example of this is illustrated in figure 2.4 using the adder circuit as described earlier with the output passed through a clocked transmission gate.



(a) CMOS gate and a transmission gate (b) C^2MOS gate

Figure 2.4: An example of a C^2MOS gate

This technique was originally developed to build low power dissipation CMOS circuits for calculators [Suzuki 73]. The reason for the low dynamic power were to do with the layout considerations when using the metal gate CMOS technology. Though the technology has changed, the technique is still used where it is necessary to have fully complementary CMOS gate followed by a clocked transmission gate.

C^2MOS gates have the same input capacitance as regular fully complementary CMOS gates, and larger rise and fall times on the outputs due to the extra clocking transistors in series to the path to the power rails. However these clocked gates are slightly faster than their equivalent circuit composed of a regular fully complementary CMOS gate followed by a clocked transmission gate.

2.4 Dynamic Circuits

Dynamic logic gates have two phases of operation: the precharge phase and the evaluation phase. During the precharge phase the output nodes of the gates are precharged to a particular level, usually high for n-type gates. During this period the path to the other level, ground for n-type gates, is turned off. The changing of the inputs of the gate must also occur during this period. This is necessary because otherwise charge redistribution effects could corrupt the output node voltage. Then by using a system clock the gate is switched from the precharge phase to the evaluation phase. For n-type gates this involves turning off the path to the high level and turning on the path to ground. Depending on the state of the inputs, the output will either remain floating high or go low. Figure 2.5 illustrates this principle on the carry out circuit of the full adder described earlier.

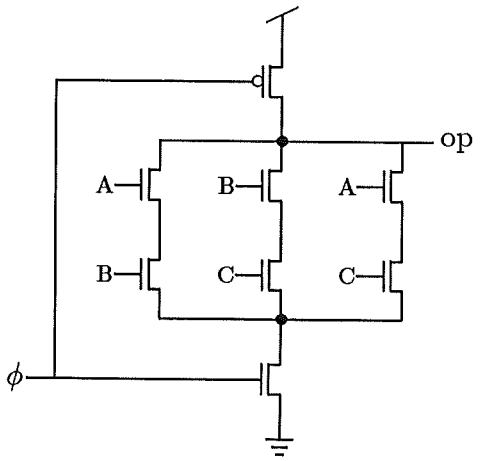


Figure 2.5: N-type dynamic gate for the carry out stage of a full adder

The advantages of dynamic circuits are that far fewer transistors are used (approximately half), which means that the silicon area used per gate is reduced. Another implication is that the the load capacitance on the output of such gates is lower than in fully complementary CMOS, since they only have to drive either n-channel or p-channel transistors in the next gate. Further, the power consumed by such circuits is lower than with NMOS since there is no pull-up transistor. The output is precharged before evaluation, and during the evaluation only one path to the power rails is open, so there is no static current path consuming power.

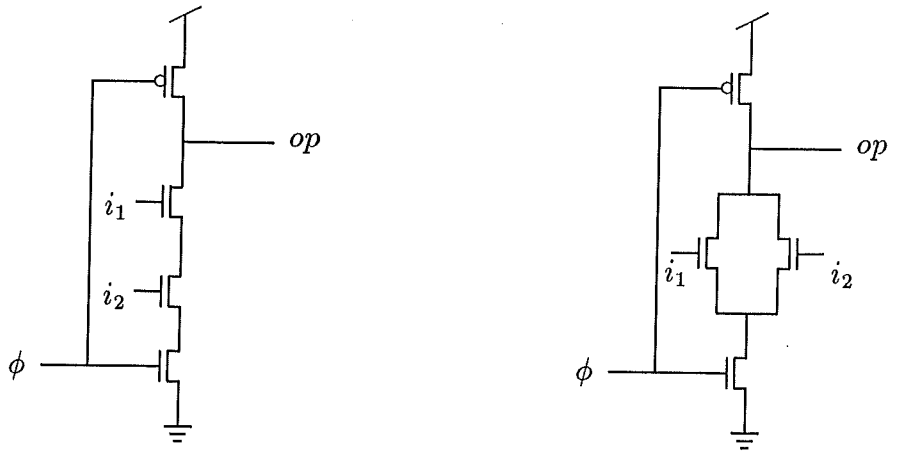
Since there is no static current path and the load capacitance on the output is comparable to that of NMOS circuits, it would appear that dynamic circuits have the advantages of fully complementary CMOS, namely low power consumption, and those of NMOS, namely high speed operation. But in real circuits there is still some power penalty as compared to fully complementary CMOS, because each gate must

be precharged on every cycle even if its output is to continue at the other level. It is also difficult to realise these apparent speed advantages in real circuits since most useful circuits generally have several logic gates in series. In dynamic circuits this is a serious problem since dynamic CMOS gates cannot be cascaded by using a simple single phase clocking scheme. In order to cascade dynamic CMOS gates, designers have to resort to complex clocking strategies which can involve anything up to eight clock lines. One such technique forms the basis of the next two sections. For a more detailed survey of some of the other techniques see Weste's book on "CMOS VLSI Design" [Weste85, pages 203–224].

One last point is that the clock in dynamic circuits cannot be stopped. There is a minimum speed at which the clock for such circuits could be operated, but they cannot be single stepped (which could be useful for debugging purposes, be it for problems due to technology or logic design).

2.5 The DOMINO Logic Design Style

The previous section mentioned that dynamic gates cannot be cascaded by use of a single phase clock. Let us examine this in a little more detail. Consider some simple examples, namely an n-type Nand and an n-type Nor gate as illustrated in figure 2.6. The previous section also dictates that the inputs must only change during the precharge period, i.e. during the period when ϕ is low.



(a) n-type dynamic Nand Gate (b) n-type dynamic Nor Gate

Figure 2.6: N-type dynamic Nand and an n-type dynamic Nor gate

One solution would be to have only one dynamic gate at the start of a chain of gates with the remaining gates in the chain being static gates. This really doesn't

buy us much, and a great deal better can be done. The real insight is to develop a set of building blocks where a single transition on the input lines can result in no more than a single transition on the output. Then these blocks could be combined to make larger blocks provided no cyclic structures are used.

Going back to the Nand and the Nor gates of figure 2.6, note that if the inputs are precharged low and the outputs are precharged high, then, when these gates go from the precharge phase to the evaluation phase, then they in fact obey the above rules. Provided the inputs make no more than one transition (from low to high, ⌐) then the output will also make no more than one transition (from high to low, ¬). This argument is true of both the Nand and the Nor gates of figure 2.6. In fact, this is true of any n-type complex gate provided the net of transistors used to compute the logical function of the gate are composed of parallel/serial combinations of n-type enhancement mode transistors only.

Now if the output of these gates are followed by a static inverter then the output of the inverter could make no more than one transition, from low to high (⌐). This in fact is exactly what is required on the inputs of the next dynamic gate. So collectively this now forms a design style where the output of each dynamic gate is buffered by a static inverter resulting in a set of building blocks consisting of And, Or, and complex noninverting gates. This design style is commonly known as the DOMINO logic design style [Krambeck 82], so called because the chain of evaluation goes sequentially from the start of the chain to the end, much as the fall of one domino causes the next to fall which in turn causes the next to fall and so on. A simple example illustrating this DOMINO style of circuit design is shown in figure 2.7 below.

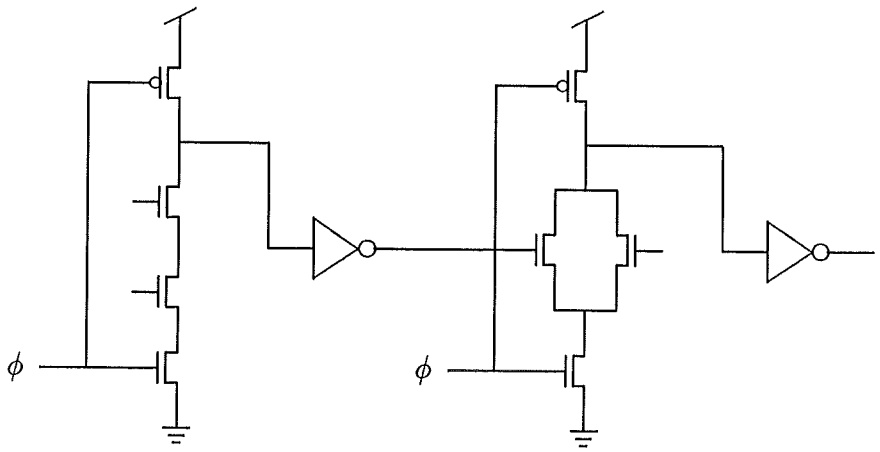


Figure 2.7: An example DOMINO circuit

2.6 The NORA Logic Design Style

By duality to the DOMINO logic design style, a family of p-type dynamic gates can be developed which have almost an identical set of rules for cascading the logic blocks. The difference is simply that the inputs of the p-type gates are precharged high and the outputs are precharged low. As before, provided the inputs make no more than a single transition (in this case from high to low, ↘), then the output will also make no more than a single transition (from low to high, ↗). Again, by buffering each dynamic gate by a static inverter, a design style consisting of p-type, And, Or and complex noninverting gates results.

However a more universal design style can be developed if both n-type and p-type gates are mixed. Note that the behaviour of the output of an n-type dynamic gate is identical to the requirements for the inputs of a p-type dynamic gate; in both cases the nodes are required to be precharged high, and during the evaluation period there is no more than a single high to low (↘) transition on these nodes. Similarly the behaviour of the output of a p-type dynamic gate is identical to the requirements for the inputs of an n-type dynamic gate; this time the nodes are required to be precharged low, and during the evaluation period there is no more than a single low to high (↗) transition on these nodes.

So by ensuring that the inputs of all n-type dynamic gates are driven by p-type dynamic gates and vice versa, a more general design style evolves known as the n-p-CMOS design style [Goncalves 82]. This time the building blocks consist of the dynamic n-type and the dynamic p-type gates without the buffering static inverters. This means that the primitive building blocks are in fact a set of inverting dynamic gates which forms a more universal logic family. Not only this, but the style of DOMINO logic design can also be used within the framework of this new design style. Naturally this new design style has a more complex set of logic design rules than the DOMINO style, but the extra generality of inverting rather than noninverting dynamic logic gates is worth this price.

There is however still another refinement that can be made to this n-p-CMOS design style. Consider what happens if there is a very long chain of gates. As soon as the circuit goes into the evaluation phase, evaluation begins at the start of the chain and ripples down to the end. Depending on the length of the chain, the amount of time needed before the last gate in the chain finishes evaluation can be fairly large. During this period, most of the gates are idle; they are either waiting for the previous gate to finish evaluation, or have finished evaluation and so are waiting to be precharged in preparation for the next evaluation phase. The only gates which are active during this time are those which are at the waveform

of evaluation propagating down the chain. The only way to get more throughput from such circuits is to have more than one wavefront of evaluation running down the chain.

Since a gate cannot be made to go into another evaluation phase without having gone through a precharge phase first, the only solution is to divide the chain into smaller subchains, and have alternate subchains go into precharge and evaluate periods at different times. This would mean that once a subchain has finished evaluation, the results are passed onto the next subchain which is just coming out of the precharge period. Now the first subchain can go into precharge in preparation for the next set of data to be evaluated. So, this alternation of precharge and evaluation can be used to have multiple wavefront of evaluation running down the resulting large chain. The control mechanism is the obvious problem which is explained next.

In order to pass the results of one subchain to the next while the first goes through the precharge phase, the results have to be stored at a node between the two subchains. This is achieved by using a simple C^2MOS latch driven by the same pair of clock lines, ϕ and $\overline{\phi}$, as used for the dynamic gates in the subchains. This latch is clocked in such a manner as to latch onto the evaluated results of the subchain just as it goes into the precharge phase, and hold this result static on the output until the next evaluation period. This resulting subcircuit, containing a subchain and a latch to hold the result while the subchain is in precharge, will be referred to as a *section*. If all consecutive sections are clocked by opposite phases of the clock then the behaviour of the resulting network is such as to allow multiple wavefronts of evaluation to propagate through the circuit at the same time as outlined above.

So from the point of view of the designer the circuit can be viewed as one which is divided into small sections, each terminated by a latch. All the clock lines are swapped between neighbouring sections so that if a p-type gate in one section is clocked by ϕ then the p-type gates in the following section will be clocked by $\overline{\phi}$. This has the effect of pipelining the computation along the row of sections. It means that when one section is in the evaluation phase then all the inputs to that section will be stable, since they all come from sections which are in the latched phase. And all sections which are in the latched phase will have all their internal dynamic gates being precharged during this period with their previously calculated results being held static on the output by the latch.

There are three distinct phases in the process of generating an answer on the output of a section. Consider a very simple section having only two dynamic gates

(an n-type and a p-type dynamic inverter) and a latch as shown in Figure 2.8. The answer on the output of this simple circuit is generated by tracing through the following three steps.

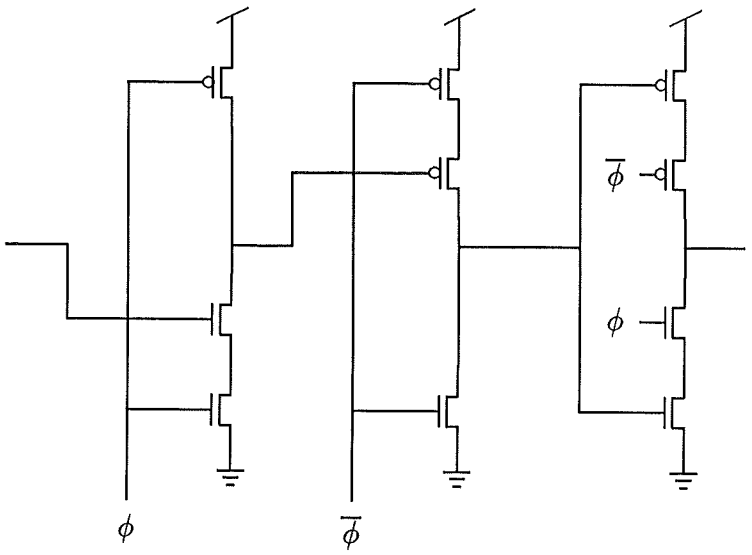


Figure 2.8: A simple NORA Circuit

Precharge Phase: ϕ = low, and $\overline{\phi}$ = high

- The outputs of all p-type and n-type gates in the section are precharged high and low respectively.
- The latch terminating the section is turned off such that the previous value on the output of the latch is maintained and any further changes on the input to the latch have no effect on the output node until the clock lines toggle.

Evaluation Phase: ϕ = high, and $\overline{\phi}$ = low

- The latch is turned on such that it becomes transparent, i.e. any changes on the input are reflected straight on the output.
- The evaluation begins from the input end of the section, bubbling the answer to the latch. Since the latch is now transparent the answer goes straight to the output of the section and is held there.

Latch and Precharge Phase: ϕ = low, and $\overline{\phi}$ = high

- The latch terminating the section is turned off such that the previous value on the output of the latch (which is the evaluated result of the section the latch terminates) is maintained, and any further changes on

the input to the latch have no effect on the output node until the clock lines toggle again.

- The outputs of all p-type and n-type gates in the section are again precharged in preparation for the next evaluation phase.

This resulting technique of mixing n-type and p-type dynamic gates together with dividing the circuit into smaller sections to allow multiple wavefronts of evaluation through the circuit is known as the NORA logic design style [Goncalves 83]. The advantages of this technique are that it provides more logic flexibility as compared to the DOMINO style, and the circuits generated are faster and more compact as compared to fully complementary CMOS.

2.6.1 Problems with NORA

Unfortunately this design technique does not scale too well. For large circuits the signals on the clock lines will deteriorate due to clock distribution and loadings and will result in slow clock rise and fall times. During the interval when the signals on the clock lines are neither high nor low but somewhere in between, the behaviour of the circuit can no longer be guaranteed to be correct. Consider what happens at the interface between the last dynamic gate in a section and the latch which terminates that section. When the signals on the clock lines are in the process of changing state, then, due to slow clock rise and fall times, the latch may still be in its transparent mode when the gate feeding it starts to go into its precharge mode. This will then result in the evaluated answer of that section being lost and the new precharged value on the output of the dynamic gate being stored on the output of the latch.

This is a recognised problem with the NORA design technique which requires that the rise and fall times on the clock lines be less than a single gate delay. To overcome such problems the solution suggested by Goncalves and de Man in [Goncalves 83] is to use clock buffers at regular intervals to help maintain relatively sharp edges on the clock lines. This is a possible solution which will require considerable planning for the distribution of the clock signals on a chip, but a more general approach to solving this would be to use a two phase non-overlapping clocking scheme which removes the root cause of this problem.

The use of a two phase non-overlapping clocking scheme gives rise to a new design style with its own set of rules for cascading logic blocks. This is the subject matter of the next chapter where it is presented in considerable detail.

2.7 Summary

In this chapter various integrated circuit design techniques have been reviewed with particular emphasis on the DOMINO logic design styles. It has been demonstrated how the basic DOMINO logic design style can be generalised by using both p-type and n-type logic gates. This, together with using a C^2MOS latch, gives rise to a design style known as NORA. Finally the failure mechanism for the NORA design style was briefly outlined. One solution, which forms the basis of the CLIC design style, uses a two phase non-overlapping clocking scheme. This is covered in considerably more detail in the next chapter.

Chapter 3

CLIC: CLock Insensitive Cmos[1]

In this chapter the CLIC *design style is presented. This design style overcomes some of the problems associated with* NORA. *First a detailed but informal description of this design style is given. Then a full account is given of the formalisation of this design style in* HOL, *beginning with simple models of the primitives such as transistors and capacitors. The last part of this chapter is devoted to simple examples illustrating how this work can be used to derive the correctness statements for circuits designed using this formal framework.*

3.1 Introduction

Increasing improvements in integrated circuit technology requires that the design techniques be continually appraised. Indeed the particular failure mode of the NORA integrated circuit design style as presented in the previous chapter, and also previously noted by others [Goncalves 83,Orton 84], is a direct consequence of the apparent speed-up of the CMOS integrated circuit technology. A solution proposed by Orton and his team at Racal Research requires that a two phase non-overlapping clocking scheme be used. This imposes a new set of rules for design, and results in a new design style known as CLIC [Orton 84].

[1]The CLIC design style was originally developed at Racal Research in 1984 [Orton 84]. The presentation given here is significantly different, especially the clock naming convention which is completely different. These differences are merely to help the formalisation of the design style; the fundamental concepts which constitute this design style are unchanged. For a more detailed electrical analysis of this see [Orton 84].

3.2 Informal Overview of the CLIC Design Style

There is much that is common between the NORA design style as presented in the previous chapter and the CLIC design style. Instead of using a simple square wave clock (ϕ) and its inverse ($\overline{\phi}$), the CLIC design style uses a two phase non-overlapping clocking scheme. The failure mode of the NORA design style is due to the fact that the same clock edges are used for both latching the results of a section and precharging it. By using the two phases of the clock to do these tasks separately, this problem is totally eliminated. In the CLIC design style, one phase of the clock and its inverse is used for precharge and evaluation, and the other phase and its inverse is used for latching the results. In this way there is a period of time when all the gates in a section will have finished evaluation and the latches will not have started latching. This will ensure that even if there are very poor edges on the clock lines, the results of the section will be latched correctly. Before looking into this in more detail, the signals on the clock lines are briefly described together with how they are generated.

3.2.1 Clock Description and Generation for CLIC

The two phase clock for the CLIC design style is generated from a single square wave clock input. On every rising edge of the external clock input an internal narrow pulse is generated on the ϕ_1 clock line. Similarly on the falling edge of the external clock another pulse is generated on the ϕ_2 clock line. These two internal clock lines are then inverted to form the remaining two internal clock lines, namely $\overline{\phi}_1$ and $\overline{\phi}_2$ respectively. These are shown in figure 3.1 together with the external clock.

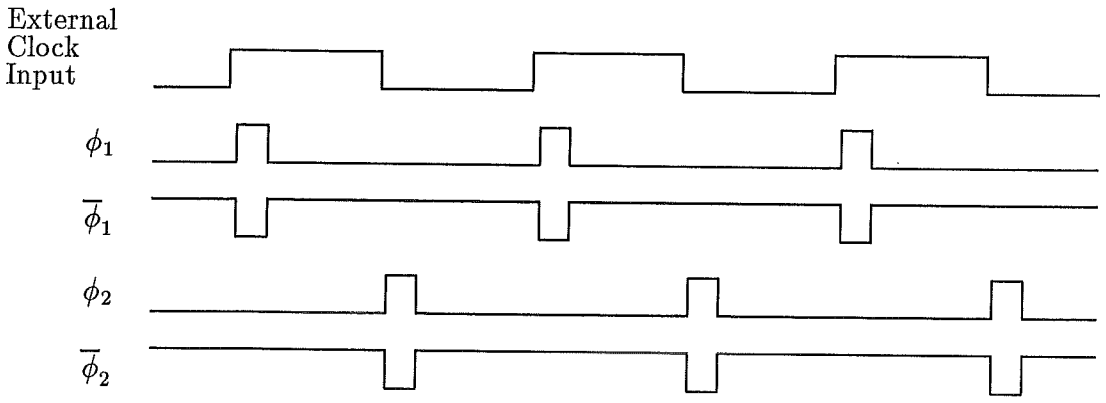


Figure 3.1: The external and the internal clock relationships

For the purposes of analysing the clocking scheme, the clock cycle can be divided into eight distinct intervals. This is shown in figure 3.2, where the shaded regions represent uncertainty in the value on the clock lines, i.e. the value could be Hi, Lo or something in between. The essential requirements of the clocking scheme are that the duration of the clock pulses, i.e. the intervals t_3 and t_7, should be long enough for the internal gates of the chip to have enough time to precharge their outputs. For a 3um CMOS technology this duration is in the region of about 10ns.

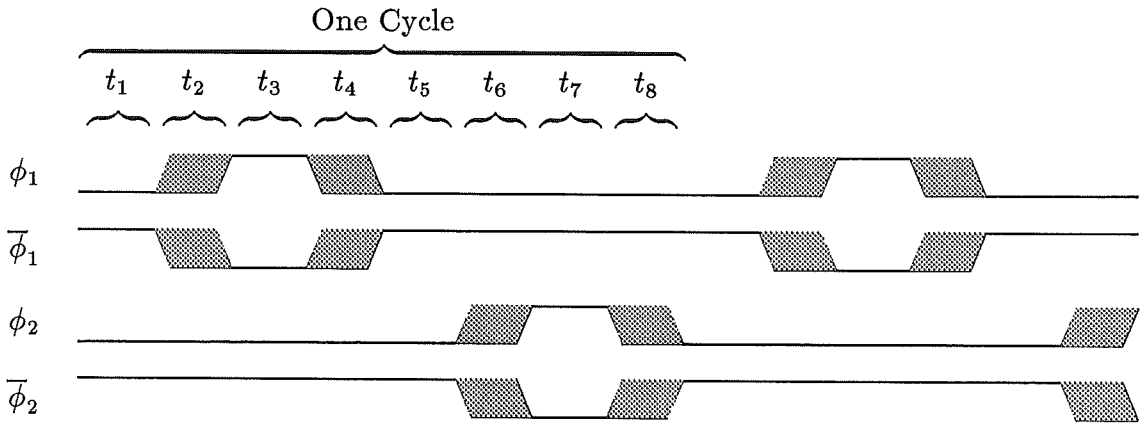
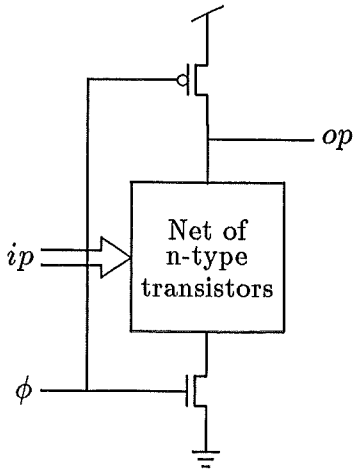


Figure 3.2: Schematic view of the internal clock lines

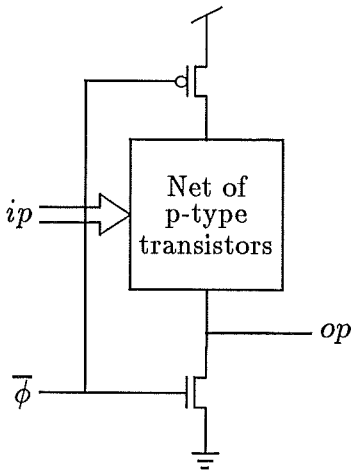
3.2.2 CLIC Primitive Gates

Except for the static inverter, all gates in the CLIC environment are dynamic logic gates and are driven by one or more of the four clock lines. The primitives used in realising a logic function are a combination of these gates. There are four basic building blocks used in CLIC circuits: the nShell, the pShell, the Latch, and the Stat_Inv. These are illustrated in figure 3.3. Both the nShell and the pShell devices need extra components, namely transistors, to be "wired" into them to make n-type and p-type gates respectively. It is these n-type and p-type gates which form the primitive gates as used in the CLIC design scheme.

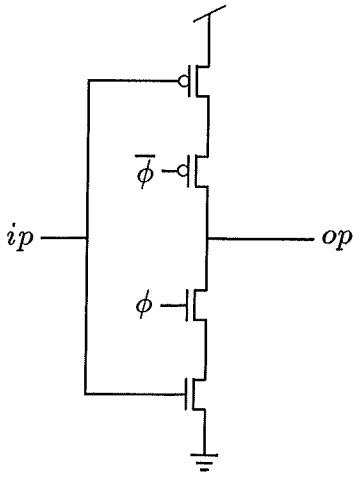
In the remainder of this section a brief account is given of how each of these devices works. The workings of the static inverter however are trivial and will not be discussed here.

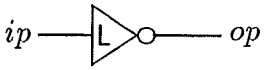


(a) nShell

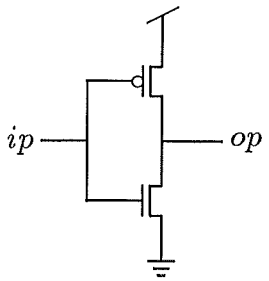


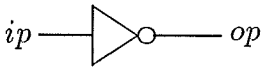
(b) nShell





(c) Latch





(d) Stat_Inv

Figure 3.3: The primitive building blocks of CLIC

3.2.2.1 The Latch

The latch as used in the CLIC environment is a dynamic device. It needs to be updated at regular intervals; otherwise the value held on its output may deteriorate. There are two types of latches in the CLIC system, one driven by ϕ_1 and $\overline{\phi}_1$, and the other driven by ϕ_2 and $\overline{\phi}_2$. The differences between the two are only in the clocks by which they are driven.

A typical latch is illustrated in figure 3.4 together with the clocks which drive it. The input is required to be stable during the times when there is a positive pulse on the ϕ line and a negative pulse on the $\overline{\phi}$ line. By stable it is meant that the input should either remain Hi or Lo, and should not be in the process of changing. This constraint on the input is necessary to stop the latch from locking onto an incorrect value.

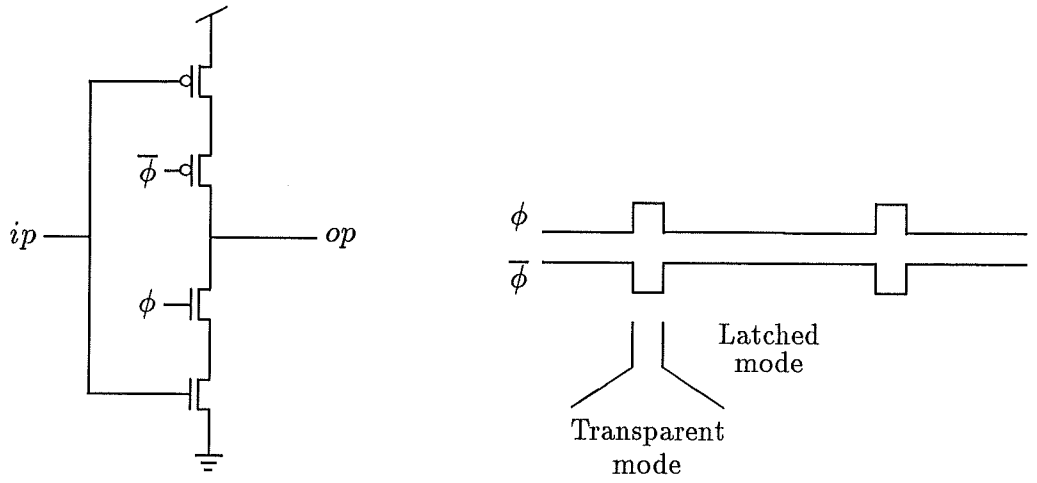


Figure 3.4: The Latch

If a latch is correctly clocked then it can be in one of two modes—Transparent mode or Latched mode.

Transparent mode ϕ = Hi, and $\overline{\phi}$ = Lo

The two transistors driven by the clock lines get switched on, so the value on the output of the latch is charged to the inverse of the input. It is also important to note that during this time the latch behaves as a static inverter, i.e. if there were any changes on the input then they would be reflected by a change on the output.

Latched mode ϕ = Lo, and $\overline{\phi}$ = Hi

The two transistors driven by the clock lines get switched off, so the output

node of the latch gets isolated from the power rails and so retains the previously charged value. During this time any changes on the input have no effect on the output. The output node is designed to hold the value long enough until the latch is refreshed again by going into the transparent mode for a short time.

The latch only needs to be in the transparent mode for a short time, long enough for the output node to be charged to the correct value. Even if there is considerable clock overlap and clock edges are poor, the latch will still lock onto the correct value provided the input is stable during the interval when there is a positive pulse on the ϕ line and a negative pulse on the $\overline{\phi}$ line. This restriction ensures that the latch behaves correctly as regards locking onto the input value.

3.2.2.2 N-type and P-type Logic Gates

The name of n-type and p-type logic gates arises from the fact that these gates use only n-type or p-type transistors respectively (except for the precharging and enabling transistors). All p-type gates are driven by either ϕ_1 or ϕ_2, and all n-type gates are driven by either $\overline{\phi}_1$ or $\overline{\phi}_2$. Perhaps the best way to understand the working of these gates is by example.

Consider a simple two input n-type nand gate as shown in figure 3.5. The working of this gate has two distinct phases—the Precharge period and the Evaluation period.

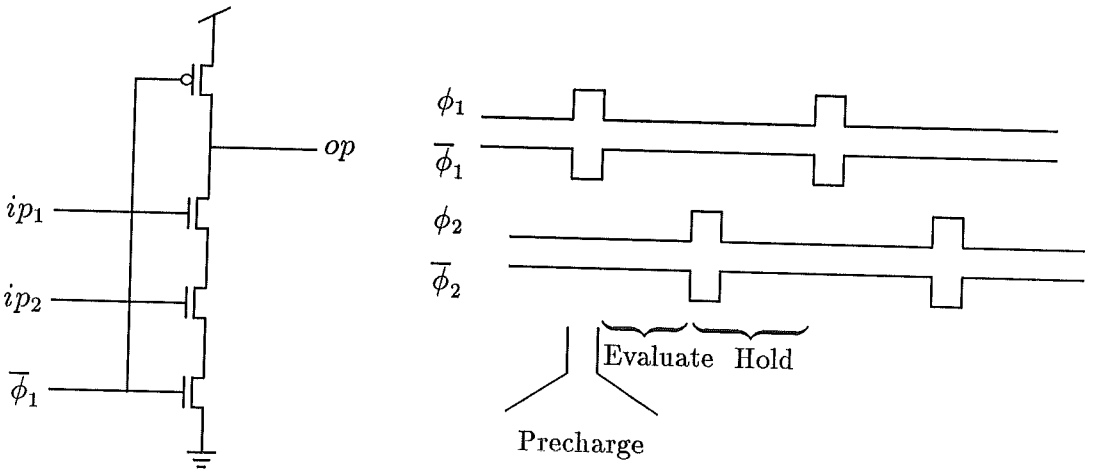


Figure 3.5: Two input n-type nand gate

Precharge $\overline{\phi}_1 = \mathsf{Lo}$

During this period the output of the gate is pulled Hi by the enabled p-transistor. Any changes on the inputs during this time have no effect on the output, since the bottom n-transistor is off and so the path to Gnd is effectively cut, i.e. the output cannot be pulled Lo.

Evaluation $\overline{\phi}_1 = \mathsf{Hi}$

When $\overline{\phi}_1$ goes Hi the top p-transistor goes off and the bottom n-transistor comes on. If both the inputs now go Hi then the output will be pulled down to logic level Lo, otherwise it will remain floating at Hi.

The correct answer is generated on the output of the gate at the end of the evaluation period and is held static until the next precharge period.

Note that the output may hold the wrong answer if the inputs are allowed to go Hi and then go Lo during the evaluation period. Since there is no pull-up during the evaluation period, if the output goes Lo then it will remain so until the next precharge period. So for example if the inputs are initially Hi and then go Lo, then at the end of the evaluation period the output and the inputs will all be Lo, which is the wrong answer for a nand gate. To overcome this problem a restriction is imposed which states that "there should be no Hi to Lo transitions on the inputs of n-type gates during the evaluation period."

Similarly the working of p-type gates can be understood by considering a two input p-type nor gate. The structure of this is shown in figure 3.6. As in the case of n-type gates, this too has the precharge period and the evaluation period.

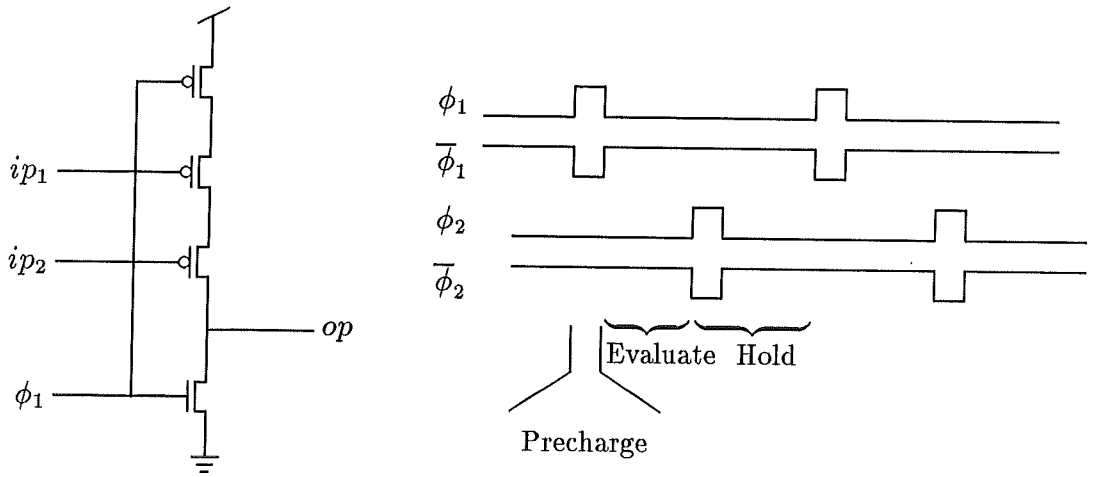


Figure 3.6: Two input p-type nor gate

Precharge $\phi_1 = \mathsf{Hi}$

During this period the output of the gate is pulled Lo by the enabled n-transistor. Any changes on the inputs during this time have no effect on the output since the top p-transistor is off and so the path to Vdd is effectively cut, i.e. the output cannot be pulled Hi.

Evaluation $\phi_1 = \mathsf{Lo}$

When ϕ_1 goes Lo the bottom n-transistor goes off and the top p-transistor comes on. If both the inputs now go Lo then the output will be pulled up to logic level Hi, otherwise it will remain floating at Lo.

Just as before the correct answer is generated on the output of the gate at the end of the evaluation period and is held static until the next precharge period.

For similar reasons to those of the n-type gates, the restriction that "there should be no Lo to Hi transitions on the inputs of p-type gates during the evaluation period" needs to be imposed.

3.2.3 Composition Rules for CLIC

The previous section's work shows that the outputs of the n-type gates cannot have Lo to Hi transitions during the evaluation period—which is exactly the requirements on the inputs of the p-type gates. Similarly the outputs of the p-type gates cannot have Hi to Lo transitions during the evaluation period—which meets the exact requirements for the inputs of the n-type gates. This is not by accident, but is built into the design style so that these gates can be composed together safely. To ensure that circuits designed using CLIC dynamic gates contain no timing hazards, rules have to be followed like the one above. All the rules necessary for the correct operation of the CLIC design style are presented in this section.

Perhaps the best method of presenting the CLIC composition design rules, is simply by listing them together with their motivations. Before giving these rules however, a few shorthands for the various types of gates and the clocks that drive them are presented:

pGate(ϕ) P-type gate driven by either ϕ_1 or ϕ_2.

nGate($\overline{\phi}$) N-type gate driven by either $\overline{\phi}_1$ or $\overline{\phi}_2$.

Latch(ϕ,$\overline{\phi}$) A latch driven by ϕ and $\overline{\phi}$ where these clock pairs are either, ϕ_1 and $\overline{\phi}_1$, or ϕ_2 and $\overline{\phi}_2$.

Note, if a clock phase ϕ, for example, is used in the statement of a rule then it is intended that the rule be interpreted as being in two parts: part one with all instances of ϕ and $\overline{\phi}$ replaced by ϕ_1 and $\overline{\phi}_1$, and part two with all instances of ϕ and $\overline{\phi}$ replaced by ϕ_2 and $\overline{\phi}_2$.

With these definitions in place the CLIC composition rules can now be given:

Rule 1. An nGate($\overline{\phi}$) may be driven by:

(a) pGate(ϕ)

(b) nGate($\overline{\phi}$) buffered by a static inverter

(c) Latch(ϕ,$\overline{\phi}$)

(d) Latch(ϕ,$\overline{\phi}$) buffered by a static inverter

Rule 2. A pGate(ϕ) may be driven by:

(a) nGate($\overline{\phi}$)

(b) pGate(ϕ) buffered by a static inverter

(c) Latch(ϕ,$\overline{\phi}$)

(d) Latch(ϕ,$\overline{\phi}$) buffered by a static inverter

Rule 3. A Latch(ϕ_1,$\overline{\phi}_1$) may be driven by:

(a) nGate($\overline{\phi}_2$)

(b) nGate($\overline{\phi}_2$) buffered by a static inverter

(c) pGate(ϕ_2)

(d) pGate(ϕ_2) buffered by a static inverter

(e) Latch(ϕ_2,$\overline{\phi}_2$)

(f) Latch(ϕ_2,$\overline{\phi}_2$) buffered by a static inverter

Rule 4. A Latch(ϕ_2,$\overline{\phi}_2$) may be driven by:

(a) nGate($\overline{\phi}_1$)

(b) nGate($\overline{\phi}_1$) buffered by a static inverter

(c) pGate(ϕ_1)

(d) pGate(ϕ_1) buffered by a static inverter

(e) Latch(ϕ_1,$\overline{\phi}_1$)

(f) Latch(ϕ_1,$\overline{\phi}_1$) buffered by a static inverter

Rules 1 and 2 above ensure that the requirements on the inputs of dynamic CLIC gates are always satisfied. For example the inputs of n-type dynamic CLIC gates must not have any Hi to Lo transitions during the evaluation phase. So the four possible ways in which such a gate can be driven as listed above ensure that the inputs behave correctly, i.e. in all of the four cases the output of such cluster of gates can not have Hi to Lo transitions during the evaluation period. Rule 2 follows a similar argument except that it is stated for p-type gates which must not have Lo to Hi transitions on the inputs during the evaluation phase.

Rules 3 and 4 ensure that the inputs to the Latch devices are always stable. This is done by ensuring that the inputs are from devices which are driven by the other phase of the clock. So the input to a latch driven by clocks ϕ_1 and $\overline{\phi}_1$ can come from any circuit terminated by a device driven by ϕ_2 and/or $\overline{\phi}_2$, and/or buffered by a static inverter. Note that if the input is from another Latch device driven by the opposite phase of the clock then the stable requirement is trivially satisfied. However if the input is from a dynamic gate then it is possible that the input may not be stable because the gate may be in the evaluation phase. To ensure that the input will be stable, the period of the master clock is assumed to be long enough to allow the input circuit to finish evaluation. Indeed this is necessary for the CLIC design style to function correctly.

3.3 Formalising the CLIC Design Style

The objective here is to use the various formal techniques to capture the major concepts which go to making a useful design style. There are a number of levels at which the development of the circuit design could be viewed: from the physics of the semiconductor devices, to the top level specifications of a system given in vague terms using English like specification languages. A designer cannot hope to view all his circuit at all of these levels at once. He will neither have the capacity nor the necessary expertise in all of these areas, so the best he can do is to work with simple models of the lower level implementations of the various devices. The formalisation of the CLIC design style presented here will thus reflect the sort of devices a logic designer might reasonably be expected to work from.

To see how to formalise the CLIC design style, the first thing that needs to be looked at is the form of the correctness statement at the top level. For any given device, the correctness statement states that "the implementation together with some conditions on the inputs implies the specification together with some conditions on the outputs." More formally this can be stated as follows:

$$\left(\begin{array}{l}\mathsf{Dev\_Imp}(ip_1,\ \ldots\ ip_n, op_1,\ \ldots\ op_m)\ \wedge \\ \mathsf{Ip\_Cond}\ ip_1\ \wedge\ \cdots\ \wedge\ \mathsf{Ip\_Cond}\ ip_n\end{array}\right)\ \supset$$
$$\left(\begin{array}{l}\mathsf{Dev\_Spec}(ip_1,\ \ldots\ ip_n, op_1,\ \ldots\ op_m)\ \wedge \\ \mathsf{Op\_Cond}\ op_1\ \wedge\ \cdots\ \wedge\ \mathsf{Op\_Cond}\ op_m\end{array}\right) \tag{3.1}$$

The derivation of the correctness statement of a device which is composed of two lower level devices, requires that the input-conditions and the output-conditions match for all those lines which are to be connected between them. Then by simple logical manipulation, the top level correctness statement can be derived purely from the lower level correctness statements.

This then is the overview of our methodology for the formalisation of a design style. Naturally the Ip_Cond and Op_Cond predicates will have to be defined to reflect the design rules for the particular design style. Also other parameters may need to be added to these predicates to formalise any peculiarities of the design style. Infact these will become apparent in the remainder of this chapter as the CLIC design style is pushed through this process.

Before the correctness statement for the various CLIC gates can be derived, a few preliminary axioms and definitions are needed. For example, the model of transistors, capacitors, and the sort of signal values that are to be propagated around the circuit need to be defined. Then the two phased non-overlapping clock needs to be formalised. And then finally the correctness statements for the various CLIC gates can be derived.

3.4 Formal Definitions of Device Primitives

Before choosing a particular model, a few observations need to be made which influence this decision. For example, note that the set of rules for constructing CLIC circuits are given at the gate level, and not at the transistor level. So the model actually used will have to be appropriately biased to allow easy abstraction to the gate level, and at the same time be accurate enough to allow the various failure modes to be observable. Instead of showing the various different models which could be used for this work and listing their strengths and weaknesses, only the one actually used is presented here.

3.4.1 The Signal Values

Any node in the circuit can only have one of four values Hi, Lo, Er and Zz. The values Hi and Lo represent the voltage levels high and low respectively. The value Zz captures the concept of a node which is not being driven, namely high impedence. And finally the value Er represents the state of a node which is being driven both high and low at the same time, namely error.

On similar lines to Bryant's work [Bryant 81], these four values can be thought of as having the strength ordering as shown in figure 3.7. Here the value Er has the highest strength, the value Zz has the lowest strength, and the values Hi and Lo have equal strengths somewhere in between. These values with these strengths form a complete lattice [Birkhoff 48]. The resultant value at a node which has more than one value being driven onto it, can be defined by using the least upper bound ($\sqcup$) over this lattice.

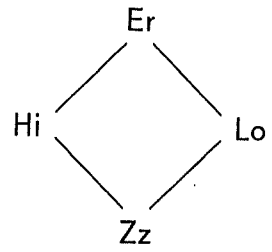


Figure 3.7: The four valued signals for CLIC

These four signal values are defined in HOL as a new type. This is a tedious, but a logically straight forward task. Having declared it as a new type the operations of $\sqcup$ are then easily defined. The following theorem which is derived from these definitions states all the useful properties of the operation $\sqcup$ over these newly defined values.

$$\vdash \begin{pmatrix} & \mathsf{Hi} \sqcup \mathsf{Lo} = \mathsf{Er} & \wedge \\ \forall x. & \mathsf{Er} \sqcup x = \mathsf{Er} & \wedge \\ \forall x. & \mathsf{Zz} \sqcup x = x & \wedge \\ \forall x\ y. & x \sqcup y = y \sqcup x & \end{pmatrix} \tag{3.2}$$

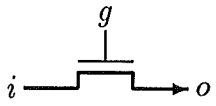
With this simple signal algebra in place, the various definitions of the primitives for the CMOS technology can now be defined.

3.4.2 CMOS Primitives

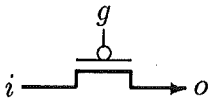
There are only six primitive devices that need to be defined in order to model all the various components as used in the CLIC design style. These are the two power rails, namely Vdd and Gnd, the n-type and the p-type transistors, namely N_Tran and P_Tran, the unit capacitor Cap_1 and the Join device for bringing together two signals at a node. These are defined in HOL as follows:

$$\mathsf{Vdd}(x) \quad =_{def} \quad \forall t.\ x(t) = \mathsf{Hi} \tag{3.3}$$

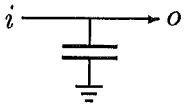
$$\mathsf{Gnd}(x) \quad =_{def} \quad \forall t.\ x(t) = \mathsf{Lo} \tag{3.4}$$



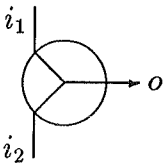
$$\mathsf{N\_Tran}(g,i,o) \quad =_{def} \quad \forall t.\ o(t) = ((g(t) = \mathsf{Lo}) \Rightarrow \mathsf{Zz} \mid i(t)) \tag{3.5}$$



$$\mathsf{P\_Tran}(g,i,o) \quad =_{def} \quad \forall t.\ o(t) = ((g(t) = \mathsf{Hi}) \Rightarrow \mathsf{Zz} \mid i(t)) \tag{3.6}$$



$$\mathsf{Cap}_1(i,o) \quad =_{def} \quad \forall t.\ o(t) = (\sim(i(t) = \mathsf{Zz}) \Rightarrow i(t) \mid i(t-1)) \tag{3.7}$$



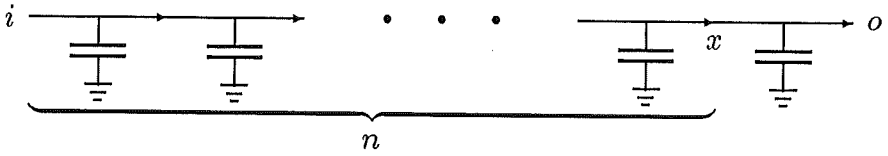
$$\mathsf{Join}(i_1,i_2,o) \quad =_{def} \quad \forall t.\ o(t) = i_1(t) \sqcup i_2(t) \tag{3.8}$$

The definitions for Vdd and Gnd (equations 3.3 and 3.4) assert the values Hi and Lo respectively on their nodes for all time. So to say "Vdd(x)" means that the value on the node x is always Hi.

The definitions of the two types of transistors (equations 3.5 and 3.6) are based on unidirectional models. In order to model the worst case in the context of the CLIC design style, these definitions assert that the transistors are ON unless the values on the gates are such as to turn them OFF. A more accurate model could have been used such as a partial definition which states exactly when the transistor is ON and when it is OFF. An example of this for an n-type transistor which says nothing when the value on the gate is other than Hi or Lo can be stated as follows:

$$\mathsf{N\_Tran}(g,i,o) \quad =_{def} \quad \forall t. \begin{pmatrix} (g(t) = \mathsf{Hi}) \supset (o(t) = i(t)) & \wedge \\ (g(t) = \mathsf{Lo}) \supset (o(t) = \mathsf{Zz}) & \end{pmatrix}$$

The definition for the Cap_1 device given in equation 3.7. This defines the behaviour of a capacitor which decays its stored value in one unit of time. Capacitors which have a decay time of n units of time are then constructed by connecting n Cap_1 devices together in a chain as follows:



$$\begin{array}{lllll} \mathsf{Cap} & 0 & i\ o & =_{def} & o = i \\ \mathsf{Cap} & (n{+}1) & i\ o & =_{def} & \exists x.\ \mathsf{Cap}\ n\ i\ x \ \wedge \\ & & & & \quad \mathsf{Cap}_1(x,o) \end{array}$$

From this definition the general behaviour of the Cap device is derived. This can be stated at its most abstract level as follows:

$$\vdash \ \mathsf{Cap}\ n\ i\ o \ = \ (o \ = \ \mathsf{Last}\ n\ i)$$

Where the predicate Last is defined:

$$\begin{array}{lllll} \mathsf{Last} & 0 & i & =_{def} & i \\ \mathsf{Last} & (n{+}1) & i\ t & =_{def} & (\sim(i\ t \ = \ \mathsf{Zz}) \ \Rightarrow \ i\ t \ \mid \ \mathsf{Last}\ n\ i\ (t{-}1)) \end{array}$$

Note that all further devices described in the remainder of this chapter will only be built out of the basic building blocks defined here.

3.5 Formal Definition of Clock

The Clock as described earlier is derived from a single square wave input. The formal definition presented here is not derived from such a single input but is simply defined to be that which such a circuit might generate. This is because the model of the various gates does not include delay due to the rather simple model of the primitives out of which these gates are constructed. It would not be too difficult to derive the given definition as an abstraction of what might be generated by the circuit if a different model for the gates were to be used.

The predicate Clock is now defined as a relation over four arguments. These four arguments would represent the four lines which would be generated by the clock generator circuit. The behaviour of the four clock lines is defined by first giving the behaviour of a single clock line, and then relating the behaviour of the other three lines to this. Figure 3.8 shows the behaviour of the clock as waveforms with some arbitrary starting point.

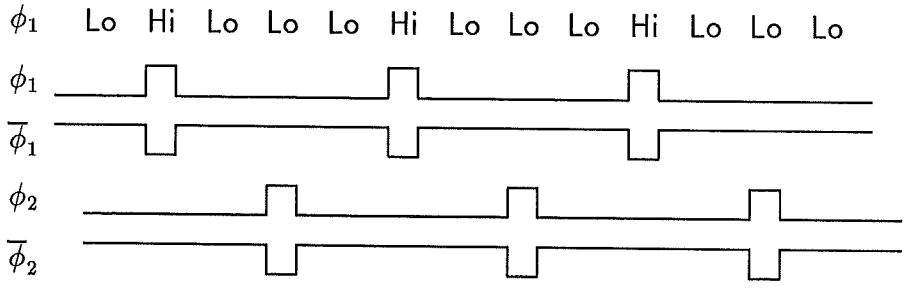


Figure 3.8: Graphical representation of the four clock lines

The clock line which is to be formally defined is the ϕ_1 line. This is also shown in figure 3.8 as a sequence of Hi s and Lo s. Note that the uncertainty states are not taken into account here. If they were taken into account then it would result in the cycle being split into eight intervals of time, rather than the present four intervals of time. These intervals are not equal in length in real time, but they are abstracted such that the clock cycle consists of four units of time in the abstract time space.

The reason for splitting the clock interval into four rather than eight subintervals, is to initially simplify the formalisation of the design style. Once these techniques are understood, the work presented here could be expanded to go into more detail at the timing, and the primitive device modeling level. For the moment however, the formalisation of the clock as presented here is at the coarser granularity.

Informally, the predicate Clock is defined by first stating the behaviour of the ϕ_1 waveform, then shifting this waveform to generate the ϕ_2 waveform, and finally inverting both of these to generate the $\overline{\phi}_1$ and $\overline{\phi}_2$ waveforms. This is stated formally as follows:

$$\begin{aligned} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) \quad =_{def} \quad & \mathsf{Cycle}\ \phi_1 \quad \wedge \\ & \mathsf{Shift}\ \phi_1\ \phi_2\ \wedge \\ & \mathsf{Invert}\ \phi_1\ \overline{\phi}_1\ \wedge \\ & \mathsf{Invert}\ \phi_2\ \overline{\phi}_2 \end{aligned} \tag{3.9}$$

Now the definitions for the various predicates used to define Clock can be given. The simplest two, namely Shift and Invert, are defined first.

$$\mathsf{Shift}\ \phi_1\ \phi_2 \quad =_{def} \quad \forall t.\ \phi_2(t) = \phi_1(t{+}2) \tag{3.10}$$

$$\mathsf{Invert}\ \phi\ \overline{\phi} \quad =_{def} \quad \forall t.\ \overline{\phi}\ t = \mathsf{NOT}(\phi\ t) \tag{3.11}$$

Where NOT is defined as the negation function over the values Hi, Lo and Er. Note that this is a partial definition, which says nothing about what happens if the value Zz is passed as an argument to this predicate. Formally this is defined as follows:

$$\begin{aligned} \mathsf{NOT\ Hi} \quad & =_{def} \quad \mathsf{Lo} \quad \wedge \\ \mathsf{NOT\ Lo} \quad & =_{def} \quad \mathsf{Hi} \quad \wedge \\ \mathsf{NOT\ Er} \quad & =_{def} \quad \mathsf{Er} \end{aligned}$$

So far, only the way in which all the clock lines are related to ϕ_1 have been defined, but a formal definition for the predicate Cycle has not been given. Before doing this, the key aspects of the behaviour of ϕ_1 are informally stated:

- ϕ_1 is cyclic over four units of time
- During *any* four consecutive units of time the value on ϕ_1 is Hi exactly once and Lo for the other three units.
- ϕ_1 can start in any of its four possible states.

Each of these three informal statements can now be formalised in logic resulting in the following definition for the predicate Cycle.

$$\begin{aligned}\mathsf{Cycle}\ \phi\ =_{def}\ & (\forall t.\ \phi(t) = \phi(t{+}4)) \wedge \\ & (\ \mathsf{Cycle\_1}\ \phi\ 0 \vee \\ & \ \ \mathsf{Cycle\_1}\ \phi\ 1 \vee \\ & \ \ \mathsf{Cycle\_1}\ \phi\ 2 \vee \\ & \ \ \mathsf{Cycle\_1}\ \phi\ 3\ )\end{aligned} \tag{3.12}$$

Where the predicate Cycle_1 is defined:

$$\begin{aligned}\mathsf{Cycle\_1}\ \phi\ t_0\ =_{def}\ & (\phi(t_0) = \mathsf{Hi}) \wedge \\ & (\phi(t_0{+}1) = \mathsf{Lo}) \wedge \\ & (\phi(t_0{+}2) = \mathsf{Lo}) \wedge \\ & (\phi(t_0{+}3) = \mathsf{Lo})\end{aligned} \tag{3.13}$$

This seems like a lengthy definition for Clock and it could have been shorter but for two reasons. Firstly, it closely mimics the way in which the signals on the clock lines were informally described, and secondly it is of the form which allows some of the latter lemmas to be more easily derived.

However, some of the more elegant definitions which were considered at the time of defining the cyclic property of clock are shown below as equations 3.14 and 3.15. These are all provably equivalent to the definition given above, so any of them could be used.

$$\begin{aligned}\mathsf{Cycle}\ \phi\ =_{def}\ & (\exists t.\ \phi(t) = \mathsf{Hi}\ \wedge \\ & \qquad \phi(t{+}1) = \mathsf{Lo}\ \wedge \\ & \qquad \phi(t{+}2) = \mathsf{Lo}\ \wedge \\ & \qquad \phi(t{+}3) = \mathsf{Lo}\ ) \wedge \\ & (\forall t.\ \phi(t{+}4) = \phi(t)\ )\end{aligned} \tag{3.14}$$

$$\begin{aligned}\mathsf{Cycle}\ \phi\ =_{def}\ & \exists n.\ (0 \leq n \leq 3)\ \wedge \\ & \qquad \forall t.\ \phi(t) = ((\mathsf{MOD4}\ t\ =\ n) \Rightarrow \mathsf{Hi} \mid \mathsf{Lo})\end{aligned} \tag{3.15}$$

Where MOD4 is the remainder of dividing its argument by 4.

Note how the various properties of Clock are separated into different predicates. This is done deliberately so that the formalisation can be followed more easily. It also reduces the risk of any errors being introduced in the translation process going from the informal description to the formal one. Finally it is simply easier to read and comprehend if the various ideas are individually defined.

3.6 Formalising the Validity Conditions of CLIC Gates

The rules governing the interconnection of CLIC gates have been described earlier in an informal way. Before formalising them, it is necessary to fully understand their operation. Consider, for example, an n-type logic gate driving a p-type logic gate. During the precharge period, the output of the n-type gate is precharged Hi, and so the input of the p-type gate is also Hi. At this point it is worth pointing out that the inputs of a p-type logic gate are always connected to the gates of p-type transistors. Since these inputs have the value Hi on them during the precharge phase, all these transistors will be off; i.e. if the logic gate went into the evaluation phase now, the output node of this p-type logic gate would become isolated. So when the clock changes and puts both of these logic gates into the evaluation phase, the output of the p-type gate does not change until its inputs change. However, if the other situation were present where the input of the p-type gate was held Lo, then as soon as the gate went into its evaluation phase the output would change to Hi. Now no matter what happens to the inputs, the output cannot be changed to Lo until the next precharge period. So effectively the gate has erroneously changed its output value.

In summary it can be stated that during the evaluation phase, a p-type gate *must* not have Lo to Hi transitions ($\mathcal{F}$) on its inputs, and an n-type gate *does* not have Lo to Hi transitions ($\mathcal{F}$) on its output. So to capture this sort of behaviour, a single predicate needs to be defined which captures both the constraints on the inputs of a p-type gate, and the behaviour of the output of an n-type gate. Here then is the formal definition of a predicate Wb which is designed to capture exactly this property.

$$\mathsf{Wb}\ x\ \phi \quad =_{def} \quad \forall t.\ \begin{pmatrix} \phi(t{+}1) = \phi(t) \wedge \\ x(t) \quad\quad = \phi(t) \end{pmatrix} \supset\ x(t{+}1) = x(t) \tag{3.16}$$

This predicate relies on the fact that the gates are clocked and that the waveforms on the clock lines are correctly behaved. What it states is that "the node x is said to be 'Well Behaved' with respect to ϕ," where ϕ is one of the four clock line as defined by the predicate Clock. The term "Well Behaved" refers to the fact that only the correct sort of transitions occur on the the node x.

So for example the output node (op) of an n-type gate driven by $\overline{\phi}_1$ satisfies "Wb op ϕ_1," which is exactly the required input conditions for a p-type gate driven by ϕ_1. In this context what "Wb op ϕ_1" means is that while ϕ_1 is Lo and $\overline{\phi}_1$ is

Hi, i.e. the n-type gate is in its evaluation phase, then the output node (*op*) of the n-type gate cannot have a rising edge on it during this period. This is indeed correct, since once the output of an n-type gate has been discharged Lo, then it cannot go to Hi again until the clock rises and precharges the gate as discussed above.

The checking of the various constraints imposed by the rules as listed above can now all be done by this one predicate. However this predicate relies on the formal definition of Clock and must always be used in conjunction with it. This is not a restriction since CLIC is a dynamic design style, and so all CLIC gates will require the existence of Clock for their correct behaviour.

3.7 Deriving the Correctness Statements of CLIC Primitive Gates

There are four types of gates in the CLIC design methodology: the n-type gate, the p-type gate, the latch and the static inverter. Statements of correctness can be individually derived for the latch and the static inverter, but it would be foolish to simply derive a statement of correctness for each of the various n-type and p-type gates separately. Rather than doing this it is far better to derive some general theorems which will then be useful for generating the statement of correctness for the individual n-type and p-type gates.

3.7.1 N-type and P-type Logic Gates

For any general theorems to be proved of n-type or p-type gates, the common elements of these various gates need to be identified and separated first. A simple split would be to separate the set of components which perform the logic specific function into one bag, and the remainder into another. This remainder of an n-type gate is called the nShell, since it has a hole in it into which other components need to be inserted before it can function as an n-type CLIC gate. The circuit which implements the nShell is illustrated in figure 3.9, and its structure can be formally defined in logic as shown below in equation 3.17.

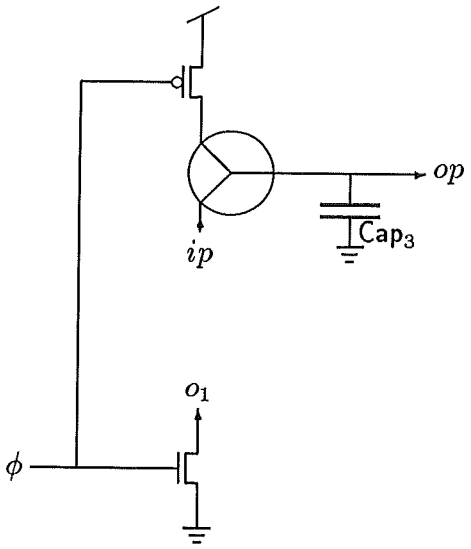


Figure 3.9: nShell as used in CLIC

$$\begin{aligned}
\mathsf{nShell}(\phi, o_1, ip, op) \quad =_{def} \quad & \exists p_0\ p_1\ p_2\ p_3. \\
& \mathsf{Gnd}(p_0) \quad \wedge \\
& \mathsf{Vdd}(p_1) \quad \wedge \\
& \mathsf{N_Tran}(\phi, p_0, o_1) \wedge \\
& \mathsf{P_Tran}(\phi, p_1, p_2) \wedge \\
& \mathsf{Join}(p_2, ip, p_3) \quad \wedge \\
& \mathsf{Cap}_3(p_3, op)
\end{aligned} \tag{3.17}$$

The Cap_3 device in the above definition is simply a capacitor with a "memory" of three units of time, just as Cap_1 has a "memory" of one unit of time. Note that Cap_3 is derived by composing three Cap_1 devices together.

Before proceeding further, the property which is held true of all those cluster of devices which may be inserted into this nShell needs to be defined. By studying the mechanism of an n-type gate it is apparent that the cluster of devices which is inserted in the nShell performs one of two functions: it either maintains a link between the ip and the o_1 nodes of the nShell, or it doesn't. This property shall be referred to as Opt_Link, standing for optional link, and it can be formally stated as follows:

$$\mathsf{Opt\_Link}(ip, op) \quad =_{def} \quad \forall t.\ (op\ t = ip\ t) \vee (op\ t = \mathsf{Zz}) \tag{3.18}$$

Here it is worth noting that the true property of two nodes being linked or not linked is not actually captured by this predicate. This is because a directional flow

of information model is being used for the various primitive devices. The best that can be done under the circumstances as stated, is say that the values on the two node are equal or that the input node to the nShell has a floating value on it. This is still not quite enough, but the extra conditions will be stated later when they are needed. However, this predicate provides enough information for the following two properties of the nShell to be derived.

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) \wedge \\ \mathsf{nShell}(\overline{\phi}_1, a, b, op) \quad \wedge \\ \mathsf{Opt\_Link}(a, b) \end{pmatrix} \supset \mathsf{Wb}\ op\ \phi_1 \tag{3.19}$$

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) \wedge \\ \mathsf{nShell}(\overline{\phi}_1, a, b, op) \quad \wedge \\ \mathsf{Opt\_Link}(a, b) \quad \wedge \\ \mathsf{isHi}\ \phi_1\ t \end{pmatrix} \supset \begin{pmatrix} \mathsf{Def}\ op\ \ t \quad \wedge \\ \mathsf{Def}\ op\ (t{+}1) \ \wedge \\ \mathsf{Def}\ op\ (t{+}2) \ \wedge \\ \mathsf{Def}\ op\ (t{+}3) \end{pmatrix} \tag{3.20}$$

Where

$$\mathsf{isHi}\ \phi\ t \quad =_{def} \quad (\phi(t) = \mathsf{Hi})$$

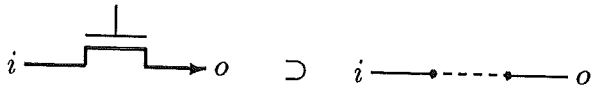
$$\mathsf{Def}\ x\ t \quad =_{def} \quad (x(t) = \mathsf{Hi}) \vee (x(t) = \mathsf{Lo})$$

The first theorem (equation 3.19) can be interpreted as saying that, *if* the nShell is implemented correctly, *and* it is correctly driven by clock, *and* the cluster of devices placed in it are correctly behaved in that they have the property of Opt_Link, *then* the output will be "Well Behaved," i.e. the output will not have Lo to Hi transitions during the evaluation phase. The second theorem (equation 3.20) says that given the same assumptions, and additionally assuming that at some time t the clock phase ϕ_1 goes Hi, then the output will be "Well Defined" for the times t to t+4, i.e. the output will be either Hi or Lo.

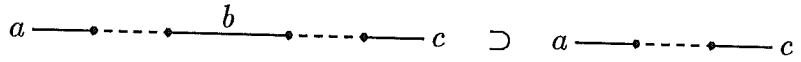
Having derived these general theorems with the property Opt_Link as an assumption over the two node where external devices are placed, it is necessary to show that this Opt_Link property is derivable for all clusters of elements that may be inserted in the nShell. For this to be truly general it will be necessary to talk of the structure of an arbitrary cluster of devices.

Any logic function which is implementable can be simplified into a particular network of transistors. This network only contains transistors in series and/or

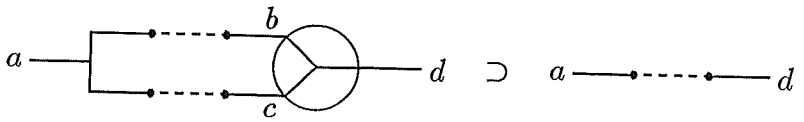
transistors in parallel. All the control inputs to a network of this kind are connected only to the gates of the transistors, and the outputs of the transistor are joined together using the Join device. On the basis of this, and the three derived theorems (equations 3.21, 3.22, and 3.23), any cluster containing only parallel and/or series transistor networks, can be shown to maintain the Opt_Link property across the two terminals by which the cluster is connected to the nShell.



$$\vdash \mathsf{N\_Tran}(g, i, o) \supset \mathsf{Opt\_Link}(i, o) \tag{3.21}$$



$$\vdash \begin{pmatrix} \mathsf{Opt\_Link}(a, b) \wedge \\ \mathsf{Opt\_Link}(b, c) \end{pmatrix} \supset \mathsf{Opt\_Link}(a, c) \tag{3.22}$$



$$\vdash \begin{pmatrix} \mathsf{Opt\_Link}(a, b) \wedge \\ \mathsf{Opt\_Link}(a, c) \wedge \\ \mathsf{Join}(b, c, d) \end{pmatrix} \supset \mathsf{Opt\_Link}(a, d) \tag{3.23}$$

This can be illustrated by a simple example: the two input n-type nand gate of figure 3.5 on page 34. The structure of this can be formally defined as follows:

$$\begin{aligned} \mathsf{nNand}_2(\phi, ip_1, ip_2, op) \quad =_{def} \quad & \exists p_1\ p_2\ p_3. \\ & \mathsf{nShell}(\phi, p_1, p_3, op) \wedge \\ & \mathsf{N\_Tran}(ip_1, p_1, p_2) \wedge \\ & \mathsf{N\_Tran}(ip_2, p_2, p_3) \end{aligned} \tag{3.24}$$

Now by using the above three derived theorems about the network of n-type transistors (equations 3.21, 3.22 and 3.23), and the two derived theorems about the behaviour of the nShell (equations 3.19 and 3.20), the following two properties can be derived. The derivation procedure is relatively straightforward based on simple logical manipulations of these equations. The two theorems thus derived state that the output of the two input n-type nand gate is "Well Behaved" (equation 3.25), and "Well Defined" over a certain interval of time with reference to the point when ϕ_1 is Hi (equation 3.26).

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_1, ip_1, ip_2, op) & \end{pmatrix} \supset \mathsf{Wb}\ op\ \phi_1 \tag{3.25}$$

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_1, ip_1, ip_2, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Def}\ op\ \ t & \wedge \\ \mathsf{Def}\ op\ (t{+}1) & \wedge \\ \mathsf{Def}\ op\ (t{+}2) & \wedge \\ \mathsf{Def}\ op\ (t{+}3) & \end{pmatrix} \tag{3.26}$$

So far a technique has been presented which shows how to prove that the output of any n-type gate is "Well Behaved" and "Well Defined." Nothing has been said about the derivation of the logical behaviour of these gates. In order to derive the logical behaviour of such gates, considerably more complex theorems are involved using other properties similar to Opt_Link. These properties include Link, No_Link and Wb_Link which state under what circumstances a "link" exists across the two nodes of the cluster of devices inserted in the nShell. These can be formally defined as follows:

$$\mathsf{Link}(x, y)\ t \ =_{def}\ (y\ t\ =\ x\ t) \tag{3.27}$$

$$\mathsf{No\_Link}(x, y)\ t \ =_{def}\ (y\ t\ = \mathsf{Zz}) \tag{3.28}$$

$$\mathsf{Wb\_Link}(x, y)\ \phi\ t \ =_{def} \begin{pmatrix} \phi(t) = \phi(t{+}1) & \wedge \\ x(t) \neq \mathsf{Zz} & \wedge \\ x(t{+}1) \neq \mathsf{Zz} & \wedge \\ \mathsf{Link}(x, y)\ t & \end{pmatrix} \supset \mathsf{Link}(x, y)(t{+}1) \tag{3.29}$$

The first two of these theorems (equations 3.27 and 3.28) are intended to capture the concepts that the two nodes are linked, or not linked respectively. The No_Link predicate as defined here doesn't really capture the concept that the two nodes are not linked, but this is the best that can be done with respect to the unidirectional flow of information model being used. The third theorem (equation 3.29) is needed to ensure that the inputs to the cluster are well behaved. It can be thought of saying that "if during the evaluation phase a link is formed across the cluster of devices, then it will continue to be a link for the remainder of the evaluation period." This predicate required that the line ϕ be one of the clock lines obeying the relation stated by Clock.

With the Opt_Link predicate, five theorems were derived which were then used collectively to show that the output of any n-type CLIC gate is "Well Behaved"

and "Well Defined." Similar theorems are derived for each of these predicates and they are then collectively used to derive the logical behaviour of n-type CLIC gates.

An overview of the algorithm is to show that there exists a function $\mathcal{F}$ over the inputs such that, *if* during the evaluation phase the inputs to the gate satisfy this function $\mathcal{F}$, *then* the cluster of devices will produce a 'link' across the two ports by which it is connected to the nShell. Hence the resultant value on the output of an n-type gate will be Lo. In formal notation this is represented as a theorem of the form:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nDev}(\phi_1, \overline{\phi}_1, i_1, \cdots i_n, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \vdots & \end{pmatrix} \supset \begin{pmatrix} \mathcal{F}(i_1, \cdots i_n)(t{+}1) & \supset \\ \sim\mathsf{Val\_Abs}\ op\ (t{+}1) & \end{pmatrix}$$

Note that the value on the output is being abstracted to the boolean domain by use of the Val_Abs predicate. The formal definition for this is given in equation 3.30 below. This is used in the correctness statements of every CLIC gate to state the behaviour at the boolean level.

$$\begin{array}{lll} \mathsf{Val\_Abs}\ x\ t & =_{def} & ((x\ t = \mathsf{Hi}) \Rightarrow \mathsf{T} \mid \\ & & ((x\ t = \mathsf{Lo}) \Rightarrow \mathsf{F} \mid \mathsf{Arb})) \end{array} \tag{3.30}$$

Where Arb is some arbitrary value of the correct type, in this case of type boolean. The formal definition of this is given by using the select operator (ε) as follows:

$$\mathsf{Arb} \quad =_{def} \quad \varepsilon\ x{:}*.\ \mathsf{F}$$

Returning to the algorithm. If the inputs did not satisfy this function $\mathcal{F}$, then a link would not exist across the cluster, and as a result the output would remain Hi for an n-type gate. Again this is captured in formal notation as follows:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nDev}(\phi_1, \overline{\phi}_1, i_1, \cdots i_n, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \vdots & \end{pmatrix} \supset \begin{pmatrix} \sim\mathcal{F}(i_1, \cdots i_n)(t{+}1) & \supset \\ \mathsf{Val\_Abs}\ op\ (t{+}1) & \end{pmatrix}$$

Now by combining these two results the final form of the derived result is obtained. This is stated in the formal notation as the following theorem.

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nDev}(\phi_1, \overline{\phi}_1, i_1, \cdots i_n, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \vdots & \end{pmatrix} \supset \begin{pmatrix} \mathcal{F}(i_1, \cdots i_n)(t{+}1) & = \\ \mathsf{Val\_Abs}\ op\ (t{+}1) & \end{pmatrix}$$

Note that the right hand side of this implication is generally what is understood to be the behaviour of the gate. But this derived theorem shows that the correctness of the logical behaviour is dependent on certain conditions; namely, the implementation, the correct operation of the clock, and the correct operation of the inputs.

This technique can now be applied to the the analysis of a two input n-type nand gate. The final derived result for this is presented below.

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_1, ip_1, ip_2, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ ip_1\ (t{+}1) & \wedge \\ \mathsf{Def}\ ip_2\ (t{+}1) & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Val\_Abs}\ op\ (t{+}1) & = \\ \sim \begin{pmatrix} \mathsf{Val\_Abs}\ ip_1\ (t{+}1) & \wedge \\ \mathsf{Val\_Abs}\ ip_2\ (t{+}1) & \end{pmatrix} & \end{pmatrix} \quad (3.31)$$

This theorem only states the behaviour of the nand gate at time $t{+}1$, where t is the time when ϕ_1 is Hi. In order to pass the evaluated result to the gates in the next section driven by the other phases of the clock, the results have to be latched. To eliminate any hazards, the CLIC design style dictates that such results should only be latched using the opposite phase of the clock. So in this case the results will be latched when ϕ_2 next goes Hi. This will happen at time $t{+}2$ using the same time reference. So a correctness statement needs to be derived stating the behaviour of the nand gate at time $t{+}2$. This is done by following a similar argument as for the $t{+}1$ case, but this time, use is made of the Wb_Link property. The final result for the two input nand gate is shown below in equation 3.32.

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_1, ip_1, ip_2, op) & \wedge \\ \mathsf{Wb}\ ip_1\ \overline{\phi}_1 & \wedge \\ \mathsf{Wb}\ ip_2\ \overline{\phi}_1 & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ ip_1\ (t{+}1) & \wedge \\ \mathsf{Def}\ ip_2\ (t{+}1) & \wedge \\ \mathsf{Def}\ ip_1\ (t{+}2) & \wedge \\ \mathsf{Def}\ ip_2\ (t{+}2) & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Val\_Abs}\ op\ (t{+}2) & = \\ \sim \begin{pmatrix} \mathsf{Val\_Abs}\ ip_1\ (t{+}2) & \wedge \\ \mathsf{Val\_Abs}\ ip_2\ (t{+}2) & \end{pmatrix} & \end{pmatrix} \quad (3.32)$$

Having described the derivation procedure for the n-type logic gates, an almost identical argument is followed for the derivation of p-type logic gates. The work is very similar, even to the point where a considerable number of the intermediate results are common to both. This is not covered here to avoid repetition.

3.7.2 The Latch

The Latch device, as described earlier, is also known as the C^{2}MOS latch, and its structure is shown in figure 3.10. This is formally captured as follows:

$$\begin{aligned}
\mathsf{Latch}(\phi, \overline{\phi}, ip, op) \quad =_{def} \quad & \exists p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6 . \\
& \mathsf{Gnd}(p_0) \quad \wedge \\
& \mathsf{Vdd}(p_1) \quad \wedge \\
& \mathsf{N_Tran}(ip, p_0, p_2) \wedge \\
& \mathsf{N_Tran}(\phi, p_2, p_4) \wedge \\
& \mathsf{P_Tran}(\overline{\phi}, p_3, p_5) \wedge \\
& \mathsf{P_Tran}(ip, p_1, p_3) \wedge \\
& \mathsf{Join}(p_4, p_5, p_6) \quad \wedge \\
& \mathsf{Cap}_3(p_6, op)
\end{aligned} \tag{3.33}$$

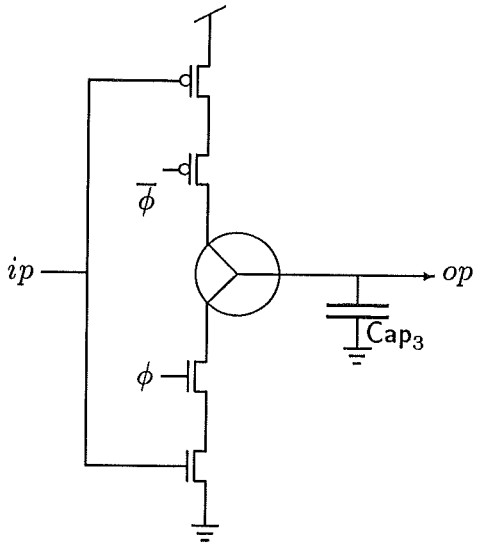


Figure 3.10: The Latch as used in CLIC

Since this is simply a one-off result, the derivation is not important. But the final derived result is, so only that is presented here. The full behaviour of the Latch device is summarised by the following three derived theorems.

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{Latch}(\phi_1, \overline{\phi}_1, ip, op) & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Wb}\ op\ \phi_1 & \wedge \\ \mathsf{Wb}\ op\ \overline{\phi}_1 & \end{pmatrix} \tag{3.34}$$

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{Latch}(\phi_1, \overline{\phi}_1, ip, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ ip\ t & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Def}\ op\ \ t & \wedge \\ \mathsf{Def}\ op\ (t{+}1) & \wedge \\ \mathsf{Def}\ op\ (t{+}2) & \wedge \\ \mathsf{Def}\ op\ (t{+}3) & \end{pmatrix} \tag{3.35}$$

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{Latch}(\phi_1, \overline{\phi}_1, ip, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ ip\ t & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Val\_Abs}\ op\ \ t & = \sim\mathsf{Val\_Abs}\ ip\ t & \wedge \\ \mathsf{Val\_Abs}\ op\ (t{+}1) & = \sim\mathsf{Val\_Abs}\ ip\ t & \wedge \\ \mathsf{Val\_Abs}\ op\ (t{+}2) & = \sim\mathsf{Val\_Abs}\ ip\ t & \wedge \\ \mathsf{Val\_Abs}\ op\ (t{+}3) & = \sim\mathsf{Val\_Abs}\ ip\ t & \end{pmatrix} \tag{3.36}$$

The first of these theorems (equation 3.34) captures the fact that the output of the Latch may drive any of p-type or n-type gates, or both at the same time. The second theorem (equation 3.35) states that the output is "Well Defined," so the results can be abstracted into the boolean domain by use of the Val_Abs abstraction function. This is an important result because the Val_Abs abstraction function is only defined for values Hi and Lo. Finally the third theorem (equation 3.36) gives the logical behaviour between the input and the output at the abstract level, i.e. on the clock tick the input is inverted and passed to the output where it is held static until the next clock tick.

3.7.3 The Static Inverter

This is the only device in the entire CLIC design style which does not need one of the clock lines for it to function correctly. It has only two external ports namely the input (*ip*) and output (*op*) ports, and its behaviour could perfectly be defined without the use of the Clock predicate. However, to enable it to be used in conjunction with other dynamic CLIC devices, its correctness statement has to be given in the same form. So to begin, here is the formal definition which captures the structure of the static inverter as shown in figure 3.3d.

$$\begin{aligned}\mathsf{Stat\_Inv}(ip, op) \quad =_{def} \quad & \exists p_0\ p_1\ p_2\ p_3. \\ & \mathsf{Gnd}(p_0) \quad \wedge \\ & \mathsf{Vdd}(p_1) \quad \wedge \\ & \mathsf{N\_Tran}(ip, p_0, p_2) \wedge \\ & \mathsf{P\_Tran}(ip, p_1, p_3) \wedge \\ & \mathsf{Join}(p_2, p_3, op)\end{aligned} \tag{3.37}$$

Now the usual three properties can be derived for this gate. The first one being that it's output is "Well Behaved."

$$\vdash \begin{pmatrix}\mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) \wedge \\ \mathsf{Stat\_Inv}(ip, op)\end{pmatrix} \supset \begin{pmatrix}(\mathsf{Wb}\ ip\ \overline{\phi}_1 \supset \mathsf{Wb}\ op\ \phi_1) \wedge \\ (\mathsf{Wb}\ ip\ \phi_1 \supset \mathsf{Wb}\ op\ \overline{\phi}_1) \wedge \\ (\mathsf{Wb}\ ip\ \overline{\phi}_2 \supset \mathsf{Wb}\ op\ \phi_2) \wedge \\ (\mathsf{Wb}\ ip\ \phi_2 \supset \mathsf{Wb}\ op\ \overline{\phi}_2)\end{pmatrix} \tag{3.38}$$

Inspecting this theorem (equation 3.38) reveals that it is in a different *form* as compared to the others presented so far. Infact it is not so different as to prevent logical inferences being made using the same techniques. However, if it were to be put in exactly the same form as the ones presented earlier, then four different clauses would result, giving rise to four theorems. Remember that the inverter is used to invert the *polarity* of a gate so that a gate may drive its own sort. E.g. a p-type gate may drive another p-type gate only if it is buffered by an inverter. Since there are two different sorts of gates, n-type and p-type, and two clock phases ϕ_1 and ϕ_2, the need arises for four very similar theorems, or one containing all four clauses.

The remaining two theorems for this device are fairly standard. Infact they are even simplified a little to take advantage of the fact that this device is not clocked. The next theorem (equation 3.39) for instance simply states that "if the input is defined then so is the output." Finally the last theorem for this device (equation 3.40) gives the logical behaviour appropriately abstracted to the boolean level using the Val_Abs predicate.

$$\vdash \begin{pmatrix}\mathsf{Stat\_Inv}(ip, op) \wedge \\ \mathsf{Def}\ ip\ t\end{pmatrix} \supset \mathsf{Def}\ op\ t \tag{3.39}$$

$$\vdash \begin{pmatrix}\mathsf{Stat\_Inv}(ip, op) \wedge \\ \mathsf{Def}\ ip\ t\end{pmatrix} \supset (\mathsf{Val\_Abs}\ op\ t\ =\ \sim\mathsf{Val\_Abs}\ ip\ t) \tag{3.40}$$

3.8 Deriving the Correctness Statements of CLIC Circuits

So far a technique has been outlined for deriving the correctness statement of arbitrary CLIC gates. This work would be entirely wasted if the old rules of thumb were then used for designing real circuits. If these correctness statements are to be actually used for the design of real circuits, rather than be used just for display due to their elegance, then formal means must be provided for doing so. I.e. formal method are needed for combining the correctness statements of an arbitrary number of gates, resulting in a new correctness statement for the new circuit.

All that a designer is interested in is obtaining the correct logical and temporal behaviour from the circuit. All the necessary rules which must be followed to achieve this end are infact just a distraction. So a technique is needed which allows the designer to compose together the behaviour component of the specifications and leave the rest of the "checking" to the system. What is proposed here is a system of simple logical inferences which are used to check the validity of connecting the output of one gate to the input of another. The inference rule involved is infact a variant of the classic Modus Ponens rule which can be stated as follows:

Modus Ponens:
$$\frac{\vdash \forall x.\ \mathsf{P}(x) \supset \mathsf{Q}(x) \qquad \vdash \mathsf{P}(y)}{\vdash \mathsf{Q}(y)}$$

Before showing how this inference rule is used to derive the correctness statements of CLIC circuits, a brief summary is first given of how the correctness statement of a gate driven by one phase of the clock results in an analogous correctness statement for the gate driven by the other phase of the clock. With this in place, the derivations of the correctness statements of CLIC circuits are then illustrated by a series of examples. Each of these examples is given to illustrate a single aspect of combining CLIC circuits.

3.8.1 Deriving ϕ_2 Correctness Statements from ϕ_1 Correctness Statements

The technique outlined above (section 3.7) for deriving the correctness statement of CLIC gates results in a correctness statement with the gate driven by ϕ_1 and/or $\overline{\phi}_1$. In any real circuit about half of the gates in it will be driven by ϕ_1 and $\overline{\phi}_1$,

and the other half will be driven by ϕ_2 and $\overline{\phi}_2$. So instead of maintaining two families of correctness statements, first for one phase of the clock and the second for the other phase of the clock, a technique is outlined here which allows the second correctness statement to be derived from the first.

This technique is illustrated through an example. Consider the correctness statements for the two input n-type nand gate, derived earlier as equations 3.25, 3.26, 3.31 and 3.32. All these correctness statements (apart from the first which does not need a time reference), have a general form which can be stated as follows:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_1, ip_1, ip_2, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \vdots & \end{pmatrix} \supset (\cdots\ t\ \cdots)$$

What needs to be changed is the clock phase by which the nand gate is driven, i.e. $\mathsf{nNand}_2(\overline{\phi}_1, \ \ldots\ )$ needs to be replaced by $\mathsf{nNand}_2(\overline{\phi}_2, \ \ldots\ )$. The general approach to achieving this is to generalise the free variables representing the clock lines, and then specialise them in a different order such that the clock phases are swapped. Doing this to the above correctness statement for the nand gate results in the following:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_2, \overline{\phi}_2, \phi_1, \overline{\phi}_1) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_2, ip_1, ip_2, op) & \wedge \\ \mathsf{isHi}\ \phi_2\ t & \wedge \\ \vdots & \end{pmatrix} \supset (\cdots\ t\ \cdots)$$

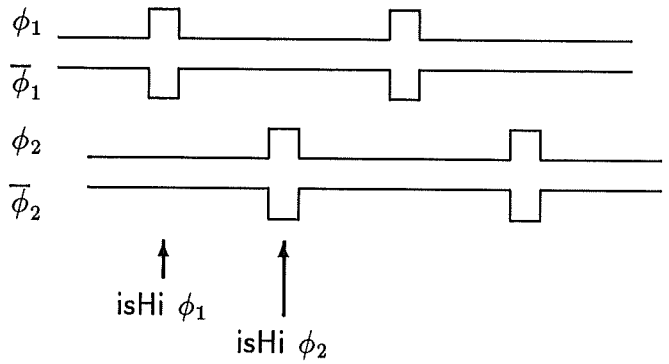
Unfortunately all that has been achieved here is a simple renaming of the variables; the basic relation between them remains unchanged. However, due to the way in which the clock was defined, the following lemma can be used:

$$\vdash \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) \supset \mathsf{Clock}(\phi_2, \overline{\phi}_2, \phi_1, \overline{\phi}_1) \tag{3.41}$$

This lemma says that it is not possible to tell the two phases apart from only the information available from the Clock predicate. Now by using this lemma, the above result for the nand gate can be modified further to the following:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_2, ip_1, ip_2, op) & \wedge \\ \mathsf{isHi}\ \phi_2\ t & \wedge \\ \vdots & \end{pmatrix} \supset (\cdots\ t\ \cdots)$$

This is better, but it is still not in the correct form. Note that the original equation used the time reference "isHi ϕ_1 t," i.e. when the ϕ_1 phase of the clock is Hi. However the derived equation here uses the time reference when ϕ_2 is Hi, i.e. "isHi ϕ_2 t." To clarify this consider a clock cycle showing both the time references on it as follows:



Due to the definition of Clock, the time difference between these two reference points is two units of time. This infact is a derived lemma about the clock and is used to resolve this time reference problem. The lemma can be stated as follows:

$$\vdash \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) \supset (\mathsf{isHi}\ \phi_2\ (t{+}2)\ =\ \mathsf{isHi}\ \phi_1\ t) \tag{3.42}$$

Now, by generalising the time variable (t) in the last derived equation for the nand gate, and specialising it to t+2 results in the term "isHi ϕ_2 (t+2)" being introduced into the equation. This is then changed to what is actually required, namely "isHi ϕ_1 t," by using the above stated clock lemma (equation 3.42). The modified equation for the nand gate now looks like the following:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_2, ip_1, ip_2, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \vdots & \end{pmatrix} \supset (\cdots\ (t{+}2)\ \cdots)$$

So the new set of correctness statements for the nand can now be stated in their full expanded form. These four theorems are:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_2, ip_1, ip_2, op) & \end{pmatrix} \supset \mathsf{Wb}\ op\ \phi_2$$

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_2, ip_1, ip_2, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Def}\ op\ (t{+}2) & \wedge \\ \mathsf{Def}\ op\ (t{+}3) & \wedge \\ \mathsf{Def}\ op\ (t{+}4) & \wedge \\ \mathsf{Def}\ op\ (t{+}5) & \end{pmatrix}$$

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_2, ip_1, ip_2, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ ip_1\ (t{+}3) & \wedge \\ \mathsf{Def}\ ip_2\ (t{+}3) & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Val\_Abs}\ op\ (t{+}3) & = \\ \sim \begin{pmatrix} \mathsf{Val\_Abs}\ ip_1\ (t{+}3) & \wedge \\ \mathsf{Val\_Abs}\ ip_2\ (t{+}3) & \end{pmatrix} & \end{pmatrix}$$

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nNand}_2(\overline{\phi}_2, ip_1, ip_2, op) & \wedge \\ \mathsf{Wb}\ ip_1\ \overline{\phi}_1 & \wedge \\ \mathsf{Wb}\ ip_2\ \overline{\phi}_1 & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ ip_1\ (t{+}3) & \wedge \\ \mathsf{Def}\ ip_2\ (t{+}3) & \wedge \\ \mathsf{Def}\ ip_1\ (t{+}4) & \wedge \\ \mathsf{Def}\ ip_2\ (t{+}4) & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Val\_Abs}\ op\ (t{+}4) & = \\ \sim \begin{pmatrix} \mathsf{Val\_Abs}\ ip_1\ (t{+}4) & \wedge \\ \mathsf{Val\_Abs}\ ip_2\ (t{+}4) & \end{pmatrix} & \end{pmatrix}$$

With this technique in place, the correctness statements for CLIC circuits can now be stated. All further correctness statements stated using either the ϕ_1 or the ϕ_2 clock phase are derived using this technique, and their derivations will not be presented. Furthermore, since the clock is cyclic over four units of time, any correctness statement which requires that the behaviour component be shifted by four units of time can be easily derived. This is justified by the following derived lemma about the Clock:

$$\vdash \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) \supset (\mathsf{isHi}\ \phi_1\ (t{+}4) = \mathsf{isHi}\ \phi_1\ t) \tag{3.43}$$

So any correctness statement that requires the behaviour to be shifted can be derived by generalising the time variable t, and then specialising it to $t{+}4$. Then, by using this clock lemma (equation 3.43), the original form of the correctness statement can be recovered.

3.8.2 Example: CLIC Gates Driven by the Same Clock Phase

In this example the inference rules used to check that a p-type gate can drive an n-type gate are illustrated. In particular, the mechanism for deriving the logical behaviour of the resultant circuit is illustrated. This technique will be used in the derivation of the correctness statement of future devices. The example used is the derivation of the correctness statement of an n-type exclusive-or gate. The circuit which implements this device is shown in figure 3.11.

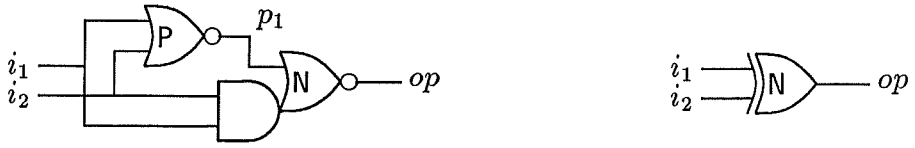


Figure 3.11: Implementation of an n-type exclusive-or gate in CLIC

This implementation is captured in formal notation by the following definition:

$$\begin{aligned} \mathsf{nXor}(\phi, \overline{\phi}, i_1, i_2, op) \quad =_{def} \quad & \exists p_1 . \\ & \mathsf{pNor}_2(\phi, i_1, i_2, p_1) \quad \wedge \\ & \mathsf{nAndNor}(\overline{\phi}, p_1, i_1, i_2, op) \end{aligned} \tag{3.44}$$

The correctness statements for the two internal devices used in this definition, namely the nAndNor and the pNor_2 gates, can be stated as the following two theorems, namely 3.45 and 3.46 respectively. These are stated here without derivation, but the procedure for deriving them has been described earlier in this chapter (in sections 3.7.1 and 3.8.1).

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nAndNor}(\overline{\phi}_1, p_1, i_1, i_2, op) & \wedge \\ \mathsf{Wb}\ p_1\ \overline{\phi}_1 & \wedge \\ \mathsf{Wb}\ i_1\ \overline{\phi}_1 & \wedge \\ \mathsf{Wb}\ i_2\ \overline{\phi}_1 & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ p_1\ (t{+}1) & \wedge \\ \mathsf{Def}\ i_1\ (t{+}1) & \wedge \\ \mathsf{Def}\ i_2\ (t{+}1) & \wedge \\ \mathsf{Def}\ p_1\ (t{+}2) & \wedge \\ \mathsf{Def}\ i_1\ (t{+}2) & \wedge \\ \mathsf{Def}\ i_2\ (t{+}2) & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Val\_Abs}\ op\ (t{+}2) \quad = \\ \sim \begin{pmatrix} \mathsf{Val\_Abs}\ p_1\ (t{+}2) \quad \vee \\ \begin{pmatrix} \mathsf{Val\_Abs}\ i_1\ (t{+}2) \ \wedge \\ \mathsf{Val\_Abs}\ i_2\ (t{+}2) \end{pmatrix} \end{pmatrix} \end{pmatrix} \tag{3.45}$$

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{pNor}_2(\phi_1, i_1, i_2, p_1) & \wedge \\ \mathsf{Wb}\ i_1\ \phi_1 & \wedge \\ \mathsf{Wb}\ i_2\ \phi_1 & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ i_1\ (t{+}1) & \wedge \\ \mathsf{Def}\ i_2\ (t{+}1) & \wedge \\ \mathsf{Def}\ i_1\ (t{+}2) & \wedge \\ \mathsf{Def}\ i_2\ (t{+}2) \end{pmatrix} \supset \begin{pmatrix} \mathsf{Wb}\ p_1\ \overline{\phi}_1 & \wedge \\ \begin{pmatrix} \mathsf{Def}\ p_1\ t & \wedge \\ \mathsf{Def}\ p_1\ (t{+}1) & \wedge \\ \mathsf{Def}\ p_1\ (t{+}2) & \wedge \\ \mathsf{Def}\ p_1\ (t{+}3) \end{pmatrix} & \wedge \\ \begin{pmatrix} \mathsf{Val\_Abs}\ p_1\ (t{+}2) & = \\ \sim \begin{pmatrix} \mathsf{Val\_Abs}\ i_1\ (t{+}2) & \vee \\ \mathsf{Val\_Abs}\ i_2\ (t{+}2) \end{pmatrix} \end{pmatrix} & \end{pmatrix} \tag{3.46}$$

Equation 3.46 states that under certain conditions, the line p_1 is "Well Behaved," and "Well Defined" over certain times. These exact conditions on the line p_1 also form the assumptions for equation 3.45. So by eliminating the assumptions of one by the conclusions of the other (by Modus Ponens), these two implications can be combined to give the following intermediate result:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{pNor}_2(\phi_1, i_1, i_2, p_1) & \wedge \\ \mathsf{nAndNor}(\overline{\phi}_1, p_1, i_1, i_2, op) & \wedge \\ \mathsf{Wb}\ i_1\ \phi_1\ \wedge\ \mathsf{Wb}\ i_1\ \overline{\phi}_1 & \wedge \\ \mathsf{Wb}\ i_2\ \phi_1\ \wedge\ \mathsf{Wb}\ i_2\ \overline{\phi}_1 & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ i_1\ (t{+}1) & \wedge \\ \mathsf{Def}\ i_2\ (t{+}1) & \wedge \\ \mathsf{Def}\ i_1\ (t{+}2) & \wedge \\ \mathsf{Def}\ i_2\ (t{+}2) \end{pmatrix} \supset \begin{pmatrix} \begin{pmatrix} \mathsf{Val\_Abs}\ p_1\ (t{+}2) & = \\ \sim \begin{pmatrix} \mathsf{Val\_Abs}\ i_1\ (t{+}2) & \vee \\ \mathsf{Val\_Abs}\ i_2\ (t{+}2) \end{pmatrix} \end{pmatrix} \\ \wedge \\ \begin{pmatrix} \mathsf{Val\_Abs}\ op\ (t{+}2) & = \\ \sim \begin{pmatrix} \mathsf{Val\_Abs}\ p_1\ (t{+}2) & \vee \\ \begin{pmatrix} \mathsf{Val\_Abs}\ i_1\ (t{+}2) & \wedge \\ \mathsf{Val\_Abs}\ i_2\ (t{+}2) \end{pmatrix} \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

By rewriting from the first conjunct on the right hand side of the implication to the second conjunct, the term "Val_Abs p_1 $(t{+}2)$" in the above equation can be eliminated. This results in the following:

$$\vdash (\cdots) \supset \begin{pmatrix} \mathsf{Val\_Abs}\ op\ (t{+}2) & = \\ \sim \begin{pmatrix} \sim \begin{pmatrix} \mathsf{Val\_Abs}\ i_1\ (t{+}2) & \vee \\ \mathsf{Val\_Abs}\ i_2\ (t{+}2) \end{pmatrix} & \vee \\ \begin{pmatrix} \mathsf{Val\_Abs}\ i_1\ (t{+}2) & \wedge \\ \mathsf{Val\_Abs}\ i_2\ (t{+}3) \end{pmatrix} & \end{pmatrix} \end{pmatrix}$$

Now by simple logical manipulation, the right hand side of this implication can be reduced to the exclusive-or function. So restating the above result with this simplification applied, gives:

$$\vdash\ (\cdots) \supset \left(\begin{array}{l} \mathsf{Val\_Abs}\ op\ (t{+}2)\ = \\ \quad \left(\begin{array}{l} \mathsf{Val\_Abs}\ i_1\ (t{+}2)\ \oplus \\ \mathsf{Val\_Abs}\ i_2\ (t{+}2) \end{array}\right) \end{array}\right)$$

This is almost in the final form for the correctness statement of the n-type exclusive-or gate. The problem is that the line p_1 still occurs on the right hand side of this implication. The following rule is used to hide this line, which existentially binds the line p_1 around only those places where it occurs.

Exists Intro: $\dfrac{\vdash\ \mathsf{tm}[x]}{\vdash\ \exists x.\ \mathsf{tm}[x]}$

Using this rule would result in the term "$\mathsf{pNor}_2(\ldots, p_1, \ldots) \wedge \mathsf{nAndNor}(\ldots, p_1, \ldots)$" being changed to "$\exists p_1.\ \mathsf{pNor}_2(\ldots, p_1, \ldots) \wedge \mathsf{nAndNor}(\ldots, p_1, \ldots)$". Now since this new term is identical to the definition of the predicate nXor as defined in equation 3.44, the final form of the correctness statement is derived by rewriting this term by $\mathsf{nXor}(\ldots)$. So now the final derived correctness statement for the n-type exclusive-or gate can be stated as follows:

$$\vdash \left(\begin{array}{ll} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{nXor}(\phi_1, \overline{\phi}_1, i_1, i_2, op) & \wedge \\ \mathsf{Wb}\ i_1\ \phi_1\ \wedge\ \mathsf{Wb}\ i_1\ \overline{\phi}_1 & \wedge \\ \mathsf{Wb}\ i_2\ \phi_1\ \wedge\ \mathsf{Wb}\ i_2\ \overline{\phi}_1 & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ i_1\ (t{+}1) & \wedge \\ \mathsf{Def}\ i_2\ (t{+}1) & \wedge \\ \mathsf{Def}\ i_1\ (t{+}2) & \wedge \\ \mathsf{Def}\ i_2\ (t{+}2) & \end{array}\right) \supset \left(\begin{array}{l} \mathsf{Val\_Abs}\ op\ (t{+}2)\ = \\ \quad \left(\begin{array}{l} \mathsf{Val\_Abs}\ i_1\ (t{+}2)\ \oplus \\ \mathsf{Val\_Abs}\ i_2\ (t{+}2) \end{array}\right) \end{array}\right) \qquad (3.47)$$

There are two points which must be mentioned before progressing further. Firstly, the final form of the correctness statement derived above (equation 3.47) is in no way different from any of the correctness statements that could be derived for the CLIC primitive gates. By this, it is meant that the correctness statement derived above has the following aspects, which are common to all correctness statements for CLIC gates and circuits.

- The gate is driven by a well behaved clock, namely Clock.
- The time reference used for stating the behaviour is isHi ϕ_1.

- The inputs have "Well Behaved" and "Well Defined" conditions.

This means that in formal analysis an arbitrary cluster of CLIC gates can be treated just as a single gate is treated.

Secondly, this correctness statement requires that the inputs be "Well Behaved" with respect to both ϕ_1 and $\overline{\phi}_1$. This is because these inputs feed both n-type and p-type gates internally to the exclusive-or gate. The only device which satisfies this behaviour is the latch. So from this it follows that the exclusive-or gate can only be driven by either a latch, or a latch buffered by a static inverter.

3.8.3 Example: CLIC Gates Driven by Different Clock Phases

This example illustrates how a correctness statement can be derived for gates driven by different phases of the clock. The example used is the design of a dynamic register cell in CLIC. This is built using two latches, one driven by ϕ_1 and $\overline{\phi}_1$, and the other driven by ϕ_2 and $\overline{\phi}_2$. The circuit for this is shown in figure 3.12.

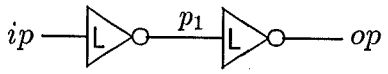


Figure 3.12: Implementing a dynamic register cell in CLIC

This implementation is captured in formal notation by the following definition:

$$\begin{aligned} \mathsf{Reg}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, ip, op) \quad =_{def} \quad & \exists p_1. \\ & \mathsf{Latch}(\phi_1, \overline{\phi}_1, ip, p_1) \wedge \\ & \mathsf{Latch}(\phi_2, \overline{\phi}_2, p_1, op) \end{aligned} \tag{3.48}$$

The correctness statements for the two latches used in this implementation can be stated as the two theorems 3.49 and 3.50. The first of these theorems is identical to the derived result as stated earlier as theorem 3.36. The second theorem (3.50), however, is derived from equation 3.36 by the technique outlined in section 3.8.1. This results in a correctness statement in which the Latch device is driven by the ϕ_2 and $\overline{\phi}_2$ clock phases.

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) \wedge \\ \mathsf{Latch}(\phi_1, \overline{\phi}_1, ip, p_1) \wedge \\ \mathsf{isHi}\ \phi_1\ t \wedge \\ \mathsf{Def}\ ip\ t \end{pmatrix} \supset \begin{pmatrix} \mathsf{Val\_Abs}\ p_1\ \ t\ \ \ \ \ \ = \sim \mathsf{Val\_Abs}\ ip\ t \wedge \\ \mathsf{Val\_Abs}\ p_1\ (t{+}1) = \sim \mathsf{Val\_Abs}\ ip\ t \wedge \\ \mathsf{Val\_Abs}\ p_1\ (t{+}2) = \sim \mathsf{Val\_Abs}\ ip\ t \wedge \\ \mathsf{Val\_Abs}\ p_1\ (t{+}3) = \sim \mathsf{Val\_Abs}\ ip\ t \end{pmatrix} \tag{3.49}$$

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{Latch}(\phi_2, \overline{\phi}_2, p_1, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ p_1\ (t{+}2) & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Val\_Abs}\ op\ (t{+}2) = {\sim}\mathsf{Val\_Abs}\ p_1\ (t{+}2) & \wedge \\ \mathsf{Val\_Abs}\ op\ (t{+}3) = {\sim}\mathsf{Val\_Abs}\ p_1\ (t{+}2) & \wedge \\ \mathsf{Val\_Abs}\ op\ (t{+}4) = {\sim}\mathsf{Val\_Abs}\ p_1\ (t{+}2) & \wedge \\ \mathsf{Val\_Abs}\ op\ (t{+}5) = {\sim}\mathsf{Val\_Abs}\ p_1\ (t{+}2) & \end{pmatrix} \tag{3.50}$$

These two correctness statements can now be combined. The derivation procedure is relatively straightforward and is identical to that used in the previous section. This is not repeated here. The final derived result is:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{Reg}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, ip, op) & \wedge \\ \mathsf{isHi}\ \phi_1\ t & \wedge \\ \mathsf{Def}\ ip\ t & \end{pmatrix} \supset \begin{pmatrix} \mathsf{Val\_Abs}\ op\ (t{+}2) = \mathsf{Val\_Abs}\ ip\ t & \wedge \\ \mathsf{Val\_Abs}\ op\ (t{+}3) = \mathsf{Val\_Abs}\ ip\ t & \wedge \\ \mathsf{Val\_Abs}\ op\ (t{+}4) = \mathsf{Val\_Abs}\ ip\ t & \wedge \\ \mathsf{Val\_Abs}\ op\ (t{+}5) = \mathsf{Val\_Abs}\ ip\ t & \end{pmatrix} \tag{3.51}$$

This derived result doesn't quite capture the conceptual behaviour of a dynamic register. What is generally understood as a register device is one which exhibits a unit delay type of behaviour. This derived result could however be moulded to show that it does have the same behaviour. This is done by abstracting the signals on the input and the output lines from a fine grain of time to a coarse grain of time. So rather than having four ticks of time per clock cycle, only one tick of time per clock cycle is used. This technique of going from one granularity of time to another is known as "temporal abstraction[2]." The most useful predicate that needs to be understood to follow the correctness statements given below is the when predicate. Informally this can be defined as follows:

[2]The ideas in this direction were first discussed in the weekly Hardware Verification Research Group meetings. In 1984 Mike Gordon presented a technique by which multiple states at one level of time could be amalgamated into a single state at the higher more abstract level. Based on this idea, Tom Melham [Melham 88a] developed the present (most elegant) set of predicates which are adopted in this thesis. John Herbert also uses some of these ideas in his thesis [Herbert 86] in showing the correctness of one of the parts of the Cambridge Fast Ring. The predicates used in this thesis have all been defined by the author and may not follow exactly the same line of development as that used by Melham. Note also that the techniques involved in applying these ideas to the two phase clocking scheme are all new.

$(sig$ when $ctl)\ n$ = The value on the wire sig at the time when the control signal ctl is true for the nth time.

In formal notation this when predicate is defined by the following four definitions (see [Melham 88a] for a more thorough discussion on how to implement this in the HOL system):

$$\begin{aligned}\mathsf{Next}\ t_1\ t_2\ f\ =_{def}\ & (t_1 < t_2)\ \wedge \\ & (f\ t_2)\ \wedge \\ & \forall t.\ t_1{<}t{<}t_2\ \supset\ \sim(f\ t)\end{aligned}$$

$$\mathsf{IsTimeOf}\ \ 0\ \ f\ t\ =_{def}\ (f\ t)\ \wedge\ \forall t'.\ (t'{<}t) \supset \sim(f\ t')$$

$$\mathsf{IsTimeOf}\ (n{+}1)\ f\ t\ =_{def}\ \exists t'.\ \mathsf{IsTimeOf}\ n\ f\ t'\ \wedge\ \mathsf{Next}\ t'\ t\ f$$

$$\mathsf{TimeOf}\ f\ n\ =_{def}\ \varepsilon\, t.\ \mathsf{IsTimeOf}\ n\ f\ t$$

$$sig\ \mathsf{when}\ ctl\ =_{def}\ \lambda n.\ sig(\mathsf{TimeOf}\ ctl\ n)$$

The first three of these predicates can now be informally described in the same way that the when predicate was described above.

Next $t_1\ t_2\ f$ = A relation stating that the next time after t_1 when f is true is at time t_2.

IsTimeOf $n\ f\ t$ = A relation stating that the nth time that f is true is at time t.

TimeOf $f\ n$ = The time of the event when f is true for the nth time.

From these definition, two of the most useful lemmas about the two phase clock can be derived. These are formally stated as equations 3.52 and 3.53. The first one captures the fact that it is always the case that the clock phase ϕ_1 is Hi exactly at the times when ϕ_1 is Hi. This is rather circular but is infact quite a useful lemma. The second lemma (equation 3.53) captures the notion that the clock phase ϕ_1 is Hi every four units of time.

$$\vdash\ \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2)\ \supset\ \forall n.\ (\mathsf{isHi}\ \phi_1)(\mathsf{TimeOf}\ (\mathsf{isHi}\ \phi_1)\ n) \tag{3.52}$$

$$\begin{aligned}\vdash\ & \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2)\ \supset \\ & \forall n.\ \mathsf{TimeOf}\ (\mathsf{isHi}\ \phi_1)\ (n{+}1)\ =\ (\mathsf{TimeOf}\ (\mathsf{isHi}\ \phi_1)\ n) + 4\end{aligned} \tag{3.53}$$

With this more abstract view of time and by using these derived lemmas about clock (equations 3.52 and 3.53), the behaviour of the register device begins to look more like what is required. The final derived theorem for this device can now be stated as follows:

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{Reg}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, ip, op) & \wedge \\ \forall n.\ ((\mathsf{Def}\ ip)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1))\ n \end{pmatrix} \supset$$
$$\begin{pmatrix} \textbf{let}\ ip_{\phi_1} = (\mathsf{Val\_Abs}\ ip)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\ \textbf{in} \\ \textbf{let}\ op_{\phi_1} = (\mathsf{Val\_Abs}\ op)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\ \textbf{in} \\ \forall t.\ op_{\phi_1}(t{+}1) = ip_{\phi_1}(t) \end{pmatrix} \tag{3.54}$$

This theorem (equation 3.54) can be stated in English terms as follows:

Assuming a correct relation exists between the clock lines,
and a correct implementation of the Reg device exists,
and the value on the input line is always defined when the value on the clock line ϕ_1 is Hi,
then the temporally abstracted value on the output is equal to the temporally abstracted value on the input delayed by one,
where a signal which is temporally abstracted means that the value on it is sampled every time that the clock line ϕ_1 is Hi.

There are two main points to note about this final derived result; firstly, all the various ideas present in it are separated into custom designed predicates, and secondly, these predicates nicely *fit* together to allow this final correctness statement to be derived.

3.8.4 Example: CLIC Circuits with Feedback

This example illustrates how CLIC circuits with feedback are handled. It is infact just a more elaborate version of the previous example. This time no derivations are presented since the procedure used is almost identical to that of the previous two cases. The example is an implementation of a toggle flip-flop cell in CLIC[3]. This device has a single input and a single output. On every clock cycle, depending on the state of the input at certain times, the output of the device is inverted if the input is Hi, and left unchanged if it is Lo. The circuit implementing this device is shown in figure 3.13.

[3]The dependency of the theories necessary for this example are shown as a tree in appendix A, and the detailed ML code for the proof of correctness of the toggle flip-flop is given in appendix B.

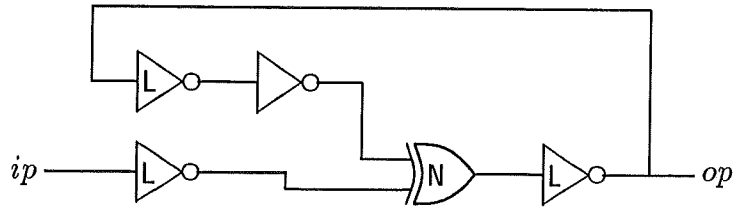


Figure 3.13: Implementing a Toggle flip-flop in CLIC

This implementation is captured in formal notation by the following definition:

$$\begin{aligned}\mathsf{Toggle}(\phi_1,\overline{\phi}_1,\phi_2,\overline{\phi}_2,ip,op) \;=_{def}\; & \exists p_1\ p_2\ p_3\ p_4.\\ & \mathsf{Latch}(\phi_1,\overline{\phi}_1,ip,p_1) \;\wedge\\ & \mathsf{Latch}(\phi_2,\overline{\phi}_2,op,p_2) \;\wedge\\ & \mathsf{Stat\_Inv}(p_2,p_3) \;\wedge\\ & \mathsf{nXor}(\phi_1,\overline{\phi}_1,p_1,p_3,p_4) \;\wedge\\ & \mathsf{Latch}(\phi_2,\overline{\phi}_2,p_4,op)\end{aligned} \tag{3.55}$$

The final derived correctness statement for this device is shown in equation 3.56. Note the similarity between this correctness statement and that of the previous example; they are almost identical except for the one extra condition. This condition, represented in this derived theorem as the line "((Def op) when (isHi ϕ_1)) 0," states that the value on the output(op) must be "Well Defined" at the point in time when the line ϕ_1 is Hi for the first time. This is necessary since the output line is used for feedback, and so the value on it must be defined before it can be used. It is important to note that this condition dictates the value on the output only for the first time. Furthermore, it does not require the output node to be set to a particular value, only that it be "Well Defined" so no errors can occur. For all subsequent times, the circuit dictates what the value on the output should be, so this is not required as an assumption in this correctness statement.

$$\begin{aligned}\vdash & \begin{pmatrix}\mathsf{Clock}(\phi_1,\overline{\phi}_1,\phi_2,\overline{\phi}_2) & \wedge\\ \mathsf{Toggle}(\phi_1,\overline{\phi}_1,\phi_2,\overline{\phi}_2,ip,op) & \wedge\\ ((\mathsf{Def}\ op)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1))\ 0 & \wedge\\ \forall n.\ ((\mathsf{Def}\ ip)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1))\ n\end{pmatrix} \supset\\ & \begin{pmatrix}\textbf{let}\ ip_{\phi_1} = (\mathsf{Val\_Abs}\ ip)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\\ \textbf{in}\\ \textbf{let}\ op_{\phi_1} = (\mathsf{Val\_Abs}\ op)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\\ \textbf{in}\\ \forall t.\ op_{\phi_1}(t{+}1) = (ip_{\phi_1}\ t) \Rightarrow \sim(op_{\phi_1}\ t)\ |\ (op_{\phi_1}\ t)\end{pmatrix}\end{aligned} \tag{3.56}$$

3.8.5 Example: Using Higher Level CLIC Building Blocks

The previous three examples have all illustrated how the correctness statements are derived for particular CLIC circuits design techniques. In this example higher level CLIC building blocks are used, such as the circuits used in the previous examples, to demonstrate that the correctness statements of these higher level building blocks can be used to derive the next higher level correctness statement, just as the correctness statements for the previous examples are derived from those of the CLIC primitive gates. The example used is infact a dynamic shift register. This is trivially built by connecting many Reg device together in a chain. The correctness statement for the Reg device has already been derived above in section 3.8.3. The circuit which implements this dynamic shift register is shown in figure 3.14

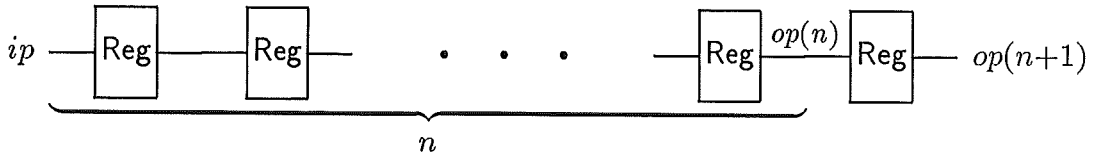


Figure 3.14: Implementing a Dynamic Shift Register in CLIC

This implementation is captured in formal notation by the following definition:

$$\begin{array}{lll} \mathsf{ShiftReg} & 0 & \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ ip\ op \ =_{def}\ (op(0) = ip) \\ \mathsf{ShiftReg} & (n{+}1) & \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ ip\ op \ =_{def} \\ & & \quad \mathsf{ShiftReg}\ n\ \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ ip\ op(n)\ \wedge \\ & & \quad \mathsf{Reg}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, op(n), op(n{+}1)) \end{array} \tag{3.57}$$

The derived correctness statement for this device is shown below as equation 3.58. Note the similarity between this theorem and that for the Reg device derived earlier as equation 3.54.

$$\vdash \begin{pmatrix} \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{ShiftReg}\ n\ \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ ip\ op & \wedge \\ \forall t.\ ((\mathsf{Def}\ ip)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1))\ t & \wedge \\ \forall m.\ m{<}n \supset ((\mathsf{Def}\ (op\ m))\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1))\ 0 \end{pmatrix} \supset \begin{pmatrix} \mathbf{let}\ ip_{\phi_1} = (\mathsf{Val\_Abs}\ ip)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\ \mathbf{in} \\ \mathbf{let}\ op_{\phi_1} = (\mathsf{Val\_Abs}\ op)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\ \mathbf{in} \\ \forall t.\ op_{\phi_1}(t{+}n) = ip_{\phi_1}(t) \end{pmatrix} \tag{3.58}$$

This theorem has the assumption that the input(ip) must be "Well Defined" every time it is sampled with ϕ_1 being Hi. This is as expected since the Reg device too has this assumption. But this theorem also has an additional assumption which states that the inputs of all the Reg devices must be "Well Defined" when the ϕ_1 line is Hi for the first time. This is necessary so that only defined values propagate down the chain. If errors were allowed to propagate down the chain then the values on the output nodes could not be abstracted to the boolean domain. Since the previously derived results are stated only in terms of signals abstracted to the boolean level, it is necessary that this assumption also be present here otherwise the previous results could not be used.

Finally, note that this extra condition assumes the nodes only to be "Well Defined," and not given any particular value. Furthermore, this is only necessary for the first time that ϕ_1 is Hi, and thereafter the circuit itself can satisfy the various constraints of the building blocks.

3.9 Summary

In this chapter a technique has been presented for formalising a dynamic integrated circuit design style in higher-order logic. The design style used as the basis of this work is known as the CLIC design style. First an informal description of this design style was given to highlight the various aspects which need to be formalised. The two phase clocking scheme used, and the charge storage problems of dynamic gates were then discussed. Finally a complete list of the rules for designing integrated circuits using this design style was compiled. The remainder of this chapter was devoted to the formalisation of these rules in logic.

In formalising the CLIC design style, first the algebra of signals on wires was axiomatised, then the six primitive devices of the CMOS technology were defined; these being, power, ground, n-type and p-type transistors, a capacitor for charge storage, and a join device for generating wired-or junctions. Note that all the building blocks used in the CLIC integrated circuit design style were designed using only these six primitive devices. For example, capacitors which have a decay time of greater than one unit of time can be built by joining together many unit-decay-time capacitors. Infact a formal proof of correctness of a capacitor with decay time n has been done, and the final statement of correctness was presented in this chapter.

The two phased clocking scheme was then formalised as the predicate Clock. This captures exactly the relation between the various clock lines, without impos-

ing extra conditions on them such as their values at time 0. This is an important aspect of the formalisation of the clock. In general if one is presented with the two phases of a clock, all that can be discerned is the relation between the two, but it is impossible to say which is ϕ_1 and which is ϕ_2. Infact calling one of the clock lines ϕ_1 is merely a label so that one can refer to it later as necessary. Indeed, the symmetry of the clock between its two phases is an important aspect which designers make use of all the time. For example a dynamic gate driven by one phase of the clock is identical in every respect to an equivalent gate driven by the other phase of the clock, except that the behaviour is also shifted as the clock phases are shifted. In this chapter, this particular aspect of the clock was also put to good use. A technique was presented which enabled the behaviour of gates driven by the ϕ_2 and $\overline{\phi}_2$ phases of the clock, to be derived from equivalent gates driven by ϕ_1 and $\overline{\phi}_1$ phases of the clock.

Having dealt with the preliminaries, the specific rules of the design style were then formalised in logic. An important predicate defined to aid in this respect was the "Well Behaved (Wb)" predicate. This captures the constraints that must be imposed on the inputs of CLIC gates to ensure that they operate correctly, and also expresses the behaviour of well-defined outputs. With the aid of this predicate Wb, a method was presented to derive the correctness statements for the entire class of CLIC gates. This method relies on the fact that the form of the correctness statement developed is uniform across the entire range, from primitive gates to large and complex circuits.

The use of these formal techniques were then demonstrated through a number of worked examples, each illustrating a different aspects of combining CLIC circuits. In each case the form of the derived correctness statement was shown to be identical. Various different abstraction techniques [Melham 88a] were also illustrated through these examples. These include representing the behaviour of circuits at different granularities of time, and using different data types.

Chapter 4

Formulating the Correctness of a Random Walk Filter

*In this chapter the statement of correctness for a Random Walk Filter (*RWF*) is formulated. The* RWF *is designed in the* CLIC *design style. Integers are used in the specification of this device since they help represent the functionality in a more natural way. This does, however, complicate the proof considerably at the higher level which is not covered in this chapter. This chapter focuses on deriving the correctness statement from a cluster of* CLIC *gates using the formal techniques of the previous chapter.*

4.1 Introduction

The Random Walk Filter (RWF) is a device used in applications where it is necessary to keep an average of a sequence of two opposing events. An example of this might be keeping a track of the number of heads over tails and vice versa in tossing a coin. In fact this device was specifically designed to provide a correction signal in a digital phase-locked loop [Cessna 72]. Here, a binary phase detector is used which gives an indication at each cycle whether the phase of the incoming signal is leading or lagging the locally generated clock. These lead/lag signals are then "averaged" using a RWF to provide a correction signal to the loop.

The RWF device can be viewed as an up/down counter which resets itself to the center position in the count range when it overflows. In fact it is more natural to view it as an up/down counter over the integers which resets itself to the zero

position if it overflows in the positive or the negative direction. This is a more natural view since the events being counted can be seen to be adding one or subtracting one from the internal state of the device. So when the internal state reaches a predetermined value, positive or negative, then the system resets itself to zero and begins again. Figure 4.1 shows a schematic view of the random walk filter from the top level with two inputs and two outputs as originally proposed by Cessna and Levy [Cessna 72]. The or-gate here is used to reset the counter back to the zero position when an output occurs on either the $+N$ or the $-N$ output ports. Note how this schematic exactly mimics the informal description given above.

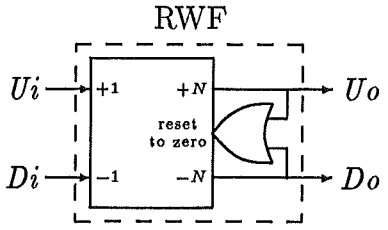


Figure 4.1: Schematic view of an implementation of the RWF

In the remainder of this chapter a specification and an implementation of this device are formally stated in higher-order logic. The implementation chosen is different to the above schematic but has identical top level behaviour and goes all the way down to the CLIC gate level. A formal proof of correctness is then outlined showing how the implementation chosen actually meets the formal specification. This proof has been completed in the HOL system and is not given here fully; only some of the more interesting parts are presented. In particular a lot of the details of the top level part of the proof are left out, since they only distract attention from the main point of this chapter, namely to show that complex devices designed in the CLIC design style can be formally analysed with the techniques outlined in the previous chapter.

4.2 Formal Specification

As a first stab at writing the specification, consider a device which is very similar to the RWF as described above, but is a little more "general." In this case the device has only one input and one output. The input, instead of being sequences of pairs of booleans, is a sequence of integers. The output similarly is a sequence of integers. This new "more general" device has an internal state which can count up to n. The top level view of this is illustrated in figure 4.2, as a device with a single input (ip), a single output (op), an internal state (S), and a parameter n which limits

the value of the internal state so that it can only be incremented/decremented to $\pm n$.

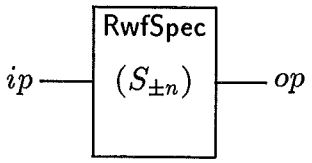


Figure 4.2: RwfSpec — An Up/Down Accumulator with overflow

Before constructing a formal specification for this device, note that it does not match the behaviour of a RWF device as described earlier. The input and the output ports of this device can take any integer values according to what has been stated above. But in the case of the actual RWF device, the signals on the input and the output ports can only be one of three possible values. These values in their abstract form in the above context can be either a $+1$, a -1, or a 0, but not any integer. So to take this into account in the specification, a new type $\mathcal{I}_{|2|}$ is declared for the input and the output signals.

This new type has three values, namely $\mathcal{N}$, $\mathcal{U}$, and $\mathcal{D}$. These stand for, "do Nothing," "change(ed) Up by one," and "change(ed) Down by one" respectively. The axiomatisation of this new type in the HOL system is a fairly tedious, but a logically straight forward task. Having declared it as a new type, the usual theorems enumerating this type and stating the that the objects of this type are distinct can be derived. These derived theorems are stated here respectively as follows:

$$\vdash \forall x.\ x = \mathcal{U} \ \vee\ x = \mathcal{N} \ \vee\ x = \mathcal{D}$$

$$\vdash \mathcal{U} \neq \mathcal{D} \ \wedge\ \mathcal{U} \neq \mathcal{N} \ \wedge\ \mathcal{N} \neq \mathcal{D}$$

The representing type used for this new type is in fact the pairs of booleans. With this new type definition two new functions are declared. These two functions map objects of the new type, to its representing type, and back again. The relationship between these two types is shown figure 4.3 with the two functions named as "$\mathsf{bbto}\mathcal{I}_{|2|} : \mathsf{bool} \times \mathsf{bool} \rightarrow \mathcal{I}_{|2|}$" and "$\mathcal{I}_{|2|}\mathsf{tobb} : \mathcal{I}_{|2|} \rightarrow \mathsf{bool} \times \mathsf{bool}$" respectively. Formally these mapping functions are defined as the two equations 4.1 and 4.2.

$$\begin{aligned}
\mathsf{bbto}\mathcal{I}_{|2|}(\mathsf{T},\mathsf{F}) &=_{def} \mathcal{U} \\
\mathsf{bbto}\mathcal{I}_{|2|}(\mathsf{T},\mathsf{T}) &=_{def} \mathcal{N} \\
\mathsf{bbto}\mathcal{I}_{|2|}(\mathsf{F},\mathsf{T}) &=_{def} \mathcal{D} \\
\mathsf{bbto}\mathcal{I}_{|2|}(\mathsf{F},\mathsf{F}) &=_{def} \mathcal{N}
\end{aligned} \tag{4.1}$$

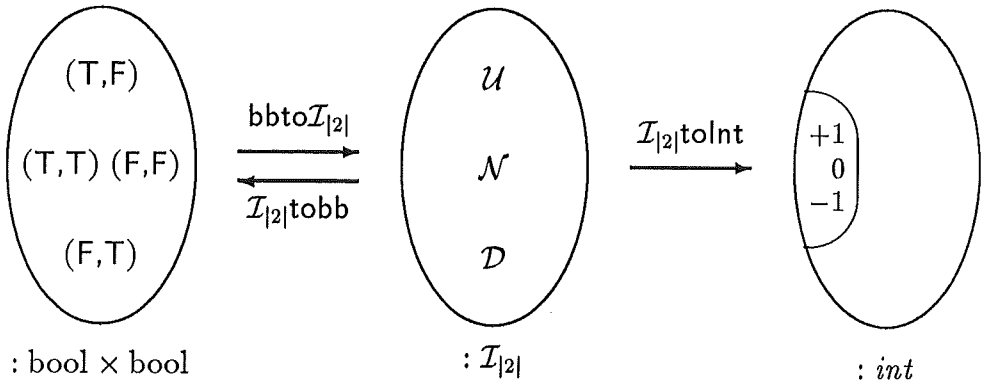


Figure 4.3: Relationship between the new type $\mathcal{I}_{|2|}$ and others

$$\begin{aligned}
\mathcal{I}_{|2|}\mathsf{tobb}\ \mathcal{U} &=_{def} (\mathsf{T},\mathsf{F}) \\
\mathcal{I}_{|2|}\mathsf{tobb}\ \mathcal{D} &=_{def} (\mathsf{F},\mathsf{T}) \\
\mathcal{I}_{|2|}\mathsf{tobb}\ \mathcal{N} &=_{def} (\mathsf{T},\mathsf{T})
\end{aligned} \tag{4.2}$$

From these definitions note that the function $\mathsf{bbto}\mathcal{I}_{|2|}$ is not equal to the inverse of the function $\mathcal{I}_{|2|}\mathsf{tobb}$. This can be stated in more formal terms as the theorem: "$\vdash \mathsf{bbto}\mathcal{I}_{|2|} \circ \mathcal{I}_{|2|}\mathsf{tobb} \neq \mathsf{I}$," where I is the identity function. The reason for this inequality is due to the fact that the function $\mathsf{bbto}\mathcal{I}_{|2|}$ is defined in such a way that it maps (F,F) to $\mathcal{N}$. If the clause $(\mathsf{bbto}\mathcal{I}_{|2|}(\mathsf{F},\mathsf{F}) =_{def} \mathcal{N})$ was missed out from the definition of this function, then the above inequality would hold. The reason for defining this type conversion (abstraction) function in this way will become clear in the proof of correctness for the RWF device.

Assuming the input and the output of the RwfSpec device are now a sequence of values of this new type ($\mathcal{I}_{|2|}$), then the behaviour for this device can be formally stated as equation 4.3. Note that this definition makes use of another type conversion (abstraction) function $\mathcal{I}_{|2|}\mathsf{toInt}$, which is also defined below, and the graphical mapping of it is illustrated in figure 4.3.

$$\begin{aligned}
&\mathsf{RwfSpec}\ n\ ip\ op\ S\ =_{def} \\
&\quad S(0) = 0\ \wedge \\
&\quad \textbf{let}\ ip_{int} = \lambda t.\ \mathcal{I}_{|2|}\mathsf{toInt}\ ip(t) \\
&\quad \textbf{in} \\
&\quad \forall t.\ |ip_{int}(t) + S(t)| < 2^{n+1} \rightarrow \\
&\qquad op(t) = \mathcal{N} \quad \wedge\ S(t{+}1) = ip_{int}(t) + S(t)\ \ | \\
&\qquad op(t) = ip(t)\ \wedge\ S(t{+}1) = 0
\end{aligned} \tag{4.3}$$

where $\mathcal{I}_{|2|}\mathsf{toInt}\ \mathcal{U} =_{def} +1$, $\mathcal{I}_{|2|}\mathsf{toInt}\ \mathcal{D} =_{def} -1$, $\mathcal{I}_{|2|}\mathsf{toInt}\ \mathcal{N} =_{def} 0$.

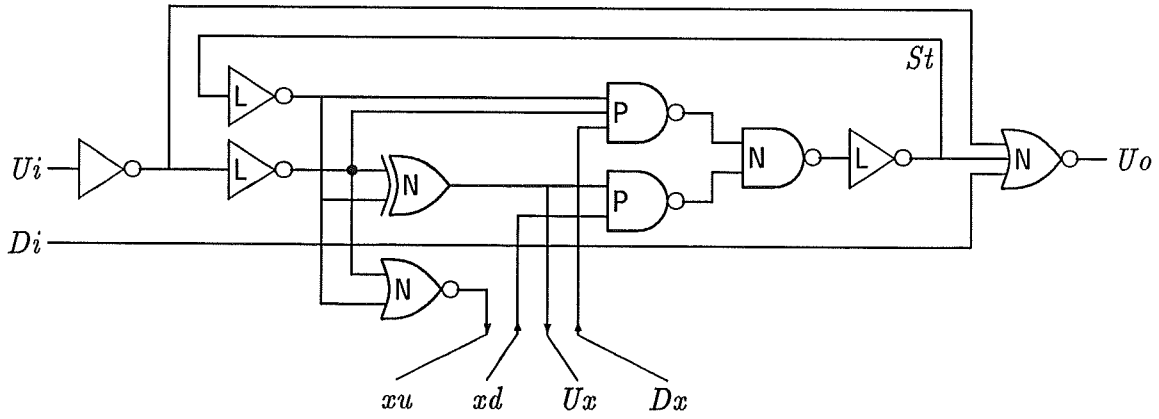
This looks like a rather complex specification, but it is defined in this way because it captures all the various aspects that are necessary. Examining it a little more closely reveals that the value of the internal state S at time zero is 0. Also, since the input and the output are now defined to have type $\mathcal{I}_{|2|}$, their abstracted value at the integer level can also be only one of the three values; $+1$, -1, or 0. This specification states that if adding the integer value of the input to the internal state causes it to overflow in either direction, then the output is set to the same value as the input (which indicates the direction of the overflow), and the value of the internal state is reset to 0. However if the integer value of the input added to the internal state does not cause it to overflow, then the output is set to indicate no-overflow($\mathcal{N}$), and the internal state is changed by adding the integer value of the input to it.

An interesting lemma that can be derived from this definition is shown in equation 4.4. This states that the RWF device behaves a bit like an adder; i.e., the sum of the input and the internal state is equal to the sum of the new state plus 2^{n+1} times the output. So the output of this device behaves a bit like the carry out of an adder.

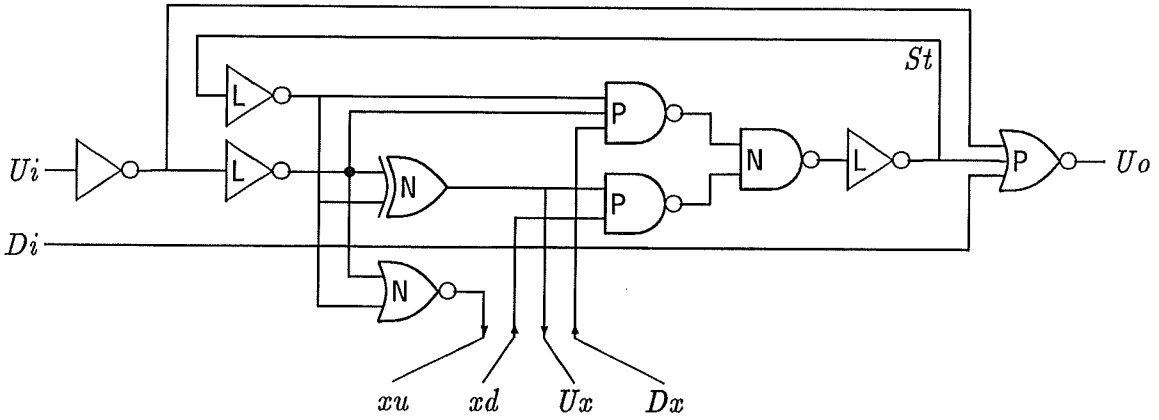
$$
\begin{array}{l}
\vdash \mathsf{RwfSpec}\ n\ ip\ op\ S\ \supset \\
\qquad \textbf{let}\ ip_{int}\ =\ \lambda t.\ \mathcal{I}_{|2|}\mathsf{toInt}\ ip(t) \\
\qquad \textbf{in} \\
\qquad \textbf{let}\ op_{int}\ =\ \lambda t.\ \mathcal{I}_{|2|}\mathsf{toInt}\ op(t) \\
\qquad \textbf{in} \\
\qquad \forall t.\ ip_{int}(t) + S(t)\ =\ 2^{n+1} op_{int}(t) + S(t{+}1)
\end{array}
\tag{4.4}
$$

4.3 Implementation

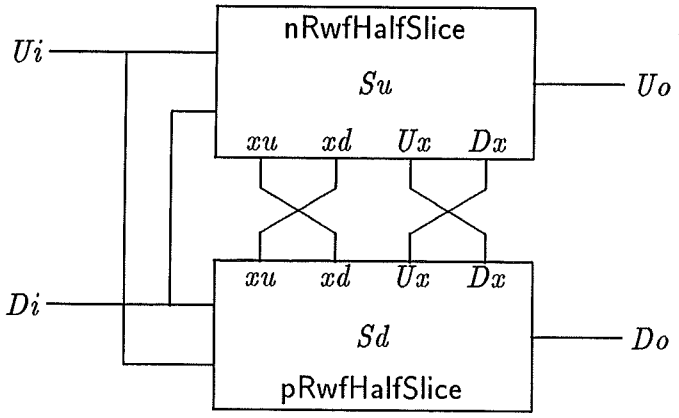
The implementation of the RWF device in the the CLIC design style is done in a structured way. The design of the whole device is hierarchically separated into the design of a series of lower level components. At the lowest level there is a device called the RwfHalfSlice. This is directly implemented in the CLIC design style using the various CLIC gates. There are two implementations of this component—an n-type and a p-type. These are illustrated in figure 4.4(a) and 4.4(b) respectively. Then the RwfSlice device is built by joining two of these RwfHalfSlice devices together. This is illustrated in figure 4.4(c). Finally a RWF of size n is built by chaining together n copies of the RwfSlice device together. A RWF device of size n means that the value at which it resets itself to zero is $\pm 2^{n+1}$.



(a) The n-type RwfHalfSlice



(b) The p-type RwfHalfSlice



(c) The RwfSlice

Figure 4.4: A CLIC gate level implementation of the RwfSlice device

The implementations of all of these devices is captured in formal notation as the following three definitions:

$$
\begin{aligned}
&\mathsf{nRwfHalfSlice}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, Ui, Di, xu, xd, Ux, Dx, Uo, St) \quad =_{def} \\
&\quad \exists p_1\ p_2\ p_3\ p_4\ p_5\ p_6. \\
&\qquad \mathsf{Stat\_Inv}(Ui, p_1) \qquad\qquad \wedge \\
&\qquad \mathsf{Latch}(\phi_1, \overline{\phi}_1, St, p_2) \qquad \wedge \\
&\qquad \mathsf{Latch}(\phi_1, \overline{\phi}_1, p_1, p_3) \qquad \wedge \\
&\qquad \mathsf{nXor}(\phi_1, \overline{\phi}_1, p_2, p_3, Ux) \qquad \wedge \\
&\qquad \mathsf{nNor}_2(\overline{\phi}_1, p_2, p_3, xu) \qquad \wedge \\
&\qquad \mathsf{pNand}_3(\phi_1, p_2, p_3, Dx, p_4) \wedge \\
&\qquad \mathsf{pNand}_2(\phi_1, Ux, xd, p_5) \qquad \wedge \\
&\qquad \mathsf{nNand}_2(\overline{\phi}_1, p_4, p_5, p_6) \qquad \wedge \\
&\qquad \mathsf{Latch}(\phi_2, \overline{\phi}_2, p_6, St) \qquad \wedge \\
&\qquad \mathsf{nNor}_3(\overline{\phi}_2, p_1, St, Di, Uo)
\end{aligned}
\tag{4.5}
$$

$$
\begin{aligned}
&\mathsf{pRwfHalfSlice}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, Ui, Di, xu, xd, Ux, Dx, Uo, St) \quad =_{def} \\
&\quad \exists p_1\ p_2\ p_3\ p_4\ p_5\ p_6. \\
&\qquad \mathsf{Stat\_Inv}(Ui, p_1) \qquad\qquad \wedge \\
&\qquad \mathsf{Latch}(\phi_1, \overline{\phi}_1, St, p_2) \qquad \wedge \\
&\qquad \mathsf{Latch}(\phi_1, \overline{\phi}_1, p_1, p_3) \qquad \wedge \\
&\qquad \mathsf{nXor}(\phi_1, \overline{\phi}_1, p_2, p_3, Ux) \qquad \wedge \\
&\qquad \mathsf{nNor}_2(\overline{\phi}_1, p_2, p_3, xu) \qquad \wedge \\
&\qquad \mathsf{pNand}_3(\phi_1, p_2, p_3, Dx, p_4) \wedge \\
&\qquad \mathsf{pNand}_2(\phi_1, Ux, xd, p_5) \qquad \wedge \\
&\qquad \mathsf{nNand}_2(\overline{\phi}_1, p_4, p_5, p_6) \qquad \wedge \\
&\qquad \mathsf{Latch}(\phi_2, \overline{\phi}_2, p_6, St) \qquad \wedge \\
&\qquad \mathsf{pNor}_3(\overline{\phi}_2, p_1, St, Di, Uo)
\end{aligned}
\tag{4.6}
$$

$$
\begin{aligned}
&\mathsf{RwfSlice}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, Ui, Di, Uo, Do, Su, Sd) \quad =_{def} \\
&\quad \exists xu\ xd\ Ux\ Dx. \\
&\qquad \mathsf{nRwfHalfSlice}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, Ui, Di, xu, xd, Ux, Dx, Uo, Su) \wedge \\
&\qquad \mathsf{pRwfHalfSlice}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, Di, Ui, xd, xu, Dx, Ux, Do, Sd)
\end{aligned}
\tag{4.7}
$$

With these definitions in place, the implementation of the RWF device of size n can be given. Figure 4.5 below shows the circuit that implements this RWF device. Note that it is entirely built out of RwfSlice devices just as the dynamic shift register was as presented in the previous chapter. The RWF was in fact designed

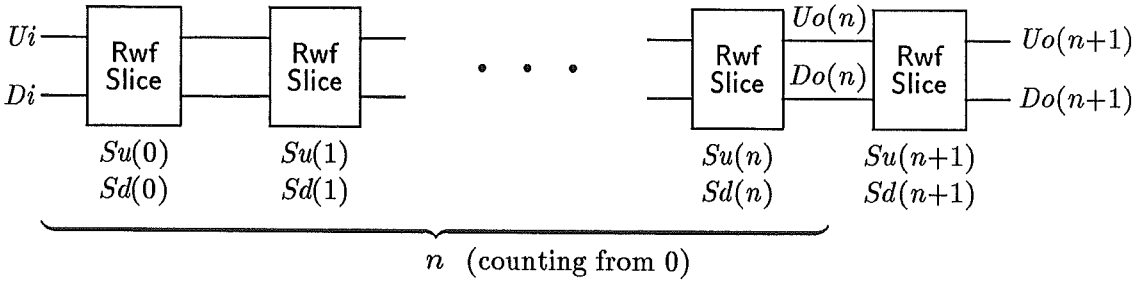


Figure 4.5: Implementing a Dynamic Shift Register in CLIC

this way intentionally. Other more efficient implementations could have been used but this style lends itself more easily to formal analysis.

This implementation is captured in formal notation as the following definition:

$$\begin{aligned}
&\mathsf{Rwf}\quad 0\quad \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ Ui\ Di\ Uo\ Do\ Su\ Sd\ =_{def}\\
&\quad \mathsf{RwfSlice}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, Ui, Di, Uo(0), Do(0), Su(0), Sd(0))\\
&\mathsf{Rwf}\ (n{+}1)\ \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ Ui\ Di\ Uo\ Do\ Su\ Sd\ =_{def}\\
&\quad \mathsf{Rwf}\ n\ \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ Ui\ Di\ Uo\ Do\ Su\ Sd\ \wedge\\
&\quad \mathsf{RwfSlice}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, Uo(n), Do(n), Uo(n{+}1), Do(n{+}1), Su(n{+}1), Sd(n{+}1))
\end{aligned} \tag{4.8}$$

4.4 Proof of Correctness

In order to make the presentation of this proof of correctness readable, most of the details of the proof are glossed over. The overall strategy used in constructing this proof however should be clear from what is described in this section. The detailed ML code for the proof of correctness of this device is given in appendix C.

The first thing that is done in the process of arriving at the correctness statement for any CLIC circuit is to ensure that the inputs and the outputs of the various gates are correctly connected. This is done by deriving the various theorems which state that the outputs of the circuit are "Well Behaved," and "Well Defined." Having ensured that the various gates are correctly interconnected, the next step is to derive the theorem which captures the behaviour of the circuit. The specification for the RWF has already been stated and defined as the predicate RwfSpec. In the remainder of this section some of the more interesting intermediate results are described, together with the final derived statement of correctness for this device.

One of the most useful lemmas in doing this proof is first stated here as equation 4.9. The derivation of this lemma is not presented here since it mostly involves

considerable manipulations to do with integers. The next chapter makes ample use of integer arithmetic and it should be clear from that how proofs involving integers are done.

$$\vdash \left(\begin{array}{ll} \exists x.\ \mathsf{RwfSpec}\ n\ ip\ x\ (\mathsf{Val}_{\mathcal{I}_{|2|}}\ n\ S) & \wedge \\ \quad \mathsf{RwfSpec}\ 0\ x\ op\ (\lambda t.\ \mathcal{I}_{|2|}\mathsf{toInt}(S\ (n{+}1)\ t)) & \end{array}\right) \supset \mathsf{RwfSpec}\ (n{+}1)\ ip\ op\ (\mathsf{Val}_{\mathcal{I}_{|2|}}\ (n{+}1)\ S) \tag{4.9}$$

where

$$\begin{aligned} \mathsf{Val}_{\mathcal{I}_{|2|}}\ \ 0\ \ S\ t\ \ &=_{def}\ \ \mathcal{I}_{|2|}\mathsf{toInt}(S\ 0\ t) \\ \mathsf{Val}_{\mathcal{I}_{|2|}}\ (n{+}1)\ S\ t\ \ &=_{def}\ \ (\mathsf{Val}_{\mathcal{I}_{|2|}}\ n\ S\ t)\ +\ 2^{n+1}(\mathcal{I}_{|2|}\mathsf{toInt}(S\ (n{+}1)\ t)) \end{aligned}$$

Informally, this lemma (equation 4.9) states that if two RwfSpec devices of size n and size 0 are composed together, then what results is a RwfSpec device of size $n{+}1$. There are a number of restrictions necessary for the validity of this lemma; namely that the value represented by the internal state is always less than 2^{n+1}, where n is the size of the RwfSpec device. This restriction is introduced implicitly in the lemma as apparent by the use of the two functions $\mathcal{I}_{|2|}\mathsf{toInt}$ and $\mathsf{Val}_{\mathcal{I}_{|2|}}$. In fact this lemma more accurately captures the behaviour of a chain of devices which are very similar to the RwfSpec device of size 0, with the only difference being that the type of the internal state of these devices is $\mathcal{I}_{|2|}$. Such a device can easily be defined and will be quite useful in the proof that follows. The definition of this is given in equation 4.10. Here a new predicate Slice is defined with three arguments all of which are now a sequence of values of type $\mathcal{I}_{|2|}$.

$$\begin{aligned} \mathsf{Slice}\ ip\ op\ S\ \ =_{def}\ \ & \textbf{let}\ S_{int}\ =\ \lambda t.\ \mathcal{I}_{|2|}\mathsf{toInt}(S\ t) \\ & \textbf{in} \\ & \mathsf{RwfSpec}\ 0\ ip\ op\ S_{int} \end{aligned} \tag{4.10}$$

With the aid of this predicate, the statement of correctness capturing the behaviour of the RwfSlice device can be derived. The actual derived theorem is stated here as equation 4.11. Note that it has the same general form as the various derived correctness statements of CLIC circuits in the previous chapter. This theorem (equation 4.11) can be stated in English as follows:

Assuming a correct relation exists between the clock lines,

and a correct implementation of the RwfSlice device exists,

and the signals on the two input lines Ui and Di are "Well Behaved," *and* "Well Defined" at certain points in time during every clock cycle,

and the values of the internal states and the outputs of the RwfSlice device are initially defined to be Hi and Lo respectively,

then allowing for certain abstractions, the inputs, the outputs and the internal states of this device satisfy the behaviour as defined by the predicate Slice.

$$
\vdash \left(\begin{array}{l}
\mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) \\
\mathsf{RwfSlice}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2, Ui, Di, Uo, Do, Su, Sd) \\
\mathsf{Wb}\ Ui\ \phi_2 \\
\mathsf{Wb}\ Di\ \overline{\phi}_2 \\
\sim((\mathsf{Val\_Abs}\ Uo)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ 0) \\
\sim((\mathsf{Val\_Abs}\ Do)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ 0) \\
((\mathsf{Val\_Abs}\ Su)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ 0) \\
((\mathsf{Val\_Abs}\ Sd)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ 0) \\
(\forall t.\ (\mathsf{Def}\ Ui)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ t) \\
(\forall t.\ (\mathsf{Def}\ Di)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ t) \\
(\forall t.\ (\mathsf{Def}\ Ui \circ \${+}3)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ t) \\
(\forall t.\ (\mathsf{Def}\ Di \circ \${+}3)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ t)
\end{array}\right) \supset
$$

$$
\begin{array}{l}
\textbf{let}\ \underline{Ui} = (\mathsf{Val\_Abs}\ Ui)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ \underline{Di} = (\mathsf{Val\_Abs}\ Di)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ ip = \lambda t.\ \mathsf{bbto}\mathcal{I}_{|2|}(\underline{Ui}\ t,\ \underline{Di}\ t) \\
\textbf{in} \\
\textbf{let}\ \underline{Uo} = (\mathsf{Val\_Abs}\ Uo)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ \underline{Do} = (\mathsf{Val\_Abs}\ Do)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ op = \lambda t.\ \mathsf{bbto}\mathcal{I}_{|2|}(\underline{Uo}\ t,\ \underline{Do}\ t) \\
\textbf{in} \\
\textbf{let}\ \underline{Su} = (\mathsf{Val\_Abs}\ Su)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ \underline{Sd} = (\mathsf{Val\_Abs}\ Sd)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ S = \lambda t.\ \mathsf{bbto}\mathcal{I}_{|2|}(\sim\underline{Su}\ t,\ \sim\underline{Sd}\ t) \\
\textbf{in} \\
\mathsf{Slice}\ ip\ op\ S
\end{array}
\qquad (4.11)
$$

The intermediate results so far can now be summarised as follows:

$$\vdash \ldots \mathsf{RwfSlice} \ldots \supset \ldots \mathsf{Slice} \ldots$$

$$\vdash \ldots \quad \mathsf{Slice} \quad \ldots = \ldots \mathsf{RwfSpec}\ 0 \ldots$$

By using these two theorems together with the lemma regarding RwfSpec stated earlier as equation 4.9, the final result can now be derived. The derivation of this theorem proceeds by a chain of derivations based on the fact that the form of the lemma (equation 4.9) is identical to the form of the implementation (equation 4.8). Note that the implementation consists of a chain of devices known as RwfSlice. This is directly mimicked by the specification, as is apparent by the lemma (equation 4.9), where use is made of the predicate Slice. Since one of the above two results (equation 4.11) shows that the RwfSlice device is a correct implementation of the specification stated by Slice, then by a little extra logical manipulation, the final result is derived. This is stated below as equation 4.12.

In its simplest form, the final derived theorem(equation 4.12) states that the implementation represented by the predicate Rwf, implies the specification represented by the predicate RwfSpec. This is naturally allowing for certain abstractions, both on the data type and the time grains at which the signals are viewed at various nodes. As before this result can be stated in English as follows:

Assuming a correct relation exists between the clock lines,

and a correct implementation of the Rwf device exists,

and the signals on the two input lines *Ui* and *Di* are "Well Behaved," *and* "Well Defined" at certain points in time during every clock cycle,

and the values of the internal states of each of the sub-components are initially Hi,

and the values on the outputs of each of the sub-components are initially Lo,

then allowing for certain abstractions, the inputs, the outputs and the internal states of this device satisfy the behaviour as defined by the predicate Rwf.

$$
\vdash \left(\begin{array}{ll}
\mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\
\mathsf{Rwf}\ n\ \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ Ui\ Di\ Uo\ Do\ Su\ Sd & \wedge \\
\mathsf{Wb}\ Ui\ \phi_2 & \wedge \\
\mathsf{Wb}\ Di\ \overline{\phi}_2 & \wedge \\
(\forall n.\ \sim((\mathsf{Val\_Abs}\ (Uo\ n))\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ 0) & \wedge \\
(\forall n.\ \sim((\mathsf{Val\_Abs}\ (Do\ n))\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ 0) & \wedge \\
(\forall n.\ ((\mathsf{Val\_Abs}\ (Su\ n))\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ 0) & \wedge \\
(\forall n.\ ((\mathsf{Val\_Abs}\ (Sd\ n))\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)\ \ 0) & \wedge \\
\vdots & \\
\text{The lines } Ui \text{ and } Di \text{ are defined at various times.} & \\
\vdots &
\end{array} \right) \supset
$$

$$
\begin{array}{l}
\textbf{let}\ \underline{Ui}\ =\ (\mathsf{Val\_Abs}\ Ui)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ \underline{Di}\ =\ (\mathsf{Val\_Abs}\ Di)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ ip\ =\ \lambda t.\ \mathsf{bbto}\mathcal{I}_{|2|}(\underline{Ui}\ t,\ \underline{Di}\ t) \\
\textbf{in} \\
\textbf{let}\ \underline{Uo}\ =\ \mathsf{Val\_Abs}(Uo\ n)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ \underline{Do}\ =\ \mathsf{Val\_Abs}(Do\ n)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1) \\
\textbf{in} \\
\textbf{let}\ op\ =\ \lambda t.\ \mathsf{bbto}\mathcal{I}_{|2|}(\underline{Uo}\ t,\ \underline{Do}\ t) \\
\textbf{in} \\
\textbf{let}\ \underline{Su}\ =\ \lambda x.\ \$\sim \mathsf{o}\ (\mathsf{Val\_Abs}(Su\ x)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)) \\
\textbf{in} \\
\textbf{let}\ \underline{Sd}\ =\ \lambda x.\ \$\sim \mathsf{o}\ (\mathsf{Val\_Abs}(Sd\ x)\ \mathsf{when}\ (\mathsf{isHi}\ \phi_1)) \\
\textbf{in} \\
\textbf{let}\ S\ =\ \lambda x\ t.\ \mathsf{bbto}\mathcal{I}_{|2|}(\underline{Su}\ x\ t,\ \underline{Sd}\ x\ t) \\
\textbf{in} \\
\mathsf{RwfSpec}\ n\ ip\ op\ (\mathsf{Val}_{\mathcal{I}_{|2|}}\ n\ S)
\end{array}
\tag{4.12}
$$

Where \$ in the above expression is a means of quoting the negation($\sim$) function.

4.5 Summary

In this chapter a fairly complex device known as the Random Walk Filter (RWF) has been analysed. A full gate level implementation of this device in the CLIC design style has been presented. The design is done as a series of cells which can be joined together to generate a RWF device of arbitrary size. The critical path through a cell is optimised such that there is minimal delay, in this case only one gate delay.

A formal specification for this device is then formulated. This specification is formulated with the assumption that at start-up time, the internal state of the RWF device is zero. Since nothing in the design of the RWF device states that the internal state at start-up time should be zero, the final derived statement of correctness instead carries it as an assumption. Also since the specification is stated at a fairly abstract level, various abstraction functions are used in the final statement of correctness to link the specification with the implementation.

The form of the correctness statements for the various cells, and for the random walk filter, are identical to the various examples presented in the previous chapter. This is an important factor in using this formal design technique. If at each level the statement of correctness has the same general form, then arbitrary mixing of complex and trivial devices can be done with ease. This is often the case in designing large circuits, where the outputs of large macro blocks are connected to the inputs of primitive gates and vice versa. The example used in the next chapter is typical in this respect.

Chapter 5

Proof Plan for the Correctness of a Window Comparator

In this chapter a plan for the proof of correctness of the Window Comparator is presented. This is also known as the CWIT *device — Compare WithIn Tolerance. The proof uses integers and modular arithmetic, but it does not go all the way down to the transistor level as in the previous chapter. However, the primitive devices used in constructing this device can be trivially proved down to the transistor level. An informal specification is first given which is then improved as the proof develops.*

5.1 Introduction

Content addressable memories, associative memories and window addressable memories: these are the areas for which the Window Comparator device was originally designed. Window Addressable Memories (or WAMs) are a class of devices where the memory architecture is set up in such a way that the incoming data is compared to the stored values, and the address of a matching data-set is output.

One particular WAM design developed at Racal Research by Orton and his team [Darby 86], had the requirements that the incoming data only had to be within the window setup by the data set, rather than being an exact match. In this design the stored data is split into n words. For each of these n words there is a corresponding tolerance value that is stored in a parallel bank of memory. These two banks of memories are set up as two barrels which can be rotated by

incrementing the address counter. Then, by wiring the window comparator in the middle of these two memory banks, the design of a window addressable memory results. This WAM provides the address of the first data-set that contains the value of the incoming data using an n-dimensional window space.

The architecture and design of this WAM is not the concern of this chapter, but the window comparator used in its design is. This WAM does however illustrate the possible uses of the window comparator. See [Darby 86] for further details of this WAM design.

Presented here is an outline of a proof of correctness of the window comparator (also known as the CWIT device — **C**ompare **W**ith**I**n **T**olerance). Some of the steps in the proof are difficult to formalise though the intuitions behind them are relatively straightforward. The presentation given relies on the readers intuition of arithmetic and hence cannot be called a formal proof. However the proof plan presented here contains sufficient details, and can be used as the basis for doing this proof in an environment in which each step is checked by a theorem proving system giving rise to a formal proof.

Before proceeding further, the following (old) question arises:

"What does it mean to do a proof?"

A proof in the context of this chapter means showing that the implementation implies the top level specification. So, to finally arrive at this implication, the remainder of this chapter is separated into the following parts:

- Formulating the specifications.
- The presentation of the implementation, i.e. a means of realising the specifications in hardware.
- A proof outline to show the correctness of this device.

5.2 Formulating the Specification

The CWIT device is simply a device which takes three numbers and returns an answer True or False. This is illustrated in figure 5.1 as a device with three inputs and a single output. The function of this device is to return a True answer if the number on the *data* input lies within the range "$mean - margin$" to "$mean + margin$," and False otherwise.

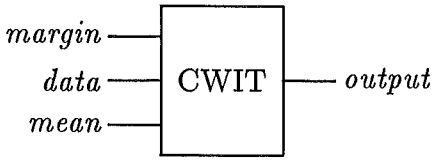


Figure 5.1: Top Level view of the CWIT device

This English description can be formally stated as follows:

$$\begin{aligned} &\mathsf{Cwit\_Spec}\ mean\ margin\ data\ output\ =_{def} \\ &\quad output\ =\ ((mean - margin) \leq data \leq (mean + margin)) \end{aligned} \tag{5.1}$$

5.3 Implementation

The implementation for performing this CWIT operation is illustrated by the diagram in figure 5.2. The architecture used is optimised for speed of calculation rather than reducing the size, or simplicity of design.

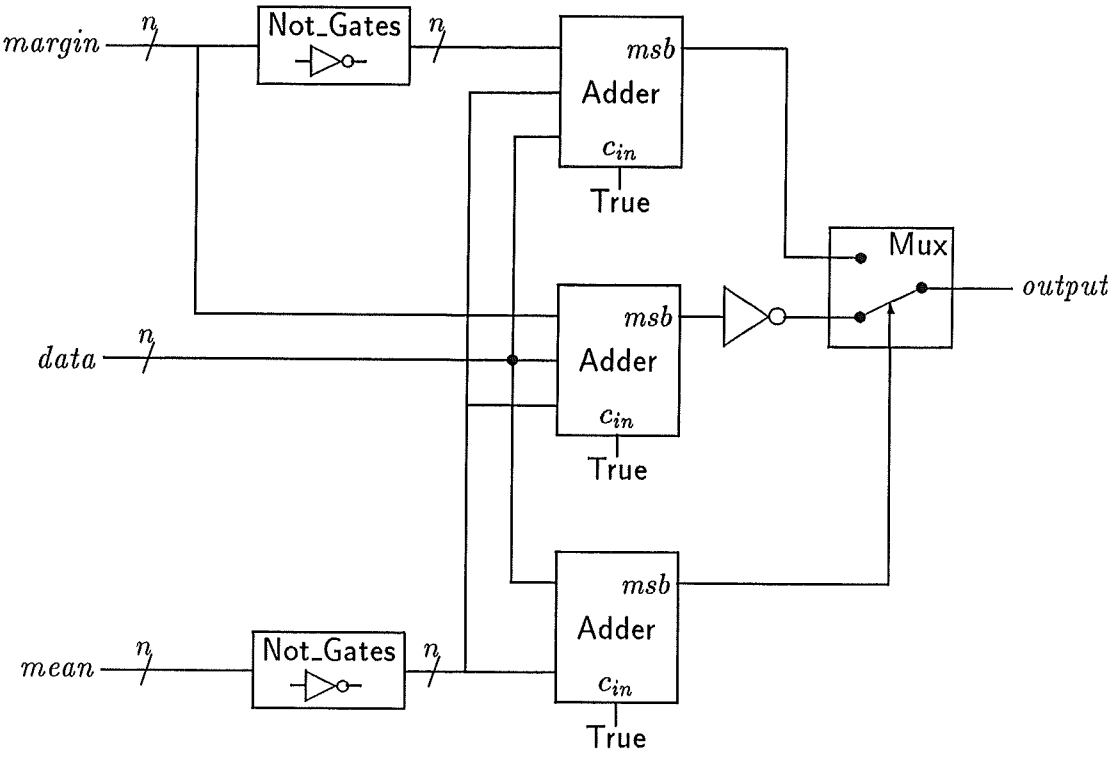


Figure 5.2: Block Diagram of the CWIT Device

Note that *data*, *mean*, and *margin* are parallel n-bit inputs and the *output* is a single bit which is either True or False. All functional boxes are individually

realisable in hardware and have relatively simple top level definitions. The three input adders could have been built out of two input adders, but this combined three input adder architecture was used for reasons of speed. Note also that only the most significant bit (MSB) of each of the adders is used, so a substantial part of the logic circuitry is missing. This further helps to speed up the calculation since it reduces the total loading on some of the internal gates. The two different kinds of NOT blocks in the above diagram are merely a short hand for a bit-wise negation of the signals going in. So the inside of an n-bit NOT block is simply n inverters in parallel.

Finally note that it is not obvious that this implementation actually performs the required CWIT operation. The purpose of the proof outline that follows will be to show that this is so.

5.4 Proof of Correctness

The proof presented is what is known as a "forward proof" where the top level statement of correctness is derived by composing the statement of correctness of the lower level devices. As an overview of the proof that is to follow, here is an approximate list of the steps that are involved:

1. State the interpretations given to the various signals on each of the wires or busses.

2. Specify the behaviour of all the internal blocks of the CWIT device in terms of the above interpretations for the various signals.

3. Compose the specifications of the internal blocks and derive the top level statement of correctness for the CWIT device.

4. Massage this derived top level statement of correctness and transform it into the form in which it is required.

5.4.1 Interpretations of Signals

Before the proof can begin, it is necessary to clearly state the interpretations given to the signals on each of the wires or busses. Consider for example the interpretation given to the *data* input. This could be interpreted as simply the set

of positive integers, or the set of positive and negative integers. In fact there are various different interpretations for negative integers that could be used.

To begin, note that in the specifications, a difference needs to be performed between two inputs as a means of finding out which of the two is greater. This implies that a subtraction needs to be done, followed by a check on the sign of the result. Now since a subtraction needs to be performed, it is convenient to use an interpretation on the inputs which allows negative number representation. This allows for many different representations but in this case the "2's complement" interpretation on the inputs is used. This is because it is easier to do both addition and subtraction using the same hardware if this interpretation is used. Also it is helpful not to have to do conversions in the proof from one number representation and another.

In summary it can be stated that all single bit wires are interpreted as booleans and all busses carrying numbers are interpreted in 2's complement form.

5.4.1.1 2's Complement Number Representation

An n-bit vector in 2's complement form represents a number in the range, -2^n to $2^n - 1$, where n in the number of bits counting from zero. e.g. for $n = 3$ the range of numbers is from -8 to 7, i.e. there are actually 4 bits in this vector.

The most significant bit of the number represents its sign, but the remaining bits do not represent the magnitude in a simple linear fashion as might be expected. This is so to make the hardware simple. i.e. a simple n-bit adder can be used to do both addition and subtraction by only using additional inverters, as is the case in the CWIT device. Note, to take the 2's complement of a number is equivalent to multiplying the number by minus one.

The following definitions are used to convert a bit-vector into an integer.

$$\begin{array}{lll} \mathsf{BitVal}\ x & =_{def} & \mathsf{If}\ \ (x) \\ & & \mathsf{Then}\ \ 1 \\ & & \mathsf{Else}\ \ 0 \end{array}$$

$$\begin{array}{lll} A^0 & =_{def} & 1 \\ A^{n+1} & =_{def} & A \times A^n \end{array}$$

$$\begin{array}{llll} \mathsf{Val} & 0 & x & =_{def}\ \ \mathsf{BitVal}\ x_0 \\ \mathsf{Val} & (n{+}1) & x & =_{def}\ \ \mathsf{Val}\ n\ x\ +\ 2^{n+1} \times \mathsf{BitVal}\ x_{n+1} \end{array}$$

$$\begin{array}{llll} \mathsf{iVal} & 0 & x & =_{def} \quad -\mathsf{BitVal}\ x_0 \\ \mathsf{iVal} & (n{+}1) & x & =_{def} \quad \mathsf{Val}\ n\ x\ -\ 2^{n+1} \times \mathsf{BitVal}\ x_{n+1} \end{array}$$

Note the last definition given above is not really recursive but it is presented in that form because the two cases are different.

5.4.2 Specification of the CWIT primitives

There are essentially three kinds of primitive devices used in the design of the CWIT device:

- The n-bit and the single bit NOT device.
- The three input and two input full adders, with carry in, and only the Most Significant Bit (MSB) as output.
- The Multiplexer (Mux) Device.

The specifications of these are given below.

5.4.2.1 The NOT Device

There are two versions of this device. One version takes a single input and generates a single output, and the other takes n inputs and generates n outputs. In order to differentiate between the two, the first shall be referred to as the Not_Gate device and the second the Not_Gates device. These can both be defined as follows:

$$\mathsf{Not\_Gate}\ x\ y \quad =_{def} \quad (y\ =\ \sim x)$$

$$\mathsf{Not\_Gates}\ n\ x\ y \quad =_{def} \quad \bigwedge_{i=0}^{n} \mathsf{Not\_Gate}\ x_i\ y_i$$

Notice how the second definition is defined in terms of the first. This directly reflects the implementation, which states that the n-bit Not_Gates device is simply a parallel array of inverters, i.e. a parallel array of Not_Gate devices.

Having stated these definitions is not enough. Recall that the 2's complement number representation is being used on all busses. So in order to make the specifications of the Not_Gates device useful, its behaviour must be represented in terms

of 2's complement numbers both going in and coming out. Rather than state the behaviour of the Not_Gates device in this way, it can be derived from the the previously defined iVal function and the other related definitions. However for the moment only the final derived result is stated:

$$\mathsf{Not\_Gates}\ n\ x\ y \ = \ (\mathsf{iVal}\ n\ y \ = \ -(\mathsf{iVal}\ n\ x \ + \ 1)\ \mathsf{imod}\ 2^{n+1}) \tag{5.2}$$

Where imod is used to represent modular arithmetic over the integers just as one uses the mod function over the natural numbers. The definition for the imod predicate is given below. An important thing to note about this definition is that it most naturally reflects the way one thinks about the addition that occurs when using an n-bit hardware adder. So this will also be useful in specifying the behaviour of the adder device.

$$\begin{array}{lll} x\ \mathsf{imod}\ 2^{n+1} =_{def} & \mathsf{If} & x \geq 2^n \quad \mathsf{Then}\ (x - 2^{n+1})\ \mathsf{imod}\ 2^{n+1} \\ & \mathsf{ElseIf} & x < -2^n \quad \mathsf{Then}\ (x + 2^{n+1})\ \mathsf{imod}\ 2^{n+1} \\ & & \mathsf{Else} \quad x \end{array}$$

5.4.2.2 The ADDER Device

The Adder device as used in the CWIT implementation is not what is normally understood to be the hardware adder. Normally an n-bit adder has an n-bit output but in the case of this Adder device, only the MSB is computed and presented at the output. In order to appreciate what is really going on, think of this Adder device as being composed of two devices. One is the traditional adder device with an n-bit output, and the other is a "ghost" device which selects the most significant bit of the output of the adder. For clarity these two new internal devices shall be called the Adder′ and the MSB′ devices respectively, and the final device they form shall be referred to as the Adder device. This is illustrated in figure 5.3.

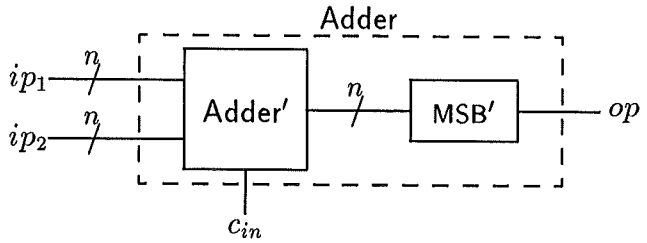


Figure 5.3: Inside view of the 2 input Adder as used in the CWIT device

In selecting the MSB of the output of the Adder′, the MSB′ "ghost" device also eats away the excess circuitry in the Adder′. So the net effect of having these

two devices connected in this way is to leave behind only that circuitry which is necessary to calculate the MSB of the add operation over the two inputs.

Rather than giving a single definition for the Adder device, given below are definitions for both the internal devices. The reason for this will become clear when the time comes to compose all of these device specifications together to generate the top level specification for the CWIT device.

Here then are the definitions for the two internal devices of the Adder. These definitions do not reflect how the Adder′ or the MSB′ devices are implemented, but merely state their most abstract specifications.

$$\mathsf{Adder}'\ n\ c_{in}\ ip_1\ ip_2\ sum \quad =_{def} \quad \mathsf{iVal}\ n\ sum\ =\ \begin{pmatrix}\mathsf{iVal}\ n\ ip_1\ +\\ \mathsf{iVal}\ n\ ip_2\ +\\ \mathsf{BitVal}\ c_{in}\end{pmatrix} \mathsf{imod}\ 2^{n+1}$$

$$\mathsf{MSB}'\ n\ ip\ sign \quad =_{def} \quad (sign\ =\ -2^n \leq (\mathsf{iVal}\ n\ ip) < 0)$$

In the definition of the MSB′ device it is necessary to make clear that two separate comparisons are not being performed followed by a conjunction of the results. Under modulo arithmetic there is no "less than" operation, so the above definition must be interpreted as a means of checking to see if (iVal n ip) lies within the specified range. An alternative to this notation is to use the "interval" notation. There are four different sorts of intervals which can be represented in this interval notation, namely:

1. $x\ \epsilon\ (a,b) \equiv a < x < b$
2. $x\ \epsilon\ (a,b] \equiv a < x \leq b$
3. $x\ \epsilon\ [a,b) \equiv a \leq x < b$
4. $x\ \epsilon\ [a,b] \equiv a \leq x \leq b$

The third one of the above interval structures would be used in the MSB′ device. The definition of the MSB′ device using this interval structure would then be modified to look as follows:

$$\mathsf{MSB}'\ n\ ip\ sign \quad =\quad (sign\ =\ (\mathsf{iVal}\ n\ ip)\ \epsilon\ [-2^n, 0))$$

An interesting feature of this particular interval structure is that, to say something does *not* belong to an interval is equivalent to swapping the two limits.

Rule 5.1 (NOT Rule) $\sim(x \in [a,b)) \equiv x \in [b,a)$

This will be useful when the specifications of these devices are composed together.

Before going on there is perhaps one further advantage about this half closed interval structure which needs to be pointed out. Recall that the specification for the Adder′ device uses the imod predicate. In natural numbers (x mod n) is always less than n; i.e. (x mod n) $< n$. However for the integers this changes and, based on the definition of the imod predicate given earlier, it becomes:

$$\forall\, x\ n.\quad -2^n \leq (x \text{ imod } 2^{n+1}) < 2^n$$

This can now be rewritten using the same half closed interval structure as used for the MSB′ device as follows:

$$\forall\, x\ n.\ (x \text{ imod } 2^{n+1}) \in [-2^n, 2^n)$$

Note: The definition for the three input Adder as used in the Window Comparator device is a simple extension of the definitions given above, and will not be presented here.

5.4.2.3 The MUX Device

This device has three inputs and a single output as illustrated in figure 5.4. It is perhaps the simplest to define, as compared to the Adder or the NOT devices as given above. In English the behaviour could be stated as "If the *ctl* input is high, Then the *output* equals ip_1, Else the *output* equals ip_2".

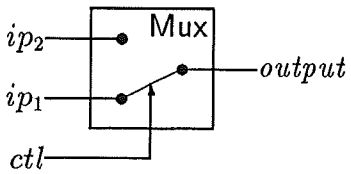


Figure 5.4: Multiplexer used in the CWIT device

This English description can be formally stated as follows:

$$\begin{array}{lll} \text{Mux } ctl\ ip_1\ ip_2\ output & =_{def} & output = \text{If } (ctl) \\ & & \text{Then } ip_1 \\ & & \text{Else } ip_2 \end{array} \tag{5.3}$$

5.4.3 Top Level Behaviour of CWIT

In this section the top level specification of the CWIT device is derived using the various definitions of the primitives and the interpretation functions stated above.

To start with, notice that the *mean* input goes through the Not_Gates device to the three Adder devices (see figure 5.2 on page 89). Also, all these three Adder devices have their c_{in} signals held true. Taking just one of these slices and exploding the Adder device to show the internal "ghost" device gives rise to the picture in figure 5.5.

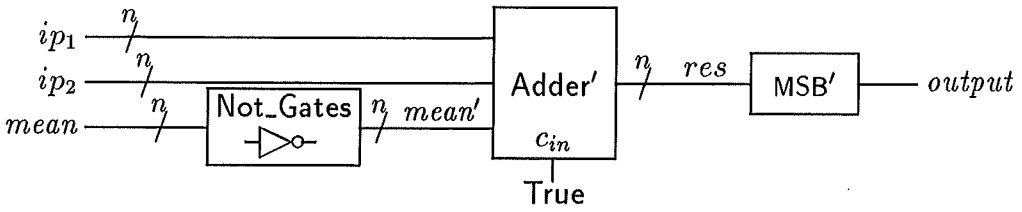


Figure 5.5: Single Stage of the inside of the CWIT device

Using the names of the internal nodes as stated in figure 5.5, the structure of this Slice can be formally stated as follows:

$$\begin{aligned}
&\mathsf{Slice}\ n\ ip_1\ mean\ ip_2\ output\ =_{def}\\
&\quad \exists mean'\ c_{in}\ res.\ (c_{in} = \mathsf{True}) \wedge\\
&\qquad\qquad \mathsf{Not_Gates}\ n\ mean\ mean' \wedge\\
&\qquad\qquad \mathsf{Adder'}\ n\ c_{in}\ ip_1\ ip_2\ mean'\ res \wedge\\
&\qquad\qquad \mathsf{MSB'}\ n\ res\ output
\end{aligned}$$

From this definition the following theorem stating the behaviour of the Slice device can easily be derived.

$$\begin{aligned}
&\mathsf{Slice}\ n\ ip_1\ mean\ ip_2\ output\ \supset\\
&output\ = \left(\begin{pmatrix}\mathsf{iVal}\ n\ ip_1 & + \\ \mathsf{iVal}\ n\ ip_2 & - \\ \mathsf{iVal}\ n\ mean & \end{pmatrix} \mathsf{imod}\ 2^{n+1}\right) \epsilon\ [-2^n, 0)
\end{aligned} \tag{5.4}$$

The steps involved in arriving at this theorem are as follows:

1. Set the goal to be proved:

$$\frac{\begin{array}{l}\mathsf{Slice}\ n\ ip_1\ mean\ ip_2\ output\ \supset \\ \text{“}\ output\ =\ \left(\begin{pmatrix}\mathsf{iVal}\ n\ ip_1\ + \\ \mathsf{iVal}\ n\ ip_2\ - \\ \mathsf{iVal}\ n\ mean\end{pmatrix}\ \mathsf{imod}\ 2^{n+1}\right)\ \epsilon\ [-2^n, 0)\ \text{”}\end{array}}{[]}$$

2. Rewrite for Slice in the above subgoal and then move the result to the assumption list. Simplify the assumption list by first choosing free names for the existentially quantified variables and then splitting up the resultant conjunct.

$$\frac{\text{“}\ output\ =\ \left(\begin{pmatrix}\mathsf{iVal}\ n\ ip_1\ + \\ \mathsf{iVal}\ n\ ip_2\ - \\ \mathsf{iVal}\ n\ mean\end{pmatrix}\ \mathsf{imod}\ 2^{n+1}\right)\ \epsilon\ [-2^n, 0)\ \text{”}}{\left[\begin{array}{l}(c_{in} = \mathsf{True}) \\ \mathsf{Not\_Gates}\ n\ mean\ mean' \\ \mathsf{Adder'}\ n\ c_{in}\ ip_1\ ip_2\ mean'\ res \\ \mathsf{MSB'}\ n\ res\ output\end{array}\right]}$$

3. Rewrite for the specifications of the Not_Gates, Adder′ and the MSB′ devices in the above assumption list.

$$\frac{\text{“}\ output\ =\ \left(\begin{pmatrix}\mathsf{iVal}\ n\ ip_1\ + \\ \mathsf{iVal}\ n\ ip_2\ - \\ \mathsf{iVal}\ n\ mean\end{pmatrix}\ \mathsf{imod}\ 2^{n+1}\right)\ \epsilon\ [-2^n, 0)\ \text{”}}{\left[\begin{array}{l}(c_{in} = \mathsf{True}) \\ (\mathsf{iVal}\ n\ mean' = -(\mathsf{iVal}\ n\ mean\ +\ 1)\ \mathsf{imod}\ 2^{n+1}) \\ \mathsf{iVal}\ n\ res\ =\ \begin{pmatrix}\mathsf{iVal}\ n\ ip_1\ + \\ \mathsf{iVal}\ n\ ip_2\ + \\ \mathsf{iVal}\ n\ mean'\ + \\ \mathsf{BitVal}\ c_{in}\end{pmatrix}\ \mathsf{imod}\ 2^{n+1} \\ (output\ =\ (\mathsf{iVal}\ n\ res)\ \epsilon\ [-2^n, 0))\end{array}\right]}$$

4. Finally the theorem is proved by rewriting with the assumptions together with the following simplifying theorems:

$$\begin{array}{rcl}\mathsf{BitVal}\ \mathsf{True} & = & 1 \\ -(\mathsf{iVal}\ n\ mean\ +\ 1)\ \mathsf{imod}\ 2^{n+1} & = & -(\mathsf{iVal}\ n\ mean\ +\ 1) \\ \cdots\ -1+1 & = & \cdots\end{array}$$

With the aid of this relatively abstract specification for the Slice(equation 5.4), a more usable description can now be given to the signals coming out of the three Adder devices as shown in figure 5.2 on page 89. The block diagram in figure 5.6 below shows the CWIT device with the Slice block inserted in place the Adder, etc. devices.

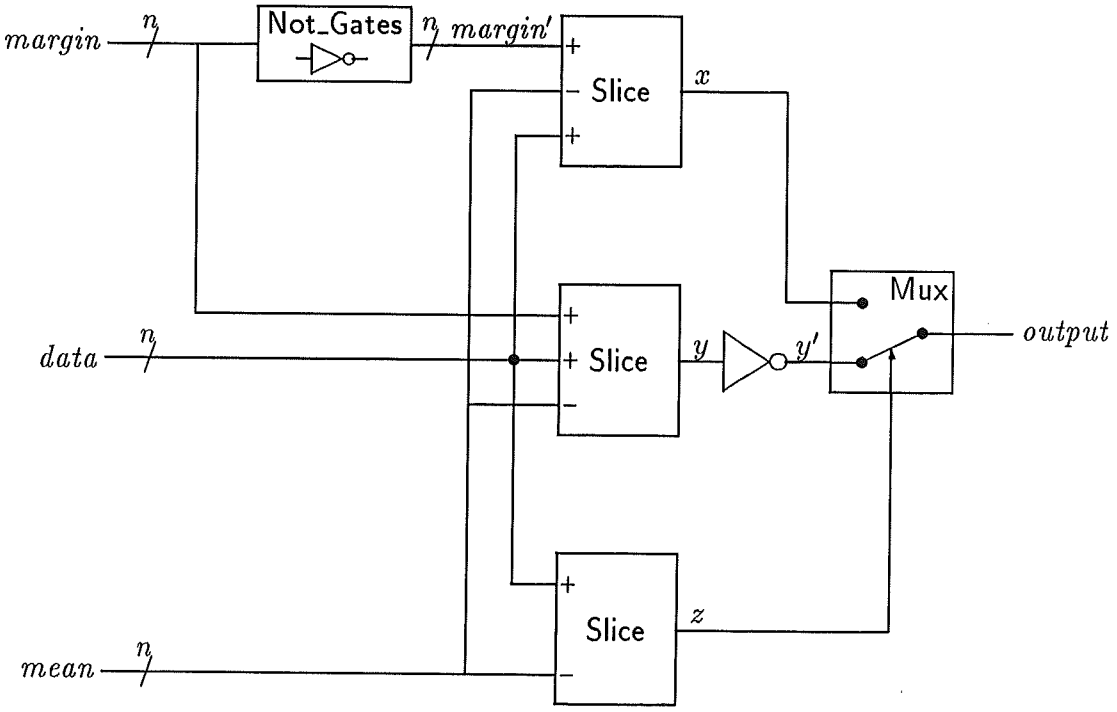


Figure 5.6: Block Diagram of the CWIT Device using Slice

Note that for this proof all arithmetic has been "imod 2^{n+1}" and will continue unchanged. Also, for all the busses, their value is always used under the assumption that they carry a number represented in 2's complement form, hence the iVal function is always used. So for brevity the following notational abbreviations will be made:

$$
\begin{aligned}
(x + y) \text{ imod } 2^{n+1} &= (x \boxplus y) \\
(x - y) \text{ imod } 2^{n+1} &= (x \boxminus y) \\
(\text{iVal } n \ data) &= \underline{data} \\
(\text{iVal } n \ mean) &= \underline{mean} \\
(\text{iVal } n \ margin) &= \underline{margin}
\end{aligned}
$$

The proof of the CWIT device given below follows almost an identical pattern as that presented for the Slice device above. So, rather than writing large equations giving all the assumptions etc., only the interesting parts of the goal are shown at

any time. This is done in the hope of making the proof more readable, but the derived result will be shown in full at the end.

Using the specifications for the Slice block as derived above (equation 5.4), the equations describing the signals on the internal wires x, y, z and y' can be stated as follows:

$$\begin{aligned} x &= (\underline{data} \boxplus \underline{margin'} \boxminus \underline{mean}) \in [-2^n, 0) \\ y &= (\underline{data} \boxplus \underline{margin} \boxminus \underline{mean}) \in [-2^n, 0) \\ z &= (\underline{data} \boxminus \underline{mean}) \in [-2^n, 0) \\ y' &= \sim y \end{aligned}$$

Where

$$\underline{margin'} = -\underline{margin} \boxminus 1 \qquad \text{by equation 5.2}$$

Rewriting for $\underline{margin'}$ in the equation for x above and simplifying gives:

$$x = (\underline{data} \boxminus \underline{margin} \boxminus \underline{mean} \boxminus 1) \in [-2^n, 0)$$

and rewriting for y in the equation for y' above and simplifying using the NOT Rule (Rule 5.1, page 95) gives:

$$y' = (\underline{data} \boxplus \underline{margin} \boxminus \underline{mean}) \in [0, -2^n)$$

Now before going on to the Mux device, it is worth rearranging the three derived equations for x, y' and z as follows:

$$\begin{aligned} x &= \underline{data} \in [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1) \\ y' &= \underline{data} \in [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n) \\ z &= \underline{data} \in [\underline{mean} \boxminus 2^n,\ \underline{mean}) \end{aligned}$$

With the equations in this form and with the aid of the following rules for the intersection and union of intervals, the resultant equation for the *output* of the Mux device can be simplified considerably.

Rule 5.2 (Intersect) $\begin{pmatrix} x \in [a,b) & \wedge \\ x \in [c,d) & \end{pmatrix} \equiv x \in ([a,b) \cap [c,d))$

Rule 5.3 (Union) $\begin{pmatrix} x \in [a,b) & \vee \\ x \in [c,d) & \end{pmatrix} \equiv x \in ([a,b) \cup [c,d))$

Where $\cap$ and $\cup$ represent interval intersection and union respectively.

Now by using equation 5.3 from page 95, the specification for the Mux device in the above context can be stated as follows:

$$\begin{array}{lll} output = & \mathsf{If} & (z) \\ & \mathsf{Then} & y' \\ & \mathsf{Else} & x \end{array}$$

Rewriting for x, y' and z in this gives:

$$\begin{array}{lll} output = & \mathsf{If} & \underline{data} \in [\underline{mean} \boxminus 2^n,\ \underline{mean}) \\ & \mathsf{Then} & \underline{data} \in [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n) \\ & \mathsf{Else} & \underline{data} \in [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1) \end{array}$$

Now since all the components of the above "If... Then... Else..." statement are booleans, the following transformation can be applied.

Rule 5.4 $\begin{pmatrix} \mathsf{If} & a \\ \mathsf{Then} & b \\ \mathsf{Else} & c \end{pmatrix} \equiv (a \wedge b) \vee (\sim a \wedge c)$

This transformation together with the NOT Rule (Rule 5.1, page 95) applied to the result of the Mux device gives:

$$output = \begin{pmatrix} \begin{pmatrix} \underline{data} \in [\underline{mean} \boxminus 2^n,\ \underline{mean}) & \wedge \\ \underline{data} \in [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n) \end{pmatrix} & \vee \\ \begin{pmatrix} \underline{data} \in [\underline{mean},\ \underline{mean} \boxminus 2^n) & \wedge \\ \underline{data} \in [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1) \end{pmatrix} \end{pmatrix}$$

And now by using the union and intersection rules, this can be transformed to:

$$output = \underline{data} \in \begin{pmatrix} \begin{pmatrix} [\underline{mean} \boxminus 2^n,\ \underline{mean}) & \cap \\ [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n) \end{pmatrix} & \cup \\ \begin{pmatrix} [\underline{mean},\ \underline{mean} \boxminus 2^n) & \cap \\ [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1) \end{pmatrix} \end{pmatrix}$$

So the final derived result for the CWIT device can be stated as follows:

$$\mathsf{Cwit\_Imp}\ mean\ margin\ data\ output \supset$$
$$output = \underline{data} \in \left(\begin{array}{l} \left([\underline{mean} \boxminus 2^n,\ \underline{mean}) \quad \cap \quad [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n) \right) \\ \cup \\ \left([\underline{mean},\ \underline{mean} \boxminus 2^n) \quad \cap \quad [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1) \right) \end{array} \right)$$

where Cwit_Imp represents the implementation of the CWIT device.

Not surprisingly, this looks nothing like the original specifications(equation 5.1) for the CWIT device. However note that the original specifications were incomplete and did not contain the complications of number wrap around when using modulo arithmetic. If the numbers are guaranteed to be small and there is no wrap around when doing arithmetic, then the above derived result will simplify to look very much like equation 5.1 of the original specifications. Before attempting this it is necessary to transform the specifications as stated in equation 5.1 to be in the same form as the above derived result.

5.4.4 Specification Transformation

Here the specifications are transformed is such a way that they use the same interval notation as used in the above derived result. The steps involved are trivial and are not stated here; only the final results are shown. However these steps can be followed by reference to the definitions of the interval structures as stated on page 94.

$$\begin{aligned} output &= (mean - margin) \leq data \leq (mean + margin) \\ &= data \in [mean - margin,\ \ mean + data] \\ &= data \in [mean - margin,\ \ mean + data + 1) \end{aligned} \tag{5.5}$$

5.4.5 Result Transformation

In order to show that the implementation meets the specification under certain circumstances, a case split is done on the basis of the value of $\underline{margin}$.

- Either $\underline{margin}$ is positive. i.e. $\underline{margin} \in [0,\ 2^n)$
- Or $\underline{margin}$ is negative. i.e. $\underline{margin} \in [2^n,\ 0)$

In the proof that follows, use is made of circular diagrams as a means of representing modulo-n integers. Figure 5.7 illustrates a simple example where the entire circle represents the entire cyclic number range, and the directed arc represents the half closed interval $[a, b)$.

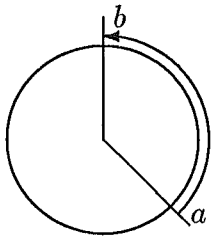


Figure 5.7: Example of representing an interval over a cyclic number range

Armed with this pictorial notation for cyclic numbers, the proof of the various cases can now begin.

5.4.5.1 Case: *margin* is positive

The equation that needs to be simplified is the right hand side of the derived result which can be stated as follows:

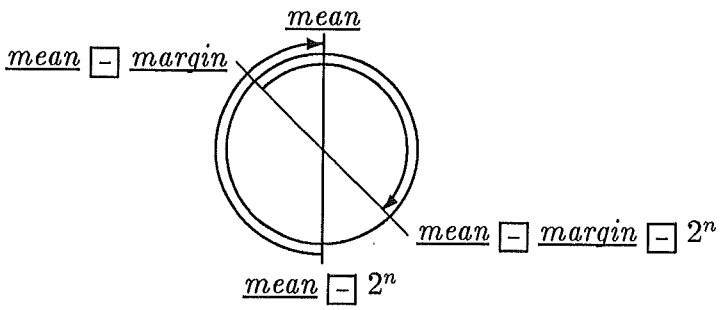
$$\underline{data} \in \left(\begin{array}{ll} ([\underline{mean} \boxminus 2^n, \underline{mean}) & \cap \\ [\underline{mean} \boxminus \underline{margin}, \underline{mean} \boxminus \underline{margin} \boxminus 2^n)) & \cup \\ ([\underline{mean}, \underline{mean} \boxminus 2^n) & \cap \\ [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n, \underline{mean} \boxplus \underline{margin} \boxplus 1)) & \end{array} \right)$$

This is best tackled by splitting the calculation into three parts; simplify the two intersections first and then take their union.

First Intersection

$$[\underline{mean} \boxminus 2^n, \underline{mean}) \cap [\underline{mean} \boxminus \underline{margin}, \underline{mean} \boxminus \underline{margin} \boxminus 2^n)$$

The simplified form of this can be easily seen from the following diagram illustrating an arbitrary intersection of these two intervals. Care has been taken to make sure that the two intervals are in the correct relation, i.e. the value "$\underline{mean} \boxminus \underline{margin}$" must lie within the range "$[\underline{mean} \boxminus 2^n, \underline{mean})$" to satisfy the initial condition on the value of $\underline{margin}$.

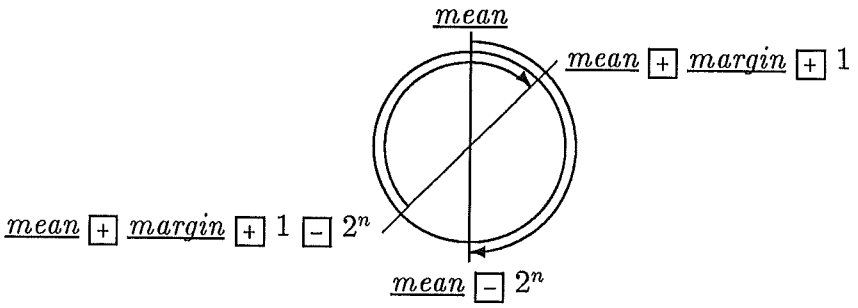


$$[\underline{mean} \boxminus 2^n,\ \underline{mean}) \ \cap \ [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n)$$
$$=_{simp} \ [\underline{mean} \boxminus \underline{margin},\ \underline{mean})$$

Second Intersection

$$[\underline{mean},\ \underline{mean} \boxminus 2^n) \ \cap \ [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1)$$

Following a similar argument to above but this time ensuring that the value "$\underline{mean} \boxplus \underline{margin} \boxplus 1$" lies within the range "$[\underline{mean},\ \underline{mean} \boxminus 2^n)$".



$$[\underline{mean},\ \underline{mean} \boxminus 2^n) \ \cap \ [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1)$$
$$=_{simp} \ [\underline{mean},\ \underline{mean} \boxplus \underline{margin} \boxplus 1)$$

Union of the above two results

Now the union of the above gives the required result:

$$\begin{aligned}&[\underline{mean} \boxminus \underline{margin},\ \underline{mean}) \ \cup \ [\underline{mean},\ \underline{mean} \boxplus \underline{margin} \boxplus 1)\\ &=_{simp} [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxplus \underline{margin} \boxplus 1)\end{aligned}$$

Stating this final result more clearly shows the similarity between this and the transformed specification as derived earlier (equation 5.5, page 101).

$$\begin{aligned}&\text{Assuming } \underline{margin} \in [0,\ 2^n)\\ &\quad \text{Then} \quad output = \underline{data} \in [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxplus margin \boxplus 1)\\ &\qquad\qquad = \underline{data} \in [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxplus margin]\end{aligned}$$

5.4.5.2 Case: $\underline{margin}$ is negative

Again the equation that needs to be simplified is the right hand side of the derived result which can be stated as follows:

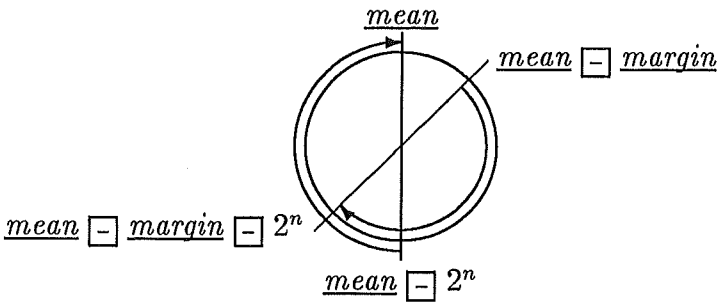
$$\underline{data} \in \left(\begin{pmatrix}[\underline{mean} \boxminus 2^n,\ \underline{mean}) \\ [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n)\end{pmatrix} \cap \cup \begin{pmatrix}[\underline{mean},\ \underline{mean} \boxminus 2^n) \\ [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1)\end{pmatrix}\right)$$

Exactly the same argument as for the previous case shall be followed except that this time around the assumption on the value of $\underline{margin}$ has changed. This will influence the way the intervals intersect and so the diagrams will change.

First Intersection

$$[\underline{mean} \boxminus 2^n,\ \underline{mean}) \ \cap \ [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n)$$

The result is again demonstrated by the familiar circular number system diagram as before but this time the value "$\underline{mean} \boxminus \underline{margin} \boxminus 2^n$" lies within the range "$[\underline{mean} \boxminus 2^n,\ \underline{mean})$" instead of "$\underline{mean} \boxminus \underline{margin}$" as in the previous case. This is because $\underline{margin}$ is negative this time around.

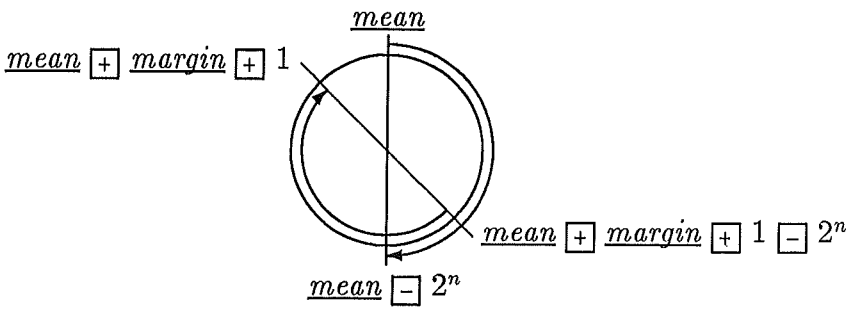


$$[\underline{mean} \boxminus 2^n,\ \underline{mean}) \ \cap \ [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n)$$
$$=_{simp} \ [\underline{mean} \boxminus 2^n,\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n)$$

Second Intersection

$$[\underline{mean},\ \underline{mean} \boxminus 2^n) \ \cap \ [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1)$$

Again following a similar argument to above but this time ensuring that the value "$\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n$" lies within the range "$[\underline{mean},\ \underline{mean} \boxminus 2^n)$".



$$[\underline{mean},\ \underline{mean} \boxminus 2^n) \ \cap \ [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxplus \underline{margin} \boxplus 1)$$
$$=_{simp} \ [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxminus 2^n)$$

Union of the above two results

Again the union of the above two intersection gives the required result.

$$\begin{pmatrix}[\underline{mean} \boxminus 2^n,\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n) \ \cup \\ [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxminus 2^n)\end{pmatrix}$$
$$=_{simp} [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n)$$

So the result for $\underline{margin}$ being negative can be summarised as follows:

Assuming $\underline{margin} \in [2^n, 0)$

Then $output = \underline{data} \in [\underline{mean} \boxplus \underline{margin} \boxplus 1 \boxminus 2^n,\ \underline{mean} \boxminus \underline{margin} \boxminus 2^n)$

$$= \underline{data} \in (\underline{mean} \boxminus 2^n \boxplus \underline{margin},\ \underline{mean} \boxminus 2^n \boxminus \underline{margin})$$

5.4.5.3 Final result by combining cases

By combining the above two cases which are based on the sign of $\underline{margin}$, the final result for the CWIT device can be derived. Since the behaviour is different dependent on the sign of $\underline{margin}$, this is reflected in the following derived equation which uses the "If... Then... Else..." construct to reflect this fact.

Cwit_Imp *mean margin data output* $\supset$

$output =$ If $\underline{margin} \in [0, 2^n)$

Then $\underline{data} \in [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxplus \underline{margin}]$

Else $\underline{data} \in (\underline{mean} \boxminus 2^n \boxplus \underline{margin},\ \underline{mean} \boxminus 2^n \boxminus \underline{margin})$

5.5 Summary

In this chapter a fairly complex proof of correctness has been outlined which uses integers and modular arithmetic. In order to construct the proof (and for the proof to be readable) many new predicates were defined which capture some of the behavioural properties of the components used. For example, the imod predicate was specifically defined in this fashion to reflect the 2's complement arithmetic capabilities of a hardware adder. Note that the CWIT device is not a trivial device, even though its top level specification is small. To give an indication of the size

of such a device, a 16-bit implementation of the CWIT device designed in the CLIC design style contains approximately 1500 transistors.

Here is a summary of the main results.

Abstract Specification:

$$\begin{aligned} output &= (mean - margin) \leq data \leq (mean + margin) \\ &= data \in [mean - margin,\ \ mean + data] \\ &= data \in [mean - margin,\ \ mean + data + 1) \end{aligned}$$

Derived Result from the Implementation:

$$\begin{aligned} &\mathsf{Cwit\_Imp}\ mean\ margin\ data\ output\ \supset \\ &\quad output = \mathsf{If} \quad \underline{margin} \in [0,\ 2^n) \\ &\qquad\qquad \mathsf{Then}\ \underline{data} \in [\underline{mean} \boxminus \underline{margin},\ \underline{mean} \boxplus \underline{margin}] \\ &\qquad\qquad \mathsf{Else}\ \ \underline{data} \in (\underline{mean} \boxminus 2^n \boxplus \underline{margin},\ \underline{mean} \boxminus 2^n \boxminus \underline{margin}) \end{aligned}$$

$$\begin{aligned} \text{Where} \qquad (x \boxplus y) &= (x + y)\ \mathsf{imod}\ 2^{n+1} \\ (x \boxminus y) &= (x - y)\ \mathsf{imod}\ 2^{n+1} \\ \underline{data} &= (\mathsf{iVal}\ n\ data) \\ \underline{mean} &= (\mathsf{iVal}\ n\ mean) \\ \underline{margin} &= (\mathsf{iVal}\ n\ margin) \end{aligned}$$

Note that the derived result uses abstraction functions. These functions map the representations of data and operations on that data from the implementation domain to their abstract counterparts. For example the function iVal maps from number represented in 2's complement form on a bus to an integer in base 10. Similarly, other functions in the derived result are used to mimic the operations performed in the specifications; but they too are limited to work within the restricted set of integers that a particular CWIT implementation is designed for.

With the aid of these abstraction functions it becomes easier to see that the implementation only meets the specifications if the inputs are restricted to be within certain limits. This is particularly so for the *margin* input. The initial specifications do not say anything about how numbers are to be represented, or even if there is any bound to the number range for which the specifications are valid. But from the derived result it is easily seen that if the conditions are not

met, then the result generated by the CWIT device will not be to the required specifications.

Note also that the initial specification was vague and incomplete. It said nothing of the range of numbers within which the device was to operate or if there were any restrictions as to its operation. However, it must be said that enough information was present to allow a designer to go ahead and design such a device. The final design works even if it imposes restrictions on the external ports, and it is up to the user of this device to say whether these restrictions are too severe or acceptable.

Perhaps the most valuable lesson that can be learnt from this exercise is that formal analysis, be it for reasons of proof or not, can, and does help improve and sharpen the final specifications.

Chapter 6

A New Design of a Verifiable Digital Phase-Locked Loop

This chapter first gives a brief overview of control systems with particular emphasis on Digital Phase-Locked Loops (DPLLs). Then a novel design for a new class of phase-locked loops is presented which uses the devices of the previous two chapters. A formal specification for this device is presented together with a sketch for an informal proof of correctness. Some of the difficulties involved in arriving at the correctness statement and constructing the proof are then discussed.

6.1 Introduction

Phase-Locked Loops (PLL) have many applications. They form the heart of most modern communications systems. Perhaps the most familiar device which uses a phase-locked loop is the high quality turntable, where the speed of the turntable is controlled by a simple PLL. Phase-locked loops constructed out of purely digital components are referred to as Digital Phase-Locked Loops (DPLL). They form a particularly interesting set of devices, since the cost of manufacturing them is considerably lowered with the use of modern integrated circuit technology. Furthermore, they do not suffer from the problems associated with their analogue counterparts: sensitivity to dc drifts and component saturations, the difficulties encountered in building higher order loops, and the need for both initial calibration and periodic adjustments. Elaborate real-time processing on the signal samples also adds to the attraction of DPLLs.

6.1.1 What is a Phase-Locked Loop?

The term "phase-locked loop" refers to a feedback control system in which the input and the feedback parameters of interest are the relative phases of the waveforms. The function of a PLL is to track small differences in phase between the input and the feedback signal. In the *IEEE Standard Dictionary* it is said that the loop circuit locks (synchronises) a variable local oscillator with the phase of an incoming signal. In fact there is slight contention in the literature as to what these devices should be called. The term "phase-locked loop" has stood the test of time, but some would argue that it does not reflect the true workings of the device. So other terms have been suggested, such as "phase-lock loop," "phase-locking loop," "phase-tracking loop" etc.. In the editorial note of the special issue of the *IEEE Transactions on Communications* on Phase-Locked Loops [Lindsey 82], Lindsey and Chie argue that the term "phase-tracking loop" should be adopted. They say that 'the loop tracks, or tries to track, as well as it can, but isn't really "locked" (after all the loop loses lock periodically).' To this a rather intriguing response is made, in the same article:

> A phase-lock(ed) loop: "After long years my phase is locked
> Synchronised to the incoming clock
> Now some thoughtless nincompoop
> Says I'm just a phase-tracking loop?"
> *–R. Huang, TRW Systems, Redondo Beach, CA*
> [Lindsey 82]

So now a "Feedback Control System" needs to be defined. A useful definition of this was given by the Institute of Radio Engineers (U.S.A) in 1956 as follows:

> **Feedback Control System.** A control system, comprising one or more feedback control loops, which combines functions of the controlled signals with functions of the commands to tend to maintain prescribed relationships between the commands and the controlled signals.
> [IRE 56, section 2.4.2]

The idea of automatic control is in fact quite old, stemming back to about 1790 when James Watt invented the centrifugal governor [Clark 68,Trinks 68] to control the speed of his steam engines. The problem was that when a load was applied the engine's speed fell, and when the load was removed the speed increased. So the centrifugal governor was used to maintain constant speed. The governor controlled

the opening of the valve feeding the steam to the engine. When an extra load is applied, the speed of the engine tends to fall which causes the governor to increase the opening of the valve allowing more steam pressure to the engine. The speed thus tends to rise, counteracting the original tendency for it to fall. Similarly if the load is removed the speed tends to increase, which causes the governer to close the valve and thus counteract any tendency for the speed to rise (for a more elaborate explanation of this control mechanism see [Atkinson 78, pages 1–6]).

So the purpose of the governer can be thought of as a device trying to maintain a constant relationship between the actual speed and the required speed of the engine. In fact it is the *difference* between the actual and the required speed which is used to control the opening and closing of the valve. This entire procedure can be modeled as a system with three basic building blocks; the *Error Detector*, the *Controller* and the *Output Element*. Figure 6.1 shows how these building blocks are used in constructing a simple feedback control system.

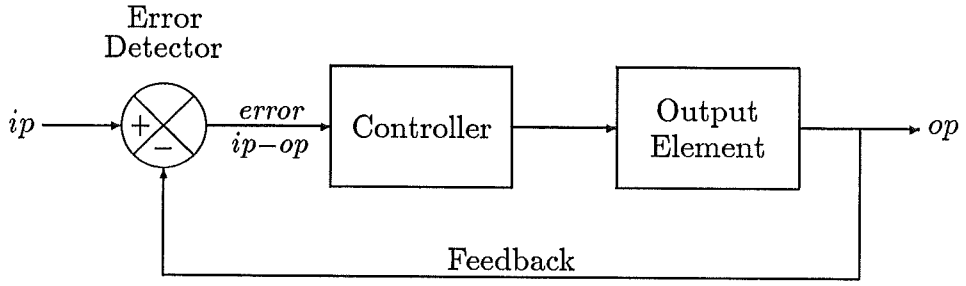


Figure 6.1: A simple feedback control system

The Error Detector generates an instantaneous difference between the input (*ip*) and the output (*op*) known as the *error*. In the case of the governer, the input would be the speed at which the engine is required to run, and the output would be the actual speed of the engine. The instantaneous error signal is then passed to the Controller. The purpose of the Controller is to translate the signals from the Error Detector into control signals that are more reliable than the instantaneous error. In the case of the governer this is accomplished by the momentum of the device which can be thought of as performing a weighted running average of the previous values. Finally the Output Element uses this control signal to modify the output in the appropriate way. In the above example this would include the equipment which opens and closes the valve, the boiler, the steam engine, and everything else necessary in make the engine run.

Like all feedback control systems, the phase-locked loop also conforms to this abstract model. The error detector is more commonly referred to as the Phase Detector. The output of this is a voltage V_e, being some function of the phase dif-

ference between its two input waveforms. This error voltage (V_e) is approximately proportional to the phase difference between the two waveforms when the phase difference is small. The controller is now a simple low-pass filter used to give a more reliable control signal (V_c) rather than the instantaneous phase difference. Finally the output element is replaced by a Voltage-Controlled Oscillator (VCO). This device generates a waveform whose frequency is controlled by its input voltage. For small input variations the rate of change of phase in the output waveform is proportional to its input voltage. An analogue PLL using these components is illustrated in figure 6.2. Note how closely this model of a phase locked loop mimics the abstract model of a feedback control system presented above.

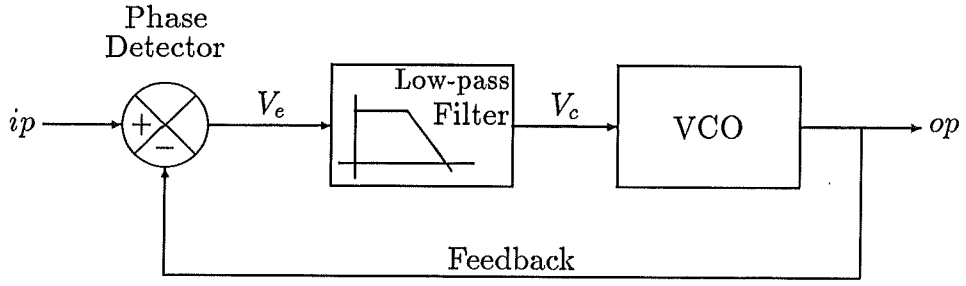


Figure 6.2: Block diagram of an analogue phase-locked loop

6.1.2 Digital Phase-Locked Loops

Early efforts on digital PLLs concentrated on partially replacing the analogue components with digital ones. Perhaps the earliest reported accounts in this direction are by Clark in 1949 [Clark49], and Hugenholtz in 1950 [Hugenholtz50]. Clark used a sample and hold circuit in the phase detector, and Hugenholtz described a method for discontinuously varying the output frequency. Surveys of DPLLs giving many references are presented in [Lindsey81,Rey82], where further details of the historical development of the various types of DPLLs can also be found.

In their survey of DPLLs [Lindsey81], Lindsey and Chie separate the various classes of loops into four categories, based on the design of the phase detector.

- *Flip Flop (FF)-DPLL*
  The phase error is derived from the duration between the set and the reset times of a flip-flop triggered by positive zero crossings of the input signal and the local clock.

- *Nyquist Rate (NR)-DPLL*
  The input signal is sampled at the Nyquist rate.

- *Zero Crossing (ZC)-DPLL*
  In this case the loop tries to sample at the zero crossings of the incoming signal.

- *Lead/Lag (LL)-DPLL*
  Here the phase detector determines at each cycle whether the input leads or lags the locally generated clock.

The one that is of interest here is the lead/lag DPLL. Only this will be described here in more detail. For details of the other types of DPLLs, see Lindsey and Chie's survey.

6.1.3 The Lead/Lag Digital Phase-Locked Loop

Beginning in 1961 at the University of Iowa, the lead/lag DPLL was employed in the Injun I-III satellite program. It was originally proposed and analysed by Cessna [Cessna 70a,Cessna 70b] in 1970. Later in 1972, Cessna and Levy [Cessna 72] presented the behaviour of this type of loop in white Gaussian noise conditions. They presented two realisations of the sequential filter (random walk filter) and made comparisons to a first-order linear loop. A block diagram of this type of loop is shown in figure 6.3.

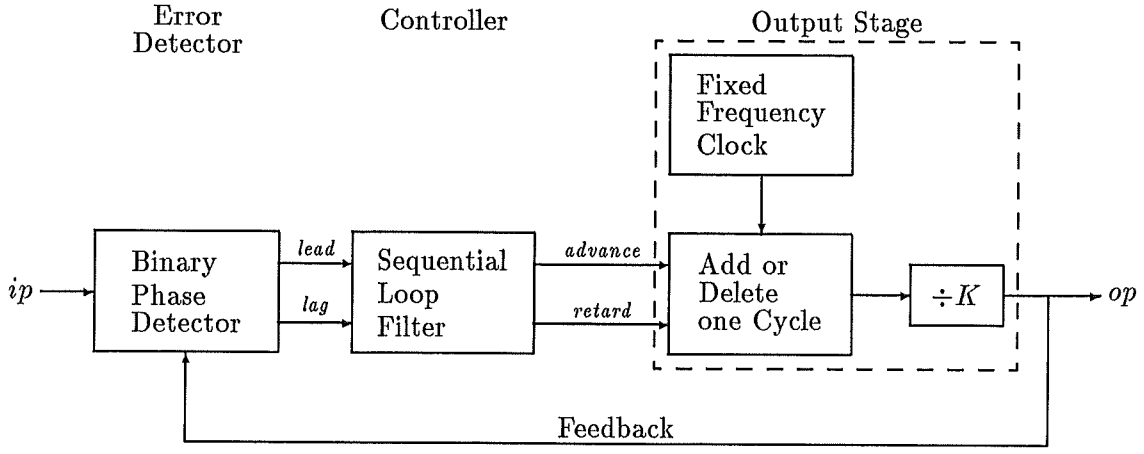


Figure 6.3: A lead/lag DPLL due to Cessna and Levy

The operation of this loop can be divided into three parts: 1) the phase detector, 2) the loop filter, and 3) the output stage. At each cycle the binary phase detector determines whether the input leads or lags the locally generated clock. This, in fact, is an extremely coarse error detector, and its result is indicated

as a pulse on either the lead or the lag output. These instantaneous error signals are then passed through the loop filter to generate advance/retard signals which are more accurate than the signals lead/lag. The loop filter accomplishes this translation by dealing with signals "lead" and "lag" in a statistical manner. These advance/retard signals are then passed to the output stage. Here a single pulse is added/subtracted to the local high frequency clock pulse train for each advance/retard signal, and then divided by K to generate the controlled output. In this way the loop output is matched to the incoming signal, phase for phase.

Note that the output can only be adjusted in phase by increments of $360/K°$. Also the time interval between such adjustments is limited to how often the correction signal can be generated by the sequential loop filter. A typical implementation of the sequential loop filter is the Random Walk Filter (RWF) as illustrated in figure 6.4. With this implementation, the maximum rate at which a correction signal can be generated is every N input transitions. The duration time between two such correction signals is maximised when the probability of the lead and the lag signals is equal.

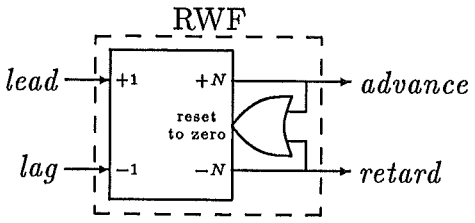


Figure 6.4: Implementation of the Random Walk Filter

The rate at which such a loop can track the phase of an input signal is limited by the size of the RWF, and the size of the phase adjustment step (K) in the output stage. Since the latter is limited by the maximum resolution of the phase error that can be tolerated by the specifications ($360/K°$), this reduces the problem of control to the RWF. If the size of N in the RWF is decreased, it results in shorter delay before any output is produced. This will have the desired effect of improving the tracking ability of the loop, but a byproduct of this is to introduce *jitter* on the output. Jitter is the tendency of the output to oscillate around the desired value.

Clearly these two required properties of DPLLs, namely low phase jitter and good tracking ability, are contrary to each other in this design. This problem has been tackled in the literature in two different ways. Firstly, by providing additional control in the loop filter, and secondly by adding additional control in the phase detector and the output stage.

6.1.3.1 Improving the Loop Filter

This technique involves introducing a sort of a "memory" into the RWF. This results in a second order loop filter type of behaviour. It was reported by Yamamoto and Mori [Yamamoto78] in 1978. If the RWF as presented above is referred to as the 0-reset-RWF, since it resets itself to the zero position when it overflows in either the positive or the negative direction, then the new filter as presented by Yamamoto and Mori is a RWF which has a variable reset position which they call the variable-reset-RWF(VR-RWF). It consists of two RWFs, one of which is the 0-reset-RWF controlling the resetter to the other. This is illustrated in figure 6.5.

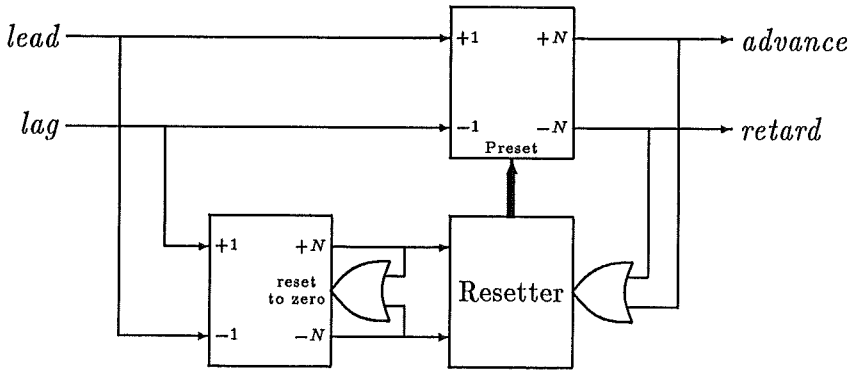


Figure 6.5: The variable-reset-RWF due to Yamamoto and Mori

The principle involved here is that if on average there are more lead signals than lag, then the reset value of the output RWF is increased to help it generate an advance signal with fewer input transitions. Similarly if there are more lag than lead signals on average, then the reset value of the output RWF is decreased which results in a retard signal being generated sooner. If the probability of both the lead and lag are equal then the whole system behaves like a 0-reset-RWF.

6.1.3.2 Improving the Phase-Detector and the Output Stage

This technique was reported in 1984 by Sandoz and Steenaart [Sandoz84] and involves having two additional binary phase detectors. These additional phase detectors are set up such that they sample the input signal shifted by $+\Delta$ and $-\Delta$ respectively. Now if the PLL is assumed to track the positive zero-crossings ($\int$ edges) of the input signal, then the sign of the additional phase detector results ($+1$ or -1) can be used to determine whether the PLL output is relatively close to the incoming $\int$ edges, or not. Figure 6.6 shows the four possible cases of the input signal with respect to the three phase detectors.

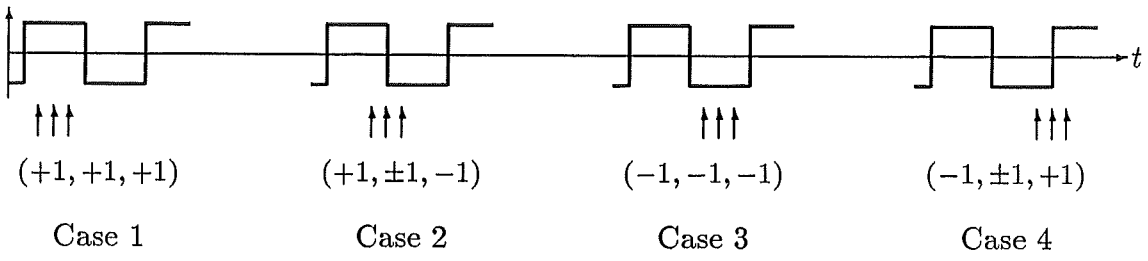


Figure 6.6: Cases of the input signal being sampled by 3 phase detectors

In cases 1, 2, and 3, the additional phase detector results are $(+1, +1)$, $(+1, -1)$ and $(-1, -1)$ respectively. These will all be interpreted by the system to mean that the output is not close to the incoming $\int$ edges, and so a larger step would be used in changing the output signal. In fact the only case under which the loop output will be considered to be "close" to the incoming $\int$ edges is in the last case where the additional phase detector results will be $(-1, +1)$; namely case 4 in figure 6.6. Here the phase correction in the PLL will be small resulting in higher resolution and hence lower steady state error on the output.

Both these techniques help improve the performance of the basic lead/lag DPLL as presented by Cessna and Levy. In the first case, the loop filter is dynamically modified in such a way as to result in the duration between the correction signals being reduced when the loop is some distance from the lock position. In the second case, instead of changing the loop filter, additional phase detectors are used to indicate when the loop output is relatively close to the lock position. While the loop is not close to the lock position, large correction steps are used in the output signal. This results in the loop output moving towards its target position somewhat faster than before. Note that both of these techniques could be used to advantage in a single design.

6.2 Overview of a New Design for a Lead/Lag DPLL

Given above was an informal review of digital phase-locked loops with particular emphasis on the lead/lag DPLL. Two techniques were described for improving the performance of this loop. In this section a more general technique is presented to further improve the design of such DPLLs. To motivate the new design, the problem of a PLL is first posed in a slightly different way.

The purpose of any control system can be summarised by saying that a certain controlled variable is to be reduced to zero. In the case of a PLL, this variable is the phase angle θ, between the input waveform and the loop output waveform (or some function of the loop output waveform). The input and the loop output signals can be represented as two vectors in a phase diagram, with the angle between them representing the phase difference between the two signals. This is illustrated in figure 6.7, with the input and the output vectors marked *ip* and *op* respectively, and with a phase difference of θ between them.

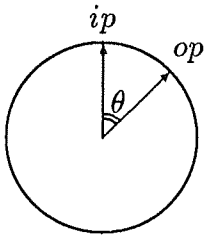


Figure 6.7: Phase relation between the input and the output of a PLL

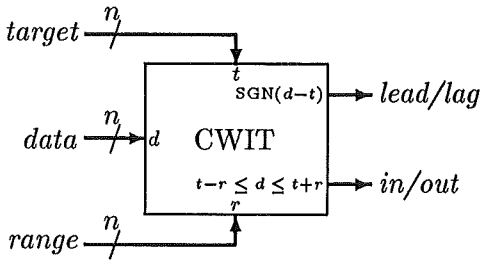
The workings of a PLL changes the phase angle θ by iteratively changing the output to reduce the error. This results in moving the *op* vector in the phase diagram, towards the reference vector *ip*. Note the similarity between this phase diagram (figure 6.7), and the familiar diagrams of the previous chapter, used in the proof outline of the window comparator (CWIT device). This CWIT device is in fact used in the design of a new type of a digital phase detector which is described next.

6.2.1 A Self-Modifying Digital Phase Detector

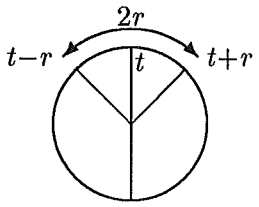
In this section, the design of a new type of a Digital Phase Detector (DPD) is outlined which uses the CWIT device. In fact, a slightly modified version of the CWIT device is used which has an extra output. This extra output is merely one of the internal signals brought to the outside, hence no extra circuitry is necessary. The top level view, and the behaviour of this new CWIT device can best be explained by using the circular diagrams of the previous chapter.

The top level view of this new CWIT device, and a modulo-n diagram illustrating the relationship between the various input and output signals is shown in Figure 6.8. If the target input (t) to the CWIT device is set to the required phase difference between the vectors *ip* and *op*, and the data input(d) is set to the actual measured phase difference θ, then what results is a device which has a similar behaviour to using three binary digital phase detectors. In fact the behaviour is

exactly the same as the three phase detector combination of section 6.1.3.2 if the range value(r) is set to Δ.



(a) The top level view



(b) The modulo-n diagram

Figure 6.8: The modified CWIT device

The behaviour of this new CWIT device is best explained by using the modulo-n diagram of figure 6.8b. Since it has two outputs, they can both be treated separately:

- The *lead/lag* output.
  This output is true if the data input is greater than the target value. Note that the concept of x being "greater than" y here means that x lies within the half circle on the right hand side of y. In the case of the CWIT device, d being greater than t would mean that the d value lies somewhere in the half circle on the right hand side of t. Or, as marked in the CWIT device, the lead/lag output is true if the sign of $d-t$ is positive. So if d is the actual measured phase difference, and t is the desired phase difference, then the lead/lag output gives an indication as to the direction of the error between these two values.

- The *in/out* output.
  The range input(r) to the CWIT device sets up a window as marked by the wedge of size $2r$ in figure 6.8b. The output in/out is true if the data input lies somewhere inside this wedge, and false otherwise.

The two outputs of this new modified CWIT device can jointly be used to indicate in which of the four carved out wedges of the modulo-n diagram, the data input (d) lies. However, in order to use the CWIT device as a phase detector, the various inputs must be digital words.

At this point it is worth noting that the output stages of the DPLLs presented above generally have a $\div K$ counter as their final device. This is used for generating both the output and the feedback signals. Consider using all the outputs of the

various stages in this counter as a "word" for feedback, instead of only using the most-significant-bit. Then, at the times of the incoming $\int$ edges, the value represented by this word will reflect the phase difference between the input and the output waveforms. Now by using a simple digital sample and hold circuit, an n-bit word becomes available which represents the phase error θ, where n is the number of stages in the $\div K$ counter.

This technique for generating the phase error, in conjunction with deploying the CWIT device as outlined above results in a new type of a Digital Phase Detector (DPD). This new DPD has two new characteristics not found in earlier designs; namely the ability to choose an arbitrary phase relation between the input and the feedback signals, and the ability to vary the size of the window around the target phase relation.

One interesting application of this new DPD would be to dynamically adjust the size of the window to the CWIT device, such that approximately 50% of the phase error samples lie within the window, and the other 50% outside. Then by using a few logic gates to combine the two outputs of the CWIT device, a new output can be computed which has three possible values: 0, +1 or −1. The value of this output can best be viewed in conjunction with the modulo-n diagram as shown in figure 6.9. The output value is 0 if the data input(d) lies within the window $2r$, otherwise it is either +1 or −1, depending on whether it is greater or less than t respectively. This three valued output is an instantaneous correction signal for that particular input edge. In general this signal will then be passed through a random walk filter to generate the more reliable signal advance/retard, which will then be used to make the actual correction in the phase of the loop output signal.

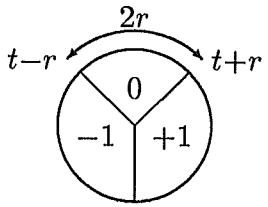
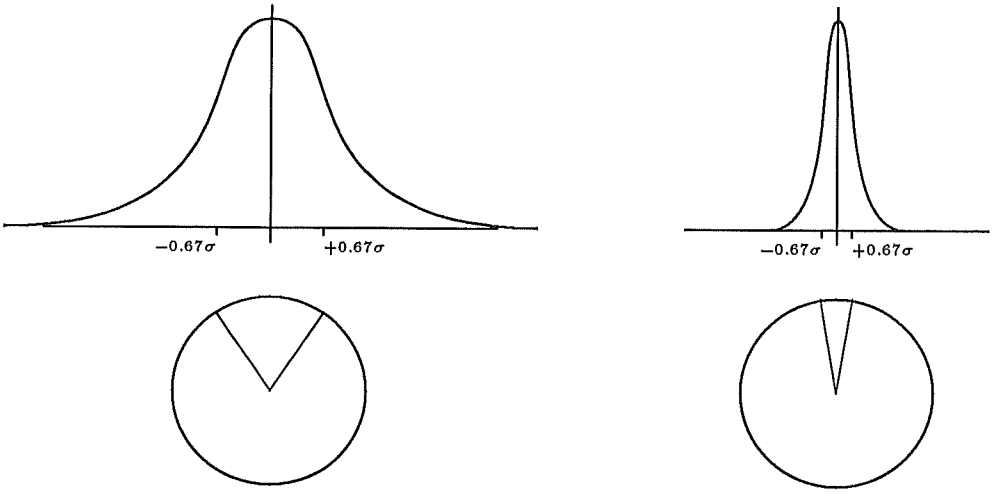


Figure 6.9: The three state output of the new digital phase detector

Note that this self modifying DPD has one very interesting characteristic: it adapts the size of the window dynamically depending on how noisy the input signal is. In any real system there will be some noise in the phase of the incoming signal. Assuming this noise has a Gaussian distribution, then the distribution of the phase error signal will be normally distributed about the actual phase difference. For a noisy input signal, this normal distribution curve will have a high standard deviation(σ), or in more informal terms the curve will tend to be relatively fat.

Similarly for a clean input signal, the standard deviation will be low, or informally the curve will tend to be relatively thin. This is illustrated in figure 6.10 where the size of the window of the CWIT device is also shown for a noisy and a clean input signal.



(a) Noisy input signal (b) Clean input signal

Figure 6.10: Normal distribution curves for noisy and clean input signals

This technique of dynamically adjusting the size of the window of the CWIT device also dynamically adapts the behaviour of the DPD, from one which is optimised to noisy environments with poor tracking ability, to one which is optimised to clean environments with good tracking ability.

6.2.2 The New Lead/Lag Digital Phase-Locked Loop

Proposed here is the design of a new digital phase locked-loop (DPLL) based on the principle of a three state, self modifying digital phase detector (DPD) as outlined in the previous section. The operation of this loop could be explained by the analysis of the three classical components that comprise any control system; namely, the phase detector, the loop filter and the output stage. Unfortunately the operation of each of these three components of this DPLL are not so clearly identifiable. Some of the components that comprise the phase detector are often shared by the loop filter and the output element. So the operations of this new DPLL is best explained by examining the general control strategy of the loop and tracing the state of the system for various input signals. A block diagram illustrating the new design of the DPLL is given in figure 6.11 on page 121.

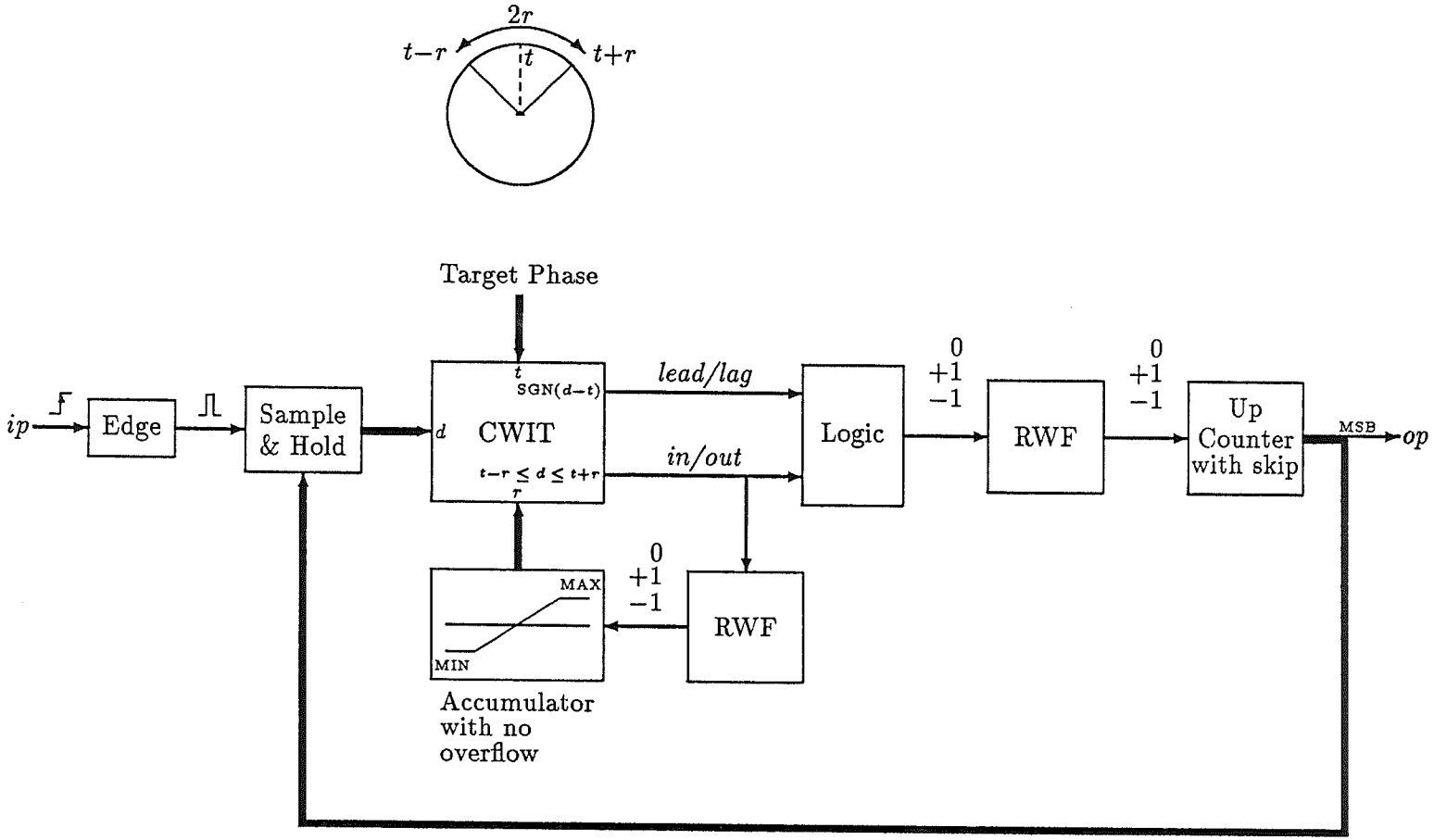


Figure 0.1: A new design for a DPLL using a self modifying Phase Detector

6.2.2.1 The Loop Control Strategy

From even a cursory glance at the block diagram of the new DPLL design (figure 6.11), it is apparent that there are two control loops and hence at least two means of controlling the behaviour of the system. In fact there are three things which influence the behaviour of this system, the third being the "target phase" input. In the design outlined here, the target phase input is merely a constant indicating the desired phase relation between the input and the loop output signal. If the target phase input is zero, for example, then the system will try to maintain zero phase error between the input and the loop output waveforms.

The inner loop forms a part of the DPD which has been described in section 6.2.1 above. Note that the in/out output of the CWIT device is used for feedback. This signal is passed through a random walk filter (RWF) to generate an "average" over time of the various in/out conditions. The output of the RWF is used to increment or decrement an accumulator. Finally the output of this accumulator is then used to control the size of the window (represented by r) of the CWIT device. This loop tries to maintain approximately 50% of the incoming error samples within the window($2r$) and the other 50% outside of it. Note that this does not provide any form of a control signal to the output stage directly, but merely varies the size of the window of the CWIT device. Another interesting point to note is that the accumulator output (or the range input to the CWIT device) provides some measure of the relationship between the system input(ip) and the output(op). If the system has high confidence with the op signal being closely matched to the ip signal, then the accumulator output is low (near to MIN), and if there is poor match then this output is high (near to MAX).

The other control mechanism actually provides a correction signal to the output stage. In figure 6.11 the output stage is marked as the "up counter with skip." This is exactly the same as the output stage of the lead/lag DPLL due to Cessna and Levy as shown earlier in figure 6.3 on page 113. The control mechanism used is also very similar to that of Cessna and Levy. The difference here is in the RWF. Instead of changing the state of the RWF for every incoming $\int$ edge, a change is only made if the error between the input and the feedback signal is outside the window of the CWIT device. Many other control mechanisms are possible. In particular, the ideas presented in sections 6.1.3.1 and 6.1.3.2 can easily be accommodated. The present design was chosen for the sake of simplicity but with full knowledge that the hooks provided in the design make it very flexible.

6.2.2.2 Loop Behaviour for Various Inputs

The DPLL design presented in figure 6.11 has in fact been designed all the way down to the CLIC gate level. A fairly minimal loop of this type was simulated as part of a fairly large project at Racal Research involving cellular telephones. The simulation environment[1] was designed to test the receiver for various noise conditions. This set up was used to generate the input waveform for the DPLL from a random data stream. The reason for using this particular set up was that it provided a fairly realistic means for modelling the noise on the input waveform to the DPLL. The noise in this case was introduced by having 200 oscillators evenly placed about the transmission frequency of the cellular-phone. All the outputs of these 200 oscillators were then summed together with the transmitted signal to generate the input to the receiver.

A number of simple simulations were performed with up to 2000 data bits. The first 25 data points for one of the more trivial simulation runs are illustrated in figure 6.12. This simulation starts with the DPLL output(op) $\sqcap$ edges being as far from the required position as possible. In this case the required position is such that the output $\sqcap$ edges be approximately 90° out of phase to the average of the incoming edges. Note that the loop quickly adjusts the output phase, so that by about the $20th$ data point, the output is approximately correct. Furthermore, the loop maintains this relation to incoming edges even if there are no input edges for some time, e.g. a short continuous stream of 1's.

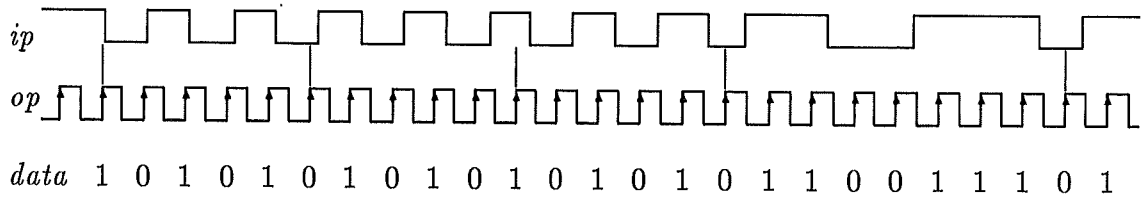


Figure 6.12: A simple simulation result for the new DPLL

The loop used for this simulation was extremely simple; with a fixed small window to the CWIT device, and no RWF in the outer loop to smooth the correction signal to the loop output stage. This set up was chosen to allow the loop to have an extremely fast response to input variations so that the operation of a PLL can be seen with very few data points. Though the loop is extremely fast, it still exhibits some jitter suppression characteristics due to the small window to the

[1]The simulation environment to test the receiver was set up by Mike Dumbrill and others of Racal Research. I did the additional work of fitting the new DPLL into this environment and analysing the results.

CWIT device. A more sophisticated loop would require significantly more data points to demonstrate this locking behaviour; but once locked it would exhibit performance significantly superior to the trivial case used for the simulation run here.

These various characteristics can be seen in the full theoretical analysis of this loop. Unfortunately such an analysis is beyond the scope of this thesis and will not be given here. However, it is recognised that a full formal analysis of this new DPLL design will have to be considered to determine its full potential. This is left for future work.

6.3 Formulating the Correctness Statement

In order to develop a correctness statement which fully characterises a particular DPLL design, it is necessary to define a number of predicates which capture all the ideas present in that design. In this section an extremely simple top level correctness statement for a DPLL is formulated. It does not take into account all of the complexities of the DPLL, because the techniques involved in formulating some of these ideas are very complex, as will become apparent in this treatment. So to begin, the DPLL in its most abstract form can be viewed as a device which has a single input and a single output as shown in figure 6.13.

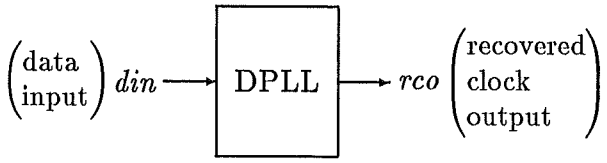


Figure 6.13: The most abstract view of a DPLL

Before giving the various clauses which constitute the correctness statement in a formal notation, they are first presented informally in English. This English description is separated into three parts, reflecting the way in which the correctness statement will be formulated.

1. *Period Correctness.*
   The DPLL generates a continuously running clock on its output irrespective of what the input is doing. The frequency of this varies within certain limits. This variation is controlled by the DPLL which is directly influenced by the input signal. In fact the control is applied in such a way as to synchronise the output waveform to the input, phase for phase.

2. *Lock Correctness.*
   If the output waveform of the DPLL is in "phase lock" with the loop input waveform, then, provided the input frequency remains within the working range of the DPLL, and provided the input does not change too quickly, the DPLL output waveform will continue to remain in "phase lock" with the input waveform.

3. *Capture Correctness.*
   No matter what the input has been doing in the past, but provided it begins to behave itself now and continue to do so for a certain period of time, then at the end of that period of time the DPLL output waveform will be in phase lock with the input waveform.

Even when trying to formulate the correctness statement of a DPLL in English as above, it is apparent that other terms, such as the concept of "phase lock," need to be defined first. In fact, when this English description is translated into a formal notation, the concepts which need to be clarified further become even more visible. However before going on to give the correctness statement in a formal notation, here is an informal definition of the term "phase lock:"

> **Phase Lock:** The DPLL input and output waveforms can be understood to be in "phase lock" if over a number of samples the input and the output waveforms continue to remain within the specified phase relation.

Rather than continuing with this English description and iteratively refining it until all subterms are clearly defined, it is better to start the formal presentation in parallel. This is because there are concepts which can more easily be presented in the formal notation using existing predicates, rather than battling to give a clear English description of them first. This is particularly so in the case of the formal statement for the first clause (Period Correctness) as presented in equation 6.2. The other two clauses of the correctness statement are captured in formal notation in equations 6.3 and 6.4.

So here is the first attempt at formulating the correctness statement for the first clause:

$$\mathsf{Period\_Correctness}\ a\ b\ rco\ =_{def}$$
$$\forall t.\ \mathsf{Edge}\ rco\ t\ \supset\ \exists t'.\ \begin{pmatrix}\mathsf{Next}\ t\ t'\ (\mathsf{Edge}\ rco)\ \wedge\\ a \leq (t'-t) \leq b\end{pmatrix} \tag{6.1}$$

This says that the time interval between all consecutive edges on the line *rco* is between a and b. Also, since the correctness statement is in the form of an implication, it is possible to satisfy this by a device with an output which *never* has any edges on it. So to exclude such devices from this correctness statement, it is necessary to add an extra clause to it. This new clause simply says that there is always at least one edge on the line *rco*. This results in the following equation for Period_Correctness.

$$\begin{array}{l}\mathsf{Period\_Correctness}\ a\ b\ rco\ =_{def}\\ \quad (\exists t.\ \mathsf{Edge}\ rco\ t)\ \wedge\\ \quad \forall t.\ \mathsf{Edge}\ rco\ t\ \supset\ \exists t'.\ \begin{pmatrix}\mathsf{Next}\ t\ t'\ (\mathsf{Edge}\ rco)\ \wedge\\ a \leq (t'-t) \leq b\end{pmatrix}\end{array} \tag{6.2}$$

In the above correctness statements two new predicates are used; namely, Next and Edge. These can now be defined separately as follows:

$$\begin{array}{lll}\mathsf{Next}\ t_1\ t_2\ f & =_{def} & t_1<t_2\ \wedge\\ & & f(t_2)\ \wedge\\ & & \forall t.\ t_1<t<t_2\ \supset\ \sim f(t)\end{array}$$

$$\mathsf{Edge}\ f\ t\ =_{def}\ \mathsf{Rise}\ f\ t\ \vee\ \mathsf{Fall}\ f\ t$$

Where

$$\begin{array}{lll}\mathsf{Rise}\ f\ t & =_{def} & f(t)=\mathsf{Lo}\ \wedge\ f(t+1)=\mathsf{Hi}\\ \mathsf{Fall}\ f\ t & =_{def} & f(t)=\mathsf{Hi}\ \wedge\ f(t+1)=\mathsf{Lo}\end{array}$$

Note how in this formulation of the first clause of the correctness statement, various concepts are iteratively refined until there is no ambiguity. Similar techniques are employed in generating the other two clauses of the correctness statement. These are given below without formal statements for the various predicates used in defining them.

$$\begin{array}{l}\mathsf{Lock\_Correctness}\ din\ rco\ =_{def}\\ \quad \forall n.\ \begin{pmatrix}\mathsf{Phase\_Lock}\ \ din\ rco\ n\ \wedge\\ \mathsf{Small\_Change}\ din\ rco\ n\end{pmatrix}\ \supset\ \mathsf{Phase\_Lock}\ din\ rco\ (n+1)\end{array} \tag{6.3}$$

$$\begin{array}{l}\mathsf{Capture\_Correctness}\ n\ din\ rco\ =_{def}\\ \quad \forall m.\ \mathsf{Stable\_For}\ n\ din\ m\ \supset\ \exists p.\ \begin{pmatrix}p \leq n\ \ \wedge\\ \mathsf{Phase\_Lock}\ din\ rco\ (m+p)\end{pmatrix}\end{array} \tag{6.4}$$

For the moment it suffices to say that these two equations (6.3 and 6.4) reflect the informal English description given earlier. The three new predicates, namely Phase_Lock, Small_Change and Stable_For, are necessary to reflect the way in which the English description was given. These predicates will have to be given a clear semantics, together with formal definitions which fully characterise them, just as the predicates Next and Edge were defined for equation 6.2 (the period correctness clause). Following the same pattern as before, given below is the informal English description for each of these predicates.

- Phase_Lock *din rco n*

  The term "phase lock" was defined earlier as a relation between two waveforms. The predicate here however, has three arguments. The first two are the waveforms which may or may not be in "phase lock." The third argument n actually states when the two waveforms *din* and *rco* are in "phase lock," such that the predicate Phase_Lock is true. However for the two waveforms to be in phase lock requires that the relation between them be monitored for a certain number of samples n. So the following question arises:

  What is n?

  This could be a measure of time, or a measure of the number of input edges. It can be argued that it should not be time, since the ticks of time bear little relation to the actual edges of the input or the output waveforms. Recall that it is on the edges of the input signal that the DPLL makes any changes to the internal state and the output waveform. So this leaves behind two options; namely, n being a direct measure of the number of the input edges or a function of a group of input edges.

  It is preferable to use small groups of input edges as a single packet, and let n be a count over these packets. The reason for this is that the DPLL makes changes to the output waveform, after observing the input waveform for a certain number of edges. So if the packet size can be made to be identical to the DPLL correction interval, then perhaps the analysis and the proofs of correctness might be simplified. So now the system behaviour at n and $n+1$ will be different depending on how the DPLL made the correction to the output waveform.

- Small_Change *din rco n*

  What this predicate is designed to capture is that the change in phase between the two waveforms, *din* and *rco*, over two sets of samples, n and $n+1$, is less than a certain amount. This amount is assumed to be a constant and

is embodied in the definition of the predicate. However in a more general case, this predicate would have an extra argument indicating the size of the change required.

- Stable_For n *din* m
  This should be read as "the waveform *din* is stable for n units starting at m." These units are in fact packets of edges on the input waveform, as described for the Phase_Lock predicate above. The Stable_For predicate captures the notion that the signal on *din* is not wildly varying, but has stabilised in phase and hence in frequency. Further this frequency is within the working range of the DPLL. Again this information regarding the frequency should be provided as an argument to this predicate for it to be more general. But for now it has been assumed as a constant and is embodied in the definition of this predicate.

Finally, with these three clauses formally captured as predicates (equations 6.2, 6.3 and 6.4), a single new predicate can now be defined for the DPLL which makes use of all of these as follows:

$$\begin{aligned} &\mathsf{DPLL\_Correctness}\ a\ b\ n\ din\ rco \quad =_{def} \\ &\quad \mathsf{Period\_Correctness}\ a\ b\ rco \quad \wedge \\ &\quad \mathsf{Lock\_Correctness}\ din\ rco \quad \wedge \\ &\quad \mathsf{Capture\_Correctness}\ n\ din\ rco \end{aligned} \tag{6.5}$$

With a single predicate in place which captures the various aspects of the DPLL behaviour as in equation 6.5, the form of the correctness statement can now be established. So here is a naive first attempt at writing the correctness statement for the DPLL system using the DPLL_Correctness predicate.

$$\mathsf{DPLL\_Imp}\ din\ rco \supset \mathsf{DPLL\_Correctness}\ a\ b\ n\ din\ rco$$

Where the predicate DPLL_Imp represents the implementation of the DPLL device as pictured in figure 6.11 on page 121.

Note that this statement is truly naive since it says little about the conditions that may be necessary on the input and output lines. It does however have the correct general form which can be expanded further. Recall that if the DPLL device is built using the CLIC design style, then there will be a clock present which will have to be connected to the DPLL device implementation for it to function

correctly. There may also be some set up conditions necessary on the input and output lines of this device. All these things can be added to the form of the above correctness statement resulting in the following more refined equation.

$$\left(\begin{array}{ll} \mathsf{Ip\_Cond}\ \mathit{din} & \wedge \\ \mathsf{Op\_Cond}\ \mathit{rco} & \wedge \\ \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{DPLL\_Imp}\ \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ \mathit{din}\ \mathit{rco} & \end{array}\right) \supset \mathsf{DPLL\_Correctness}\ a\ b\ n\ \mathit{din}\ \mathit{rco} \quad (6.6)$$

Here the predicates Ip_Cond and Op_Cond reflect the set up conditions on the input and output lines *din* and *rco* respectively.

6.4 Formulating the Proof Plan

Just as the correctness statement was split up into three parts, so the proof too can be split into three similar parts. The three clauses which need to be proved to establish the correctness statement for the DPLL device can be stated as follows:

$$\left(\begin{array}{ll} \mathsf{Ip\_Cond}\ \mathit{din} & \wedge \\ \mathsf{Op\_Cond}\ \mathit{rco} & \wedge \\ \mathsf{Clock}(\phi_1, \overline{\phi}_1, \phi_2, \overline{\phi}_2) & \wedge \\ \mathsf{DPLL\_Imp}\ \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ \mathit{din}\ \mathit{rco} & \end{array}\right) \supset \mathsf{Period\_Correctness}\ a\ b\ \mathit{rco} \quad (6.7)$$

$$\left(\begin{array}{l} \vdots \\ \mathsf{DPLL\_Imp}\ \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ \mathit{din}\ \mathit{rco} \end{array}\right) \supset \mathsf{Lock\_Correctness}\ \mathit{din}\ \mathit{rco} \quad (6.8)$$

$$\left(\begin{array}{l} \vdots \\ \mathsf{DPLL\_Imp}\ \phi_1\ \overline{\phi}_1\ \phi_2\ \overline{\phi}_2\ \mathit{din}\ \mathit{rco} \end{array}\right) \supset \mathsf{Capture\_Correctness}\ n\ \mathit{din}\ \mathit{rco} \quad (6.9)$$

The proof of the first of these clauses (equation 6.7) is in fact fairly straightforward. Note that the last stage in the DPLL design is a device called "Up Counter with skip" (see figure 6.11 on page 121). From the implementation of this device, coupled with the fact that the clock driving it is continuously running, it can easily be proved that the edge to edge time on the output is always between 2^x-1 and 2^x+1, where x is the number of stages in the counter and the count sequence is simple binary. Note also that the rest of the devices in the implementation of the

DPLL are not used in this proof. In fact the proof is done as three separate cases based on the three possible values that can occur on the input line of this counter. Finally all that remains to be checked for the proof to be finished is that $a \leq 2^x - 1$ and $2^x + 1 \leq b$ to satisfy the predicate Period_Correctness.

The proof of the next two clauses however requires significantly different proof techniques than those which have been used up to now. In order to prove these clauses, a "measure" needs to be generated indicating how close in phase the output waveform is to the input waveform. Some function of the error signal indicating the phase difference between the input and the output waveforms could form the basis for evaluating this "measure ($\mathcal{E}$)." Based on the fact that the error signal dictates the behaviour of the DPLL system, a new equation can then be developed which uses $\mathcal{E}$ instead of the error signal. This equation in its most abstract form could be formulated to state that the next value of $\mathcal{E}$ is a function(P) of the previous value of $\mathcal{E}$, where the function P embodies the behaviour of the DPLL system. This equation is referred to in the literature as the difference equation for the loop, and its abstract form can be stated as follows:

$$\mathcal{E}_{t+1} = \mathsf{P}(\mathcal{E}_t) \tag{6.10}$$

In fact the behaviour of the DPLL system can be thought of as a special purpose device which tries to find a solution to equation 6.10 for various input conditions. The convergence of this equation to a steady state value for various input conditions also indicates whether the DPLL will lock. If such a solution exists, and is within the scope of the DPLL system to find it, then, the DPLL system will exhibit a steady state response settling to a value $\mathcal{E}_*$ as a solution for this equation, i.e. $\lim_{t\to\infty} \mathcal{E}_t = \mathcal{E}_*$. This can be stated in the same form as equation 6.10 as follows:

$$\mathcal{E}_* = \mathsf{P}(\mathcal{E}_*) \tag{6.11}$$

The value $\mathcal{E}_*$ satisfying equation 6.11 is called a *fixed point* [Bird 76, chapter 7] of P. The conditions under which such convergence occurs can be used as the basis for showing the correctness of the third clause (equation 6.9). This difference equation can also be used to show the correctness of the second clause, by analysing the the range over which the system will continue to maintain a steady state solution if the input phase is swept. Indeed at this point it seems questionable whether the correctness statements captures the required aspects of the behaviour of the DPLL. Other more interesting aspects, such as the time to settle to a steady

state value, can now be derived from this difference equation for various input conditions.

The only other work in this area using the fixed point technique for finding a solution to the DPLL system is that done by Osborne [Osborne 80a, Osborne 80b]. In [Osborne 80a] the fixed point technique for analysing a first order (CW)DPLL is presented. This work is extended to the analysis of the second and third order loops in [Osborne 80b]. It is hoped that similar techniques can be used for the analysis of the self modifying lead/lag type of DPLL presented here. Further research work in this area will also lead to improvements in the correctness statement and the proof.

6.5 Summary

In this chapter a brief overview of control systems was given with particular emphasis on digital phase-locked loops (DPLLs). Then a new design for a lead/lag digital phase-locked loop was presented. This new DPLL is adaptive and can dynamically adjust its characteristics. Depending on how noisy the input signal is, the new DPLL can change its characteristics from one which is optimised to noisy environments with poor tracking ability, to one which is optimised to clean environments with good tracking ability. The particular DPLL design described in this chapter is also considerably more flexible, because the phase relation between the input and the output waveforms can be easily programmed. In fact this control mechanism can also be used to do simple tests on the new DPLL. For example, suddenly changing the "target phase" input (i.e. the required phase relation) from 0° to 90° has the effect of the output tracing out the step response of the DPLL system. Also by varying this target phase relation in other ways, different characteristics of the new DPLL can be tested. To demonstrate the viability of this new DPLL design, it was simulated with various data patterns as part of a cellular telephone receiver.

Also in this chapter a correctness statement for a DPLL was developed as three separate clauses capturing the three basic properties of a DPLL, namely:

1. *Period Correctness.*
   The DPLL generates a continuously running clock on its output. The period of this is dictated within certain limits by the behaviour of the input.

2. *Lock Correctness.*
   If the output waveform of the DPLL is in "phase lock" with the loop input

waveform, then, provided the input frequency remains within the working range of the DPLL, and provided the input does not change too fast, then the DPLL output waveform will continue to remain in "phase lock" with the input waveform.

3. *Capture Correctness.*
   No matter what the input has been doing in the past, but provided it begins to behave itself now and continue to do so for a certain period of time, then at the end of that period of time the DPLL output waveform will be in phase lock with the input waveform.

A proof plan was then presented outlining how this correctness statement could be derived. The proof of the first clause is shown to be relatively straightforward, but not so for the other two clauses. Finally a technique was briefly outlined which transforms the formal definitions of the DPLL into a form known as the difference equation. Solutions to this difference equation are then used as the basis for showing the correctness of these two clauses. One technique for finding such solutions, briefly described here, is the use of fixed points. The conditions and the range over which convergence to a fixed point occurs can be used as the basis for showing the correctness of these two clauses.

Chapter 7

Concluding Remarks

7.1 Summary of Work Done

A technique has been presented to show that the informal design rules of an integrated circuit design style can be formalised in higher-order logic. The design style used as the basis of this work was the CLIC design style. First an informal description of this design style was given to highlight the various aspects which need to be formalised. Then a complete list of the rules for designing integrated circuits using this design style was compiled. Based on this description, the primitive components for building devices were then defined in logic, together with the axiomatisation of the four-valued algebra for signals. The specific rules of the design style were then formalised in logic. An important predicate defined to aid in this respect was the "Well Behaved (Wb)" predicate. This captures the constraints that must be imposed on the inputs of CLIC gates to ensure that they operate correctly, and also expresses the behaviour of well-defined outputs. With the aid of this predicate Wb, a method was presented to derive the correctness statements for the entire class of CLIC gates. This method relies on the fact that the form of the correctness statement developed is uniform across the entire range, from primitive gates to large and complex circuits.

The use of these formal techniques were then demonstrated through a number of worked examples, ranging from an exclusive-or gate to a random walk filter. In each case the form of the derived correctness statement was identical. This is an important factor in using such formal design techniques. If the statement of correctness has the same general form at each level, then arbitrary mixing of complex and trivial devices can be done with ease. This is often the case in designing large circuits, where the outputs of large macro blocks are connected to the inputs of primitive gates and vice versa.

Two further case studies were presented, illustrating how formal techniques help in the specification and the design of devices. The first of these is a window comparator device, which is useful in the design of content addressable memories, associative memories, and window addressable memories. Based on a simple top level specification for this device, an informal proof of correctness was outlined illustrating how such formal techniques can help improve and sharpen the final specifications.

The last case study presented shows a new design for a digital phase-locked loop (DPLL). This has been designed down to the gate level using the CLIC design style, and has been described and simulated using ELLA. Two of the major components used in the design of this device form the basis of the previous two case studies. So while the correctness statements for some of the components of this device were stated without proof, others were formally derived. The correctness statement for the phase-locked loop was then formulated by three separate clauses, capturing the following properties:

1. *Period Correctness.*
   The DPLL generates a continuously running clock on its output. The period of this is dictated within certain limits by the behaviour of the input.

2. *Lock Correctness.*
   If the output waveform of the DPLL is in "phase lock" with the loop input waveform, then, provided the input frequency remains within the working range of the DPLL, and provided the input does not change too fast, the DPLL output waveform will continue to remain in "phase lock" with the input waveform.

3. *Capture Correctness.*
   No matter what the input has been doing in the past, but provided it begins to behave itself now and continues to do so for a certain period of time, then at the end of that period of time the DPLL output waveform will be in phase lock with the input waveform.

A proof plan was presented outlining how these clauses could be satisfied. Finally a technique was briefly outlined which transforms the formal definitions of the DPLL into a form known as the difference equation. Solutions to this difference equation could then be used as the basis for showing the correctness of these clauses. One technique for finding such solutions, which was briefly described, is the use of fixed points. The conditions and the range over which convergence to a fixed point occurs can be used as the basis for showing the correctness of these clauses.

7.2 Discussion and Future Work

A number of difficulties were encountered in doing some of this work. These are discussed below, together with some ideas for further research. Also outlined in this section is how the work done here can be related to other fields.

The greatest problem in formalising the CLIC design style was to do with the rather simple models of the primitives used, particularly the transistors. The proofs of correctness of primitive CLIC gates are based on a simple notion of whether two nodes are linked or not linked. The transistor model used, however, does not directly reflect this fact. The solution used was satisfactory, but in general a more accurate model of the primitive devices to reality is needed. A model based on the link relation being the primitive would be ideal, but it is not clear how to adopt this into the present framework. The work of Winskel [Winskel 88] is particularly relevant, where the simulation model used in [Bryant 81] is embedded in logic. But this is not yet developed to the point where it is usable for the dynamic behaviour of circuits. Another approach would be to formally show that the simpler models are adequate in the environment in which they are used. This requires relating models at one level to models at another [Winskel 87].

The work presented here using the "Well Behaved (Wb)" predicate to check which CLIC gates can be connected together, can be viewed as an elaborate way of doing type-checking. An interesting area for future research would be to formally specify and derive types for the external ports of devices in the particular design style. So now, checking that the rules of a design style are obeyed is reduced to doing simple type-checking on the ports. For example, in the case of the CLIC design style, the ports of a device which satisfy "Wb ϕ_1" could be viewed as ports which have type Wb_{ϕ_1}. The type definition package for the HOL system recently developed by Melham [Melham 88b] should be a useful tool in this respect.

The other area where considerable difficulty is encountered is in writing formal specifications for large and complex systems. This was clearly illustrated by the last case study, where a concise top level statement of correctness for the digital phase-locked loop was not formulated, but only an outline was presented. The difficulty here lies in trying to capture intuitive notions about the system, without making the specifications so complex that they no longer reflect its intended behaviour. So a number of new predicates are needed, carefully chosen and defined to make the specification task easier. These predicates must embody key ideas specific to the particular application. For example, designers often use different notations for writing specifications for specialist areas, such as Digital Filters, Computer Architectures, Control Systems etc. There is a good reason why a par-

ticular notation is used in each case, and forcing the designer to reformulate his ideas in a single notation may be unacceptable. If specifications *must* be written in a unified notation, then providing the designer with *application specific phrases* to capture his intuitions would greatly ease the task of writing specifications.

In order to illustrate the usefulness and difficulty of this task, consider what it means for a Phase-Locked Loop system to *lock* on to an input signal. This is not merely a simple relation stating that the output is equal to the input. Part of what it means for a system to be "in lock," is, "if the input *frequency* changes within certain constraints then the output will *track* the input." Note the introduction of two new concepts, namely, "change in input frequency," and "output tracking the input." Concepts such as these need to be formulated as predicates in higher-order logic, so that they can be used in writing concise formal specifications.

Some of the more directly related areas where these ideas can be applied include other integrated circuit design styles, such as "Hot-Clocks NMOS" [Seitz 84] and "Zipper CMOS" [Lee 86]. Indeed, the various concepts developed here should be applicable to any system which uses building blocks with interconnection rules. Customised predicates can be developed for the appropriate application, just as the Wb predicate was developed for the CLIC design style. An interesting new application would be to apply these ideas to the structure of pipelines in integrated circuits. In fact, circuits built in the CLIC design can abstractly be viewed as pipelines, since all evaluations in the CLIC design style propagate down chains.

From a more general perspective, the work presented in this thesis has concentrated on roughly the middle of the integrated circuit design process. If a fully formal integrated circuit design methodology is to be developed—going from the early stages of concept development to the final components—then what has been presented here should aid the design, and (in particular) the specification and verification process. These phases in the design process are still some distance from the final manufacturing stage. As the verification technology matures and permeates both top level specifications and the lower level descriptions of circuits, formally verified devices in safety-critical applications should become a reality.

Bibliography

[Atkinson 78] P. Atkinson. *Feedback Control Theory for Engineers.* Heineman Educational Books Ltd, 1978.

[Bardeen 48] J. Bardeen and W. H. Brattin. The transistor, a semiconductor triod. *Physical Review*, 230–1, July 1948. The announcement also acknowledged the help of William Schockley and others at Bell Laboratories.

[Barrow 84] H. G. Barrow. Proving the correctness of digital hardware designs. *VLSI Design*, V(7):64–77, July 1984.

[Bird 76] R. Bird. *Programs and Machines: An Introduction to the Theory of Computation.* Wiley, 1976.

[Birkhoff 48] Garrett Birkhoff. *Lattice Theory – Revised Edition.* Volume XXV, American Mathematical Society Colloquium Publications, 531 West 116th Street, New York City, 1948.

[Boole 54] G. Boole. *An Investigation of the Laws of Thought, on which are Founded the Mathematical Theories of Logic and Probability (1849).* New York: Dover, (reprint), 1954.

[Boyer 79] R. Boyer and J. Moore. *A Computational Logic.* Academic Press, 1979.

[Braun 78] Ernest Braun and Stuart MacDonald. *Revolution in Miniature.* Cambridge University Press, Cambridge, England, 1978.

[Bryant 81] Randal Everitt Bryant. *A Switch-Level Simulation Model for Integrated Logic Circuits.* PhD thesis, Laboratory for Computer Science, MIT, Massachusetts, March 1981. Available as Technical Report MIT/-LCS/TR-259.

[Camilleri87] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. In *Proceedings of the IFIP WG 10.2 Working Conference: From H.D.L. Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, editor, pages 43–67, North-Holland, Amsterdam, 1987. Conference held in Grenoble, September 1986.

[Cessna70a] James R. Cessna. *Steady State and Transient Analysis of a Class of Digital Phase-Locked Loops Employing Coarse Amplitude Quantization and Sequential Filters.* PhD thesis, Department of Electrical Engineering, University of Iowa, Iowa City, January 1970.

[Cessna70b] James R. Cessna. Steady state and transient analysis of a bit-synchronization phase-locked loop. In *Proceedings of the IEEE International Conference on Communications*, June 1970.

[Cessna72] James R. Cessna and Donald M. Levy. Phase noise and transient times for a binary quantized digital phase-locked loop in white Gaussian noise. *IEEE Transactions on Communications*, COM-20(2):94–104, April 1972.

[Church40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[Clark49] E. L. Clark. Automatic frequency phase control of television sweep circuits. *Proceeding of the IRE*, 37:497, 1949.

[Clark68] Maxwell J. Clark. On governers. *Proceedings of the Royal Society*, London, 16:270–283, 1868.

[Cohn88] Avera Cohn. A proof of correctness of the Viper microprocessor: the first level. In *VLSI Specification, Verification and Synthesis*, Graham Birtwistle and P. A. Subrahmanyam, editors, pages 27–71, Kluwer Academic Publishers, 1988. Proceedings of the conference held on 12-16 January 1987 at Calgary, Canada.

[Cullyer88] W. J. Cullyer. Implementing safety-critical systems: the Viper microprocessor. In *VLSI Specification, Verification and Synthesis*, Graham Birtwistle and P. A. Subrahmanyam, editors, pages 1–25, Kluwer Academic Publishers, 1988. Proceedings of the conference held on 12-16 January 1987 at Calgary, Canada.

[Darby86] B. J. Darby and D. W. R. Orton. Structured approaches to design. *IEE Proceedings*, 133-E(3):123–126, May 1986.

[Dhingra88] I. S. Dhingra. Formal validation of an integrated circuit design style. In *VLSI Specification, Verification and Synthesis*, Graham Birtwistle and P. A. Subrahmanyam, editors, pages 293–321, Kluwer Academic Publishers, 1988. Proceedings of the conference held on 12-16 January 1987 at Calgary, Canada.

[Goncalves82] Nelson F. Goncalves and Hugo J. de Man. n-p-CMOS: a racefree dynamic CMOS technique for pipelined logic structures. In *ESSCIRC Digest of Technical Papers*, pages 141–144, September 1982.

[Goncalves83] Nelson F. Goncalves and Hugo J. de Man. NORA: a racefree dynamic CMOS technique for pipelined logic structures. *IEEE Journal of Solid-State Circuits*, SC-18(3):261–266, June 1983.

[Gordon79] M. J. C. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1979.

[Gordon82] M. J. C. Gordon. *A Model of Register Transfer Systems with Applications to Microcode and VLSI correctness.* Technical Report CSR-82-81, Department of Computer Science, University of Edinburgh, U.K., May 1982.

[Gordon83a] M. J. C. Gordon. *LCF-LSM: A System for Specifying and Verifying Hardware.* Technical Report 41, Computer Laboratory, University of Cambridge, U.K., 1983.

[Gordon83b] M. J. C. Gordon. *Proving a Computer Correct with the LCF-LSM Hardware Verification System.* Technical Report 42, Computer Laboratory, University of Cambridge, U.K., 1983.

[Gordon85a] M. J. C. Gordon and J. Herbert. *A Formal Hardware Verification Methodology and its Application to a Network Interface Chip.* Technical Report 66, Computer Laboratory, University of Cambridge, U.K., 1985.

[Gordon85b] M. J. C. Gordon. *HOL: A Machine Oriented Formulation of Higher-Order Logic.* Technical Report 68, Computer Laboratory, University of Cambridge, U.K., 1985.

[Gordon85c] M. J. C. Gordon. *Hardware Verification by Formal Proof.* Technical Report 74, Computer Laboratory, University of Cambridge, U.K., August 1985.

[Gordon 86] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Conference on ,VLSI*, G. J. Milne and P. A. Subrahmanyam, editors, pages 153–177, North-Holland, 1986.

[Gordon 88] Michael J. C. Gordon. HOL: a proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*, Graham Birtwistle and P. A. Subrahmanyam, editors, pages 73–128, Kluwer Academic Publishers, 1988. Proceedings of the conference held on 12-16 January 1987 at Calgary, Canada.

[Hale 88b] Roger W. S. Hale. *Forthcoming.* PhD thesis, Computer Laboratory, University of Cambridge, England, 1988.

[Hanna 86a] F. K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEEE Proceedings*, 133-E(5):242–254, September 1986.

[Hanna 86b] F. K. Hanna and N. Daeche. Specification and verification using higher-order logic: a case study. In *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Conference on ,VLSI*, G. J. Milne and P. A. Subrahmanyam, editors, pages 179–213, North-Holland, 1986.

[Herbert 86] J. M. J. Herbert. *Applications of Formal Methods to Digital Systems Design.* PhD thesis, Computer Laboratory, University of Cambridge, England, December 1986.

[Hugenholtz 50] E. H. Hugenholtz. The application of impulse-governed oscillators in aircraft transmitters. *Communication News*, 11(13), May 1950.

[Hunt 87] W. A. Hunt. The mechanical verification of a microprocessor design. In *Proceedings of the IFIP WG 10.2 Working Conference: From H.D.L. Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, editor, pages 89–129, North-Holland, Amsterdam, 1987. Conference held in Grenoble, September 1986.

[IRE 56] IRE Standards Committee. Standards on terminology for feedback control systems. *Proceedings of the IRE*, 107–109, January 1956.

[Joyce 88] Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. In *VLSI Specification, Verification and Synthesis*, Graham Birtwistle and P. A. Subrahmanyam, editors, pages 129–157, Kluwer Academic Publishers, 1988. Proceedings of the conference held on 12-16 January 1987 at Calgary, Canada.

[Krambeck 82] R. H. Krambeck, Charles M. Lee, and Hung-Fai Stephen Law. High-speed compact circuits with CMOS. *IEEE Journal of Solid-State Circuits*, SC-17(3):614–619, June 1982.
This is also known as DOMINO Logic.

[Lee 86] Charles M. Lee and Ellen W. Szeto. Zipper CMOS. *IEEE Circuits and Devices Magazine*, 2(3):10–16, May 1986.

[Leeser 87] M. E. Leeser. *Reasoning about the Function and Timing of Integrated Circuits with Prolog and Temporal Logic.* PhD thesis, Computer Laboratory, University of Cambridge, England, December 1987.

[Leisenring 69] A. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol.* Macdonald and Co. Ltd., London, 1969.

[Lindsey 81] William C. Lindsey and Chak Ming Chie. A survey of digital phase-locked loops. *Proceedings of the IEEE*, 69(4):410–431, April 1981.

[Lindsey 82] William C. Lindsey and Chak Ming Chie. Editorial note: special issue on phase-locked loops. *IEEE Transactions on Communications*, COM-30(10):2221–2223, October 1982.

[Mealy 55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technology Journal*, 34(5):1045–1080, 1955.

[Melham 88a] Thomas F. Melham. Abstraction mechanisms for hardware verification. In *VLSI Specification, Verification and Synthesis*, Graham Birtwistle and P. A. Subrahmanyam, editors, pages 267–291, Kluwer Academic Publishers, 1988. Proceedings of the conference held on 12-16 January 1987 at Calgary, Canada.

[Melham 88b] Thomas F. Melham. *Forthcoming.* PhD thesis, Computer Laboratory, University of Cambridge, England, 1988.

[Milne 79] G. J. Milne and R. Milner. Concurrent processes and their syntax. *Journal of the ACM*, 26(2), April 1979.

[Milne 83a] G. J. Milne. CIRCAL: a calculus for circuit description. *Integration*, 1(2–3):121–160, October 1983.
Also available as Technical Report CSR-122-82 from Department of Computer Science, University of Edinburgh, U.K.

[Milne 83b] G. J. Milne. *CIRCAL and the Representation of Communication, Concurrency and Time.* Technical Report CSR-151-83, Department of Computer Science, University of Edinburgh, U.K., November 1983.

[Milner 80] R. Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer Verlag, 1980.

[Milner 82] R. Milner. *Calculi for Synchrony and Asynchrony.* Technical Report CSR-104-82, Department of Computer Science, University of Edinburgh, U.K., August 1982.

[Moore 64] E. F. Moore. *Sequential Machines: Selected Papers.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1964.

[Morison 84] J. Morison et al. ELLA: hardware description or specification? In *Proceedings IEEE International Conference, CAD-84*, Santa Clara, November 1984.

[Moszkowski 83a] B. C. Moszkowski. A temporal logic for multi-level reasoning about hardware. In *Proceedings of the 6th International Symposium on Computer Hardware Description Languages*, pages 79–90, North-Holland Publishing Co., Pittsburgh, Pennsylvania, May 1983.
Also available as Technical Report STAN-CS-82-952 from Department of Computer Science, Stanford University.

[Moszkowski 83b] B. C. Moszkowski. *Reasoning about Digital Circuits.* PhD thesis, Department of Computer Science, Stanford University, July 1983. Also available as Technical Report STAN-CS-83-970.

[Moszkowski 86] B. C. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[Orton 84] D. W. R. Orton. *Clocked Dynamic Logic for CMOS.* Internal Memo, Racal Research Ltd, Worton Drive, Worton Grange Industrial Est., Reading RG2 OSB, England, January 1984.

[Osborne 80a] Holly C. Osborne. Stability analysis of an Nth power digital phase-locked loop—part I: first-order DPLL. *IEEE Transactions on Communications*, COM-28(8):1343–1354, August 1980.

[Osborne 80b] Holly C. Osborne. Stability analysis of an Nth power digital phase-locked loop—part II: second- and third-order DPLL's. *IEEE Transactions on Communications*, COM-28(8):1355–1364, August 1980.

[Paulson 83a] L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(2):119–149, August 1983.

[Paulson 83b] L. C. Paulson. *Tactics and Tacticals in Cambridge LCF.* Technical Report 39, Computer Laboratory, University of Cambridge, U.K., 1983.

[Rey 82] Thomas J. Rey. Comments on "A survey of digital phase-locked loops". *Proceedings of the IEEE*, 70(2):201–202, February 1982.

[Sandoz 84] Jean-Paul SanDoz and Willem Steenaart. Performance improvements of a binary quanitzed all-digital phase-locked loop with a new aided-acquisition technique. *IEEE Transactions on Communications*, COM-32(12):1269–1984, December 1984.

[Seitz 84] Charles L. Seitz et al. *Hot-Clocks nMOS.* Technical Report 5177:TR:85, Computer Science Department, California Institute of Technology, 1984.

[Sheeran 83] Mary Sheeran. *μFP, An Algebraic VLSI Design Language.* PhD thesis, University of Oxford, U.K., November 1983.

[Suzuki 73] Y. Suzuki, K. Oagawa, and T. Abe. Clocked CMOS calculator circuitry. *IEEE Journal of Solid-State Circuits*, SC-8:462–469, December 1973.

[Traub 83] N. Traub. *A Lisp Based CIRCAL Environment.* Technical Report CSR-152-83, Department of Computer Science, University of Edinburgh, U.K., November 1983.

[Traub 87] Niklas Traub. *A Formal Approach to Hardware Analysis.* PhD thesis, Department of Computer Science, University of Edinburgh, U.K., March 1987. Also available as Technical Report CST-43-87.

[Trinks 68] W. Trinks. *Governers and the Governing of Prime Movers.* D. Van Nostrand Co., Princeton, New Jersey, 1868.

[Weste 85] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[Winskel 87] Glynn Winskel. Relating two models of hardware. In *Category Theory and Computer Science.* Volume 283 of *Lecture Notes in Computer Science*, D. H. Pill, A. Poigné, and D. E. Rydeheard, editors, pages 98–113, Springer-Verlag, Berlin, 1987.

[Winskel 88] Glynn Winskel. A compositional model of MOS circuits. In *VLSI Specification, Verification and Synthesis*, Graham Birtwistle and P. A. Subrahmanyam, editors, pages 323–347, Kluwer Academic Publishers, 1988. Proceedings of the conference held on 12-16 January 1987 at Calgary, Canada.

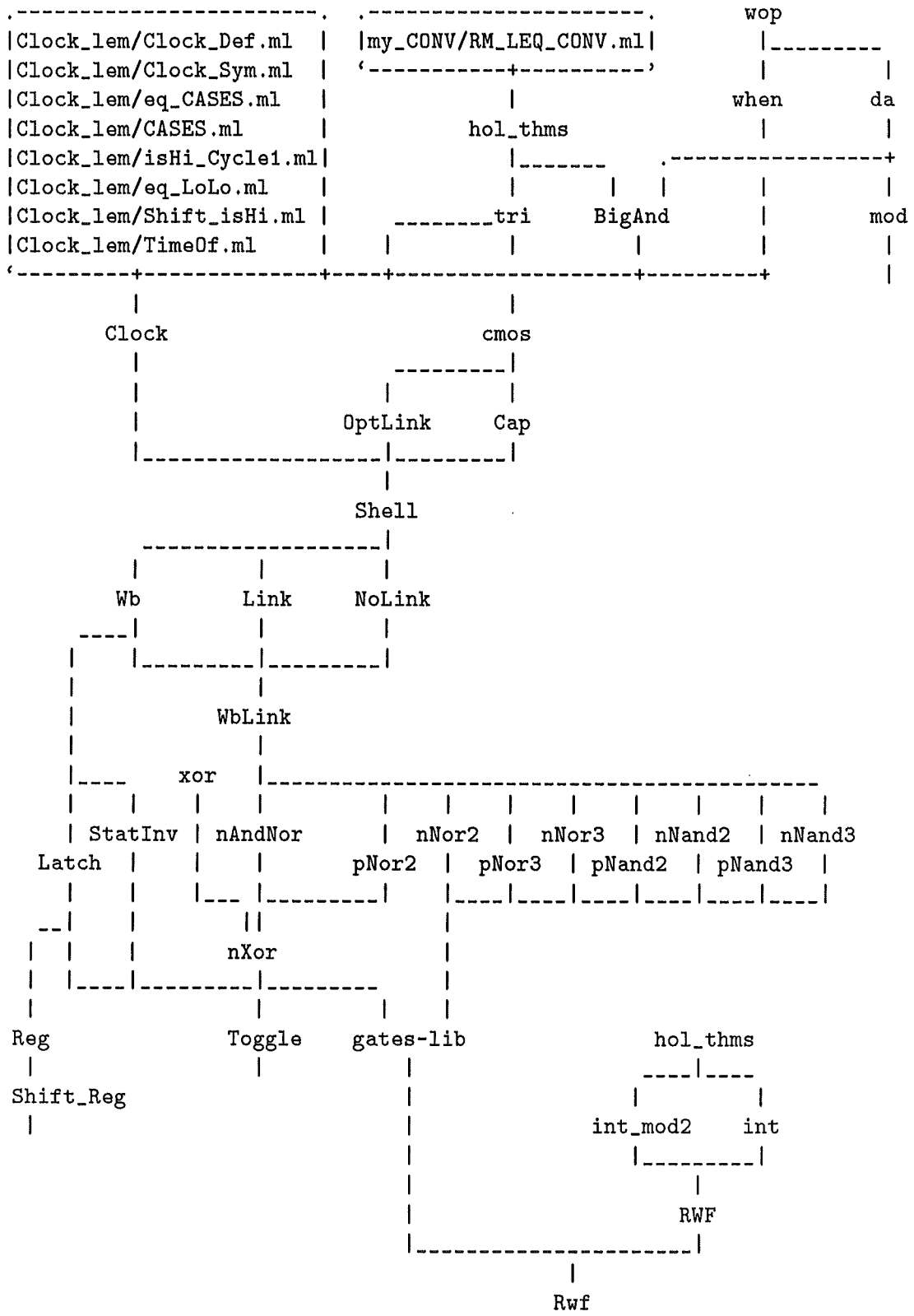
[Yamamoto78] Hisao Yamamoto and Shinsaku Mori. Performance of a binary quantised all digital phase-locked loop with a new class of sequential filter. *IEEE Transactions on Communications*, COM-26(1):35–45, January 1978.

[Yukawa73] J. Yukawa and S. Mori. A binary quantised all digital phase-locked loop. *IECE*, 56-A(12), December 1973.

Appendix A

The Hierarchy of Theories

This appendix contains the comments in the makefile which builds the entire set of theories for the CLIC world. These comments show the dependencies of the various theories represented as a tree. The leaves of this tree represent the list of theories corresponding to the various examples used in this thesis. The top most parent to all this work is the hol theory representing the basic HOL system which is not shown in here


Clock_lem/Clock_Def.ml
Clock_lem/Clock_Sym.ml
Clock_lem/eq_CASES.ml
Clock_lem/CASES.ml
Clock_lem/isHi_Cycle1.ml
Clock_lem/eq_LoLo.ml
Clock_lem/Shift_isHi.ml
Clock_lem/TimeOf.ml
my_CONV/RM_LEQ_CONV.ml
hol_thms
tri
BigAnd
wop
when
da
mod
Clock
cmos
OptLink
Cap
Shell
Wb
Link
NoLink
WbLink
xor
StatInv
nAndNor
nNor2
nNor3
nNand2
nNand3
Latch
pNor2
pNor3
pNand2
pNand3
nXor
Reg
Toggle
gates-lib
hol_thms
Shift_Reg
int_mod2
int
RWF
Rwf

Appendix B

ML Code for the Correctness of the Toggle Device

This appendix contains the ML code for the proof of correctness of a Toggle flip-flop device designed in the CLIC design style. Considerable background knowledge of the HOL system is needed in order to follow this code.

Before giving the actual code, here is a table which shows how the logical connectives as used in the body of this thesis are represented in the machine readable form as used in this appendix.

Logical Terms	Machine Readable Form
$\sim x$	`~x`
$t_1 \wedge t_2$	`t1 /\ t2`
$t_1 \vee t_2$	`t1 \/ t2`
$t_1 \supset t_2$	`t1 ==> t2`
$t_1 \equiv t_2$	`t1 <=> t2`
$t_1 = t_2$	`t1 = t2`
$\forall x.\ \mathrm{tm}[x]$	`!x. tm[x]`
$\exists x.\ \mathrm{tm}[x]$	`?x. tm[x]`
$\epsilon x.\ \mathrm{tm}[x]$	`@x. tm[x]`
$(b \Rightarrow t_1 \mid t_2)$	`(b => t1 | t2)`
let $x = a$ **in** $f(x)$	`let x = a in f(x)`

```
%-----------------------------------------------------------------------
| An approximation to a TOGGLE flip-flop designed in the CLIC.
|
| FILE          : Toggle.ml
| DESCRIPTION   : All the useful theorems to do with the Toggle device
|                 are proved in this file.  These theorems are collated
|                 and then brought together at the end into a single
|                 theorem as usual called:  Toggle_THM
|                 The Toggle device takes a single input and if the
|                 input is high then the output is toggled otherwise
|                 the output is held static at its old value.
|                 This can be formally stated as follows:
|
|                    op(t+1)  =  (ip t) => ~(op t) | (op t)
|
|                    op(t+1)  =  (ip t) xor (op t)
|
|
|                .---------------------------------.
|                |    _                            |
|                |   | \    |\                      |
|                `---|L 0---| 0--.                  |
|                    |_/    |/   |    ___      _    |
|                     _           `--))   \    | \  |
|                    | \             )) N  >---|L 0--+--- op
|             ip ----|L 0-----------))___/    |_/
|                    |_/
|
|                                 -------
|                                |  ___  |
|                                |   |   |
|                      ip -->--  |   |   |-->-- op
|                                |       |
|                                |_______|
|
|
| READS FILES   : nXor.th Latch.th StatInv.th
| WRITES FILES  : Toggle.th
|
| DATE          : 15.APR.86
| AUTHOR        : I. S. Dhingra
-----------------------------------------------------------------------%
new_theory `Toggle`;;

maptok new_parent `nXor Latch StatInv`;;

let sig  = ":num -> tri";;
```

```
let Toggle =
 new_definition
  ('Toggle',
   "!phi1 phi1' phi2 phi2' ip op :^sig.
         Toggle(phi1, phi1', phi2, phi2', ip, op) =
    ?p1 p2 p3 p4.       Latch(phi1, phi1', ip, p1)     /\
                        Latch(phi1, phi1', op, p2)     /\
                        StatInv(p2, p3)                /\
                         nXor(phi1, phi1', p1, p3, p4) /\
                        Latch(phi2, phi2', p4, op)      "
  );;


let Toggle_Spec =
 new_definition
  ('Toggle_Spec',
   "!ip op.  Toggle_Spec(ip,op) =
    !t.  op(t+1) = (ip t) => ~(op t) | op t"
  );;


close_theory();;


load_theorem 'Clock'   'Clock_Sym';
load_theorem 'Clock'   'Clock_Shift_isHi';
load_theorem 'Clock'   'Clock_isHi_TimeOf_isHi';
load_theorem 'Clock'   'Clock_TimeOf_isHi_plus4';

load_theorem 'nXor'    'nXor_Wb';
load_theorem 'nXor'    'nXor_Def';
load_theorem 'nXor'    'nXor_SPEC_1';
load_theorem 'nXor'    'nXor_SPEC_2';

load_theorem 'Latch'   'Latch_Wb';
load_theorem 'Latch'   'Latch_Def';
load_theorem 'Latch'   'Latch_Def_phi2';
load_theorem 'Latch'   'Latch_SPEC';
load_theorem 'Latch'   'Latch_SPEC_phi2';

load_theorem 'StatInv' 'StatInv_Wb';
load_theorem 'StatInv' 'StatInv_Def';
load_theorem 'StatInv' 'StatInv_SPEC';

load_theorem    'xor'  'xor_CLAUSES';

load_definition 'when' 'when';;
```

```
let BETA_TAC  = CONV_TAC  (DEPTH_CONV BETA_CONV);;


let EQ_RES_TAC = (IMP_RES_TAC o GEN_ALL o fst o EQ_IMP_RULE o SPEC_ALL);;


let Toggle_Wb =
 prove_thm
  (`Toggle_Wb`,
   "!phi1 phi1' phi2 phi2'.
            Clock(phi1,phi1',phi2,phi2') ==>
    !ip op. Toggle(phi1,phi1',phi2,phi2',ip,op) ==> (Wb op phi2 /\
                                                     Wb op phi2' )",
   PURE_REWRITE_TAC [Toggle]
   THEN REPEAT STRIP_TAC
   THEN EQ_RES_TAC Clock_Sym
   THEN IMP_RES_TAC Latch_Wb
   THEN RES_TAC
  );;


let Toggle_Def =
 prove_thm
  (`Toggle_Def`,
   "!phi1 phi1' phi2 phi2'.
            Clock(phi1,phi1',phi2,phi2') ==>
    !ip op. Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
    !t.                    (isHi phi1 t) ==> ( Def op (t+2) /\
                                               Def op (t+3) /\
                                               Def op (t+4) /\
                                               Def op (t+5)  )",
   PURE_REWRITE_TAC [Toggle]
   THEN REPEAT (FILTER_STRIP_TAC "isHi")
   THEN REPEAT DISCH_TAC
   THEN IMP_RES_TAC nXor_Def
   THEN IMP_RES_TAC Latch_Def_phi2
   THEN RES_TAC
   THEN ASM_REWRITE_TAC []
  );;
```

```
let Toggle_SPEC =
 prove_thm
  (`Toggle_SPEC`,
   "!phi1 phi1' phi2 phi2'.
           Clock(phi1,phi1',phi2,phi2') ==>
    !ip op.
     Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
    !t.                    (isHi phi1 t) ==>
                              Def ip t  ==>
                              Def op t  ==>
         (ValAbs op (t+2) = (ValAbs ip t)  xor  (ValAbs op t)) /\
         (ValAbs op (t+3) = (ValAbs ip t)  xor  (ValAbs op t)) /\
         (ValAbs op (t+4) = (ValAbs ip t)  xor  (ValAbs op t)) /\
         (ValAbs op (t+5) = (ValAbs ip t)  xor  (ValAbs op t))  ",
   PURE_REWRITE_TAC [Toggle]
   THEN REPEAT (FILTER_STRIP_TAC "Def")
   THEN REPEAT DISCH_TAC
   THEN EQ_RES_TAC Clock_Sym
   THEN IMP_RES_TAC Latch_Def
   THEN IMP_RES_TAC StatInv_Def
   THEN IMP_RES_TAC nXor_Def
   THEN IMP_RES_TAC Latch_Def_phi2
   THEN IMP_RES_TAC Latch_SPEC
   THEN IMP_RES_TAC StatInv_SPEC
   THEN IMP_RES_TAC nXor_SPEC_2
   THEN IMP_RES_TAC Latch_SPEC_phi2
   THEN IMP_RES_TAC Latch_Wb
   THEN IMP_RES_TAC StatInv_Wb
   THEN IMP_RES_TAC nXor_Wb
   THEN RES_TAC
   THEN ASM_REWRITE_TAC [xor_CLAUSES]
  );;
```

```
let Toggle_THM =
 prove_thm
  (`Toggle_THM`,
   "!phi1 phi1' phi2 phi2'.
           Clock(phi1,phi1',phi2,phi2') ==>
    !ip op.
     Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
          ( Wb op phi2                                              /\
            Wb op phi2'                                             /\
            !t.  (isHi phi1 t) ==>  ( Def op (t+2) /\
                                      Def op (t+3) /\
                                      Def op (t+4) /\
                                      Def op (t+5)  )               /\
                      Def ip t ==>
                      Def op t ==>
            (ValAbs op(t+2) = (ValAbs ip t) xor (ValAbs op t)) /\
            (ValAbs op(t+3) = (ValAbs ip t) xor (ValAbs op t)) /\
            (ValAbs op(t+4) = (ValAbs ip t) xor (ValAbs op t)) /\
            (ValAbs op(t+5) = (ValAbs ip t) xor (ValAbs op t))
          )",
   REPEAT STRIP_TAC
   THEN IMP_RES_TAC Toggle_Wb
   THEN IMP_RES_TAC Toggle_Def
   THEN IMP_RES_TAC Toggle_SPEC
   THEN RES_TAC
  );;
```

```
let Toggle_when_phi1 =
 prove_thm
  (`Toggle_when_phi1`,
   "!phi1 phi1' phi2 phi2'.
           Clock(phi1,phi1',phi2,phi2') ==>
    !ip op.
     Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
          ((Def op) when (isHi phi1)) 0  ==>
     (!t. ((Def ip) when (isHi phi1)) t) ==>
    !t. (((ValAbs op) when (isHi phi1)) (t+1) =
         ((ValAbs ip) when (isHi phi1)) t  xor
         ((ValAbs op) when (isHi phi1)) t      )",
   REWRITE_TAC [when]
   THEN BETA_TAC
   THEN REPEAT (FILTER_STRIP_TAC "Def ip")
   THEN DISCH_THEN (\th. INDUCT_TAC  THEN  MP_TAC th )
   THENL [ DISCH_THEN (ASSUME_TAC o (SPEC "0"))
           THEN IMP_RES_THEN (ASSUME_TAC o (SPEC "0"))
                             Clock_isHi_TimeOf_isHi
           THEN IMP_RES_THEN (\th.  REWRITE_TAC [th])
                             Clock_TimeOf_isHi_plus4
           THEN IMP_RES_TAC Toggle_SPEC
         ; DISCH_THEN (MP_TAC o (SPEC "t+1"))
           THEN IMP_RES_THEN (MP_TAC o (SPEC "t+1"))
                             Clock_isHi_TimeOf_isHi
           THEN IMP_RES_THEN (\th.  REWRITE_TAC [ADD1; th])
                             Clock_TimeOf_isHi_plus4
           THEN IMP_RES_THEN (ASSUME_TAC o (SPEC "t"))
                             Clock_isHi_TimeOf_isHi
           THEN (IMP_RES_THEN o IMP_RES_THEN)
                   IMP_RES_TAC
                   Toggle_Def
           THEN REPEAT STRIP_TAC
           THEN IMP_RES_TAC Toggle_SPEC
         ]
 );;
```

```
let xor_lemma =
 TAC_PROOF
  (([],
   "!a b.  a xor b = (a => ~b | b)"),
   X_GEN_TAC "a"
   THEN BOOL_CASES_TAC "a"
   THEN REWRITE_TAC [xor_CLAUSES]
  );;


let Toggle_Correct =
 save_thm
  ('Toggle_Correct',
   REWRITE_RULE [xor_lemma] Toggle_when_phi1
  );;
```

Here is the printout of the theory that is generated by all this code.

```
#print_theory 'Toggle';;

The Theory Toggle
Parents --  HOL     nXor     Latch     StatInv
Constants --
  Toggle
    ":(num -> tri) #
      ((num -> tri) #
       ((num -> tri) # ((num -> tri) # ((num -> tri) # (num -> tri))))) ->
      bool"
  Toggle_Spec ":(num -> bool) # (num -> bool) -> bool"
Definitions --
  Toggle
    |- !phi1 phi1' phi2 phi2' ip op.
        Toggle(phi1,phi1',phi2,phi2',ip,op) =
        (?p1 p2 p3 p4.
          Latch(phi1,phi1',ip,p1) /\
          Latch(phi1,phi1',op,p2) /\
          StatInv(p2,p3) /\
          nXor(phi1,phi1',p1,p3,p4) /\
          Latch(phi2,phi2',p4,op))
  Toggle_Spec
    |- !ip op.
        Toggle_Spec(ip,op) = (!t. op(t + 1) = (ip t => ~op t | op t))
```

```
Theorems --
  Toggle_Wb
    |- !phi1 phi1' phi2 phi2'.
        Clock(phi1,phi1',phi2,phi2') ==>
        (!ip op.
          Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
          Wb op phi2 /\ Wb op phi2')
  Toggle_Def
    |- !phi1 phi1' phi2 phi2'.
        Clock(phi1,phi1',phi2,phi2') ==>
        (!ip op.
          Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
          (!t.
            isHi phi1 t ==>
            Def op(t + 2) /\
            Def op(t + 3) /\
            Def op(t + 4) /\
            Def op(t + 5)))
  Toggle_SPEC
    |- !phi1 phi1' phi2 phi2'.
        Clock(phi1,phi1',phi2,phi2') ==>
        (!ip op.
          Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
          (!t.
            isHi phi1 t ==>
            Def ip t ==>
            Def op t ==>
            (ValAbs op(t + 2) = (ValAbs ip t) xor (ValAbs op t)) /\
            (ValAbs op(t + 3) = (ValAbs ip t) xor (ValAbs op t)) /\
            (ValAbs op(t + 4) = (ValAbs ip t) xor (ValAbs op t)) /\
            (ValAbs op(t + 5) = (ValAbs ip t) xor (ValAbs op t))))
```

```
Toggle_THM
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!ip op.
        Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
        Wb op phi2 /\
        Wb op phi2' /\
        (!t.
          isHi phi1 t ==>
          (Def op(t + 2) /\
           Def op(t + 3) /\
           Def op(t + 4) /\
           Def op(t + 5)) /\
          Def ip t ==>
          Def op t ==>
          (ValAbs op(t + 2) = (ValAbs ip t) xor (ValAbs op t)) /\
          (ValAbs op(t + 3) = (ValAbs ip t) xor (ValAbs op t)) /\
          (ValAbs op(t + 4) = (ValAbs ip t) xor (ValAbs op t)) /\
          (ValAbs op(t + 5) = (ValAbs ip t) xor (ValAbs op t))))
Toggle_when_phi1
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!ip op.
        Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
        ((Def op) when (isHi phi1))0 ==>
        (!t. ((Def ip) when (isHi phi1))t) ==>
        (!t.
          ((ValAbs op) when (isHi phi1))(t + 1) =
          (((ValAbs ip) when (isHi phi1))t) xor
          (((ValAbs op) when (isHi phi1))t)))
Toggle_Correct
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!ip op.
        Toggle(phi1,phi1',phi2,phi2',ip,op) ==>
        ((Def op) when (isHi phi1))0 ==>
        (!t. ((Def ip) when (isHi phi1))t) ==>
        (!t.
          ((ValAbs op) when (isHi phi1))(t + 1) =
          (((ValAbs ip) when (isHi phi1))t =>
           ~((ValAbs op) when (isHi phi1))t |
           ((ValAbs op) when (isHi phi1))t)))

************************************
() : void

#
```

Appendix C

ML Code for the Correctness of the Random Walk Filter

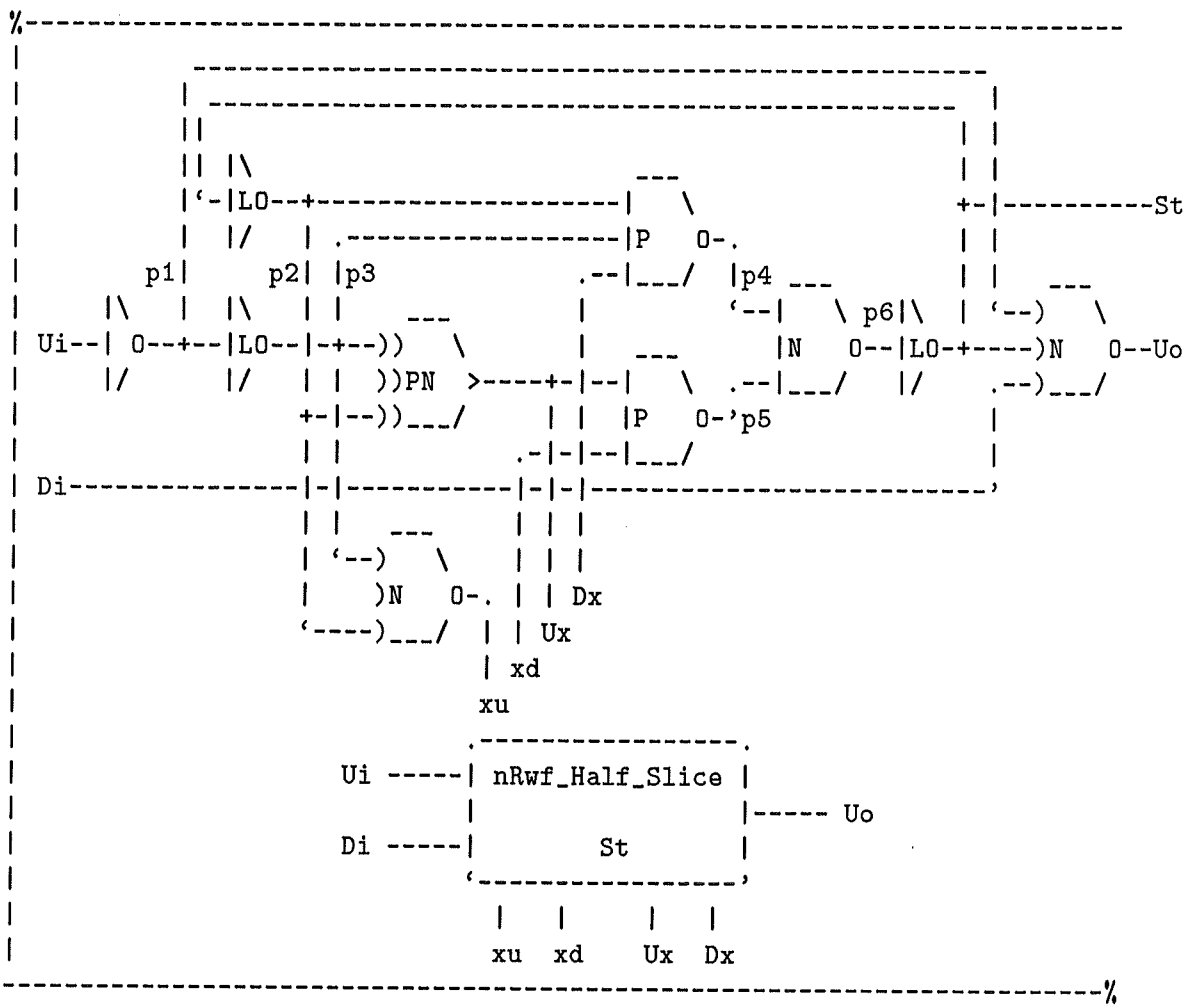
This appendix contains the ML code for the proof of correctness of a random walk filter device designed in the CLIC design style. Considerable background knowledge of the HOL system is needed in order to follow this code.

Before giving the actual code, here is a table which shows how the logical connectives as used in the body of this thesis are represented in the machine readable form as used in this appendix.

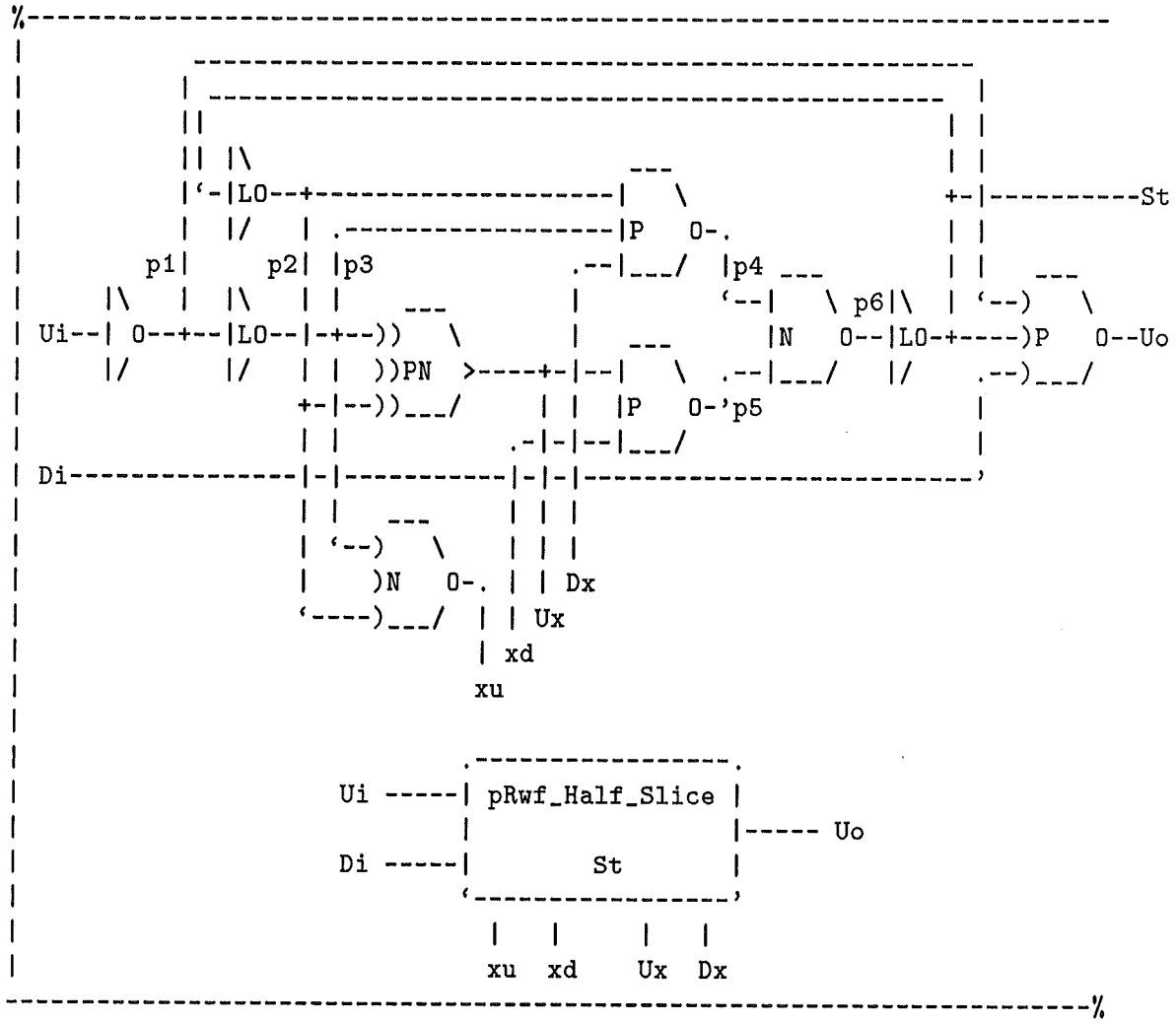
Logical Terms	Machine Readable Form
$\sim x$	`~x`
$t_1 \wedge t_2$	`t1 /\ t2`
$t_1 \vee t_2$	`t1 \/ t2`
$t_1 \supset t_2$	`t1 ==> t2`
$t_1 \equiv t_2$	`t1 <=> t2`
$t_1 = t_2$	`t1 = t2`
$\forall x.\ \mathrm{tm}[x]$	`!x. tm[x]`
$\exists x.\ \mathrm{tm}[x]$	`?x. tm[x]`
$\epsilon x.\ \mathrm{tm}[x]$	`@x. tm[x]`
$(b \Rightarrow t_1 \mid t_2)$	`(b => t1 | t2)`

let $x = a$
in
$f(x)$
	`let x = a`
	`in`
	`f(x)`

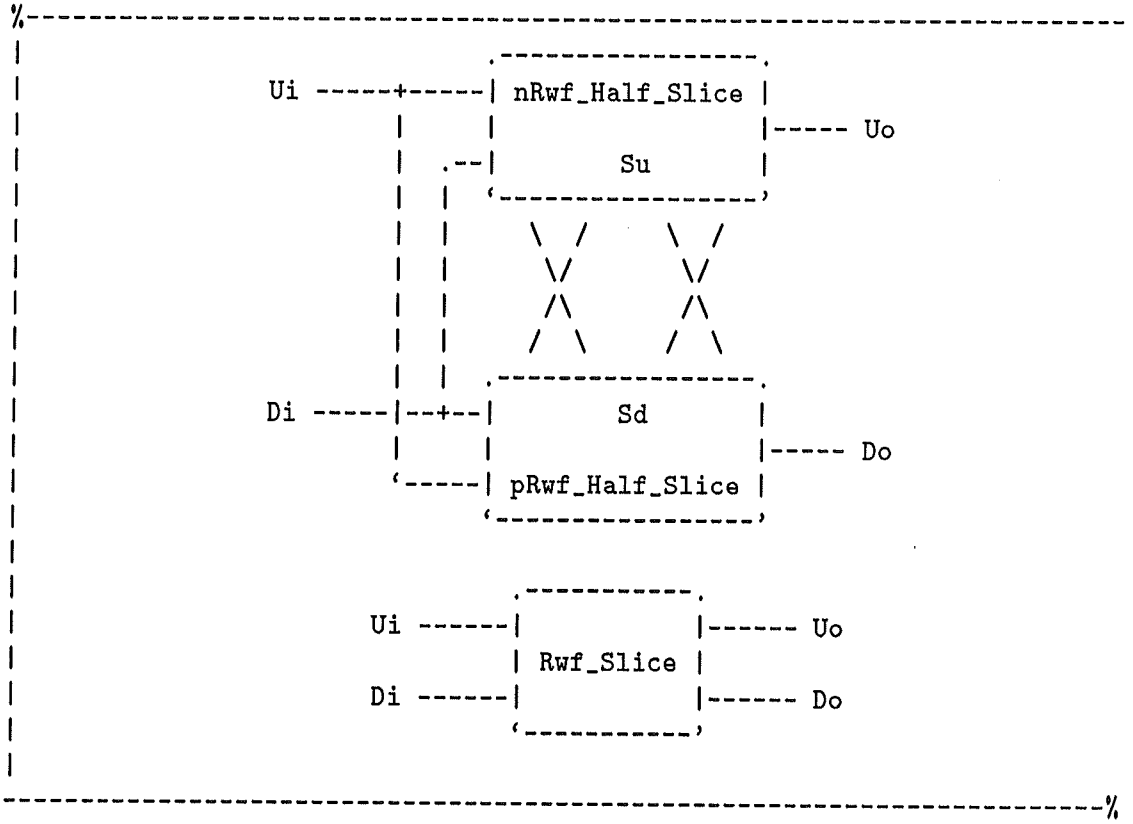
```
new_theory `Rwf`;;

maptok new_parent `gates-lib RWF`;;
```




```
let nRwf_Half_Slice =
 new_definition
  (`nRwf_Half_Slice`,
   "!               phi1 phi1' phi2 phi2' Ui Di xu xd Ux Dx Uo St.
    nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) =
    ?p1 p2 p3 p4 p5 p6. StatInv(                          Ui,         p1) /\
                         Latch(phi1,phi1',                St,         p2) /\
                         Latch(phi1,phi1',                p1,         p3) /\
                          nXor(phi1,phi1',                p2, p3,     Ux) /\
                          nNor2(     phi1',               p2, p3,     xu) /\
                         pNand3(phi1,                     p2, p3, Dx, p4) /\
                         pNand2(phi1,                     Ux, xd,     p5) /\
                         nNand2(     phi1',               p4, p5,     p6) /\
                          Latch(           phi2,phi2',    p6,         St) /\
                          nNor3(                phi2',    p1, St, Di, Uo)
  ");;
```




```
let pRwf_Half_Slice =
 new_definition
  ('pRwf_Half_Slice',
   "!              phi1 phi1' phi2 phi2' Ui Di xu xd Ux Dx Uo St.
    pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) =
    ?p1 p2 p3 p4 p5 p6. StatInv(                          Ui,         p1) /\
                         Latch(phi1,phi1',                St,         p2) /\
                         Latch(phi1,phi1',                p1,         p3) /\
                          nXor(phi1,phi1',                p2, p3,     Ux) /\
                         nNor2(     phi1',                p2, p3,     xu) /\
                        pNand3(phi1,                      p2, p3, Dx, p4) /\
                        pNand2(phi1,                      Ux, xd,     p5) /\
                        nNand2(     phi1',                p4, p5,     p6) /\
                         Latch(           phi2,phi2',     p6,         St) /\
                         pNor3(           phi2,           p1, St, Di, Uo)
   "
  );;
```




```
let Rwf_Slice =
 new_definition
  ('Rwf_Slice',
   "!          phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd.
     Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) =
     ?xu xd Ux Dx.
      nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,Su) /\
      pRwf_Half_Slice(phi1,phi1',phi2,phi2',Di,Ui,xd,xu,Dx,Ux,Do,Sd)
   "
  );;
```

```
%-----------------------------------------------------------------------
|                            Rwf (SUC n)
|      /-------------------------------------------------------\
|                      Rwf n                      Rwf_Slice
|          /-------------------------\          /-------\
|                ---                    ---                ---
|      Ui -->|   |-->  ...  -->|   |--> Uo(n) -->|   |--> Uo(n+1)
|            |   |             |   |             |   |
|      Di -->|___|-->  ...  -->|___|--> Do(n) -->|___|--> Do(n+1)
|           Su(0)      ...     Su(n)             Su(n+1)
|           Sd(0)      ...     Sd(n)             Sd(n+1)
-----------------------------------------------------------------------%
let Rwf =
 new_prim_rec_definition
  ('Rwf',
   "(!           phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd.
      Rwf      0 phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd   =
      Rwf_Slice (phi1,phi1',phi2,phi2',Ui,Di,(Uo 0),(Do 0),
                                          (Su 0),(Sd 0))      ) /\
    (!         n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd.
      Rwf (SUC n)phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd   =
      Rwf      n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd   /\
      Rwf_Slice (phi1,phi1',phi2,phi2',(Uo n),     (Do n),
                                       (Uo(SUC n)),(Do(SUC n)),
                                       (Su(SUC n)),(Sd(SUC n)) ) )
   ");;


%-----------------------------------------------------------------------
|   Rwf_Half_Slice_Spec =
|   |- Rwf_Half_Slice_Spec t Ui Di xu xd Ux Dx Uo St  =
|      !t. (Ux t    =  St t xor Ui t                   ) /\
|          (xu t    = ~St t /\ ~Ui t                   ) /\
|          (Uo t    =  Ui t /\  St t /\ ~Di t           ) /\
|          (St(t+1) =  St t /\  Ui t /\  Dx t  \/
|                     (St t xor Ui t)/\  xd t)
-----------------------------------------------------------------------%
let Rwf_Half_Slice_Spec =
 new_definition
  ('Rwf_Half_Slice_Spec',
   "!Ui Di xu xd Ux Dx Uo St.
    Rwf_Half_Slice_Spec Ui Di xu xd Ux Dx Uo St  =
    !t. (Ux t    =  St t xor Ui t                   ) /\
        (xu t    = ~St t /\ ~Ui t                   ) /\
        (Uo t    =  Ui t /\  St t /\ ~Di t           ) /\
        (St(t+1) =  St t /\  Ui t /\  Dx t  \/
                   (St t xor Ui t)/\  xd t)
  ");;
```

```
%-----------------------------------------------------------------------
|   Rwf_Slice_Spec =
|   |- Rwf_Slice_Spec Ui Di Uo Do Su Sd =
|      !t.  (Uo (t+1) = Ui (t+1) /\ Su (t+1) /\ ~Di (t+1)     ) /\
|           (Do (t+1) = Di (t+1) /\ Sd (t+1) /\ ~Ui (t+1)     ) /\
|           (Su (t+1) = Su t /\  Ui t  /\ (Sd t xor Di t) \/
|                       (Su t xor Ui t) /\ ~Sd t /\ ~Di t     ) /\
|           (Sd (t+1) = Sd t /\  Di t  /\ (Su t xor Ui t) \/
|                       (Sd t xor Di t) /\ ~Su t /\ ~Ui t     )
-----------------------------------------------------------------------%
let Rwf_Slice_Spec =
 new_definition
  ('Rwf_Slice_Spec',
   "!Ui Di Uo Do Su Sd.
    Rwf_Slice_Spec Ui Di Uo Do Su Sd =
    !t.  (Uo (t+1) = Ui (t+1) /\ Su (t+1) /\ ~Di (t+1)     ) /\
         (Do (t+1) = Di (t+1) /\ Sd (t+1) /\ ~Ui (t+1)     ) /\
         (Su (t+1) = Su t /\  Ui t  /\ (Sd t xor Di t) \/
                     (Su t xor Ui t) /\ ~Sd t /\ ~Di t     ) /\
         (Sd (t+1) = Sd t /\  Di t  /\ (Su t xor Ui t) \/
                     (Sd t xor Di t) /\ ~Su t /\ ~Ui t     )
   "
  );;


%-----------------------------------------------------------------------
|   Rwf_Spec n                        ---
|                                    |   |
|                        ip ---------| s |--------- op
|                  ":int_mod2"       |___|     ":int_mod2"
|
| Note: The parameter "n" is used to determine to what value the internal
|       state "s" is suppose to count up to.
-----------------------------------------------------------------------%
let Rwf_Spec =
 new_definition
  ('Rwf_Spec',
   "!n ip op s.  Rwf_Spec n ip op s =
         (s  0     = INT 0)  /\
    !t.  (NUM(int_mod2_INT ip t  plus  s t)  <  2 EXP (n+1))
         => ( (op t = zero)  /\  (s (SUC t) = (int_mod2_INT ip t  plus  s t)))
          | ( (op t = ip t)  /\  (s (SUC t) = INT 0) )
   "
  );;


close_theory();;
```

```
let BETA_TAC  = CONV_TAC  (DEPTH_CONV BETA_CONV);;

let EQ_RES_TAC = (IMP_RES_TAC o GEN_ALL o fst o EQ_IMP_RULE o SPEC_ALL)
and EQ_RES_THEN ttac eqth =
   IMP_RES_THEN ttac ((GEN_ALL o fst o EQ_IMP_RULE o SPEC_ALL) eqth);;


loadt `my_CONV/REPEAT_IMP_RES_TAC`;;


load_theorem `hol_thms` `fun_comp`;

load_theorem `Clock`    `Clock_Sym`;
load_theorem `Clock`    `Clock_Shift_isHi`;
load_theorem `Clock`    `Clock_isHi_TimeOf_isHi`;
load_theorem `Clock`    `Clock_TimeOf_isHi_plus4`;

load_theorem `StatInv`  `StatInv_Wb`;
load_theorem `StatInv`  `StatInv_Def`;
load_theorem `StatInv`  `StatInv_SPEC`;

load_theorem `Latch`    `Latch_Wb`;
load_theorem `Latch`    `Latch_Def`;
load_theorem `Latch`    `Latch_Def_phi2`;
load_theorem `Latch`    `Latch_SPEC`;
load_theorem `Latch`    `Latch_SPEC_phi2`;

load_theorem `nXor`     `nXor_Wb`;
load_theorem `nXor`     `nXor_Def`;
load_theorem `nXor`     `nXor_SPEC_1`;
load_theorem `nXor`     `nXor_SPEC_2`;

load_definition `xor`   `xor`;
load_theorem    `xor`   `xor_CLAUSES`;

load_theorem `nNor2`    `nNor2_Wb`;
load_theorem `nNor2`    `nNor2_Def`;
load_theorem `nNor2`    `nNor2_SPEC_1`;
load_theorem `nNor2`    `nNor2_SPEC_2`;

load_theorem `nNor3`    `nNor3_Wb`;
load_theorem `nNor3`    `nNor3_Def`;
load_theorem `nNor3`    `nNor3_Def_phi2`;
load_theorem `nNor3`    `nNor3_SPEC_1`;
load_theorem `nNor3`    `nNor3_SPEC_2`;
load_theorem `nNor3`    `nNor3_SPEC_phi2`;
```

```
load_theorem `pNor3`    `pNor3_Wb`;
load_theorem `pNor3`    `pNor3_Def`;
load_theorem `pNor3`    `pNor3_Def_phi2`;
load_theorem `pNor3`    `pNor3_SPEC_1`;
load_theorem `pNor3`    `pNor3_SPEC_2`;
load_theorem `pNor3`    `pNor3_SPEC_phi2`;

load_theorem `pNand3`   `pNand3_Wb`;
load_theorem `pNand3`   `pNand3_Def`;
load_theorem `pNand3`   `pNand3_SPEC_1`;
load_theorem `pNand3`   `pNand3_SPEC_2`;

load_theorem `pNand2`   `pNand2_Wb`;
load_theorem `pNand2`   `pNand2_Def`;
load_theorem `pNand2`   `pNand2_SPEC_1`;
load_theorem `pNand2`   `pNand2_SPEC_2`;

load_theorem `nNand2`   `nNand2_Wb`;
load_theorem `nNand2`   `nNand2_Def`;
load_theorem `nNand2`   `nNand2_SPEC_1`;
load_theorem `nNand2`   `nNand2_SPEC_2`;

load_definition `when`  `when`;;


%----------------------------------------------------------------------
|   Rwf_Half_Slice_Spec_EQ =
|   |- Rwf_Half_Slice_Spec = RWF_HALF_SLICE
----------------------------------------------------------------------%
let Rwf_Half_Slice_Spec_EQ =
 prove_thm
  (`Rwf_Half_Slice_Spec_EQ`,
   "Rwf_Half_Slice_Spec    = RWF_HALF_SLICE",
   REPEAT (CHANGED_TAC (CONV_TAC (ONCE_DEPTH_CONV FUN_EQ_CONV)))
   THEN REWRITE_TAC [ Rwf_Half_Slice_Spec
                    ; xor
                    ; CONV_RULE (ONCE_DEPTH_CONV AND_FORALL_CONV)
                                (theorem `RWF` `RWF_HALF_SLICE_THM`)
                    ]
  );;
```

```
%----------------------------------------------------------------------
|   Rwf_Slice_Spec_EQ =
|   |- !Ui Di Uo Do Su Sd.
|       (Rwf_Slice_Spec Ui Di Uo Do Su Sd /\
|           ~(Uo 0) /\ ~(Do 0) /\ ~(Su 0) /\
|                                  ~(Sd 0)   ) =
|            (RWF_SLICE Ui Di Uo Do Su Sd /\
|                                  ~(Su 0) /\
|                                  ~(Sd 0)   )
----------------------------------------------------------------------%
let Rwf_Slice_Spec_EQ =
 prove_thm
  (`Rwf_Slice_Spec_EQ`,
   "!Ui Di Uo Do Su Sd.
     (Rwf_Slice_Spec Ui Di Uo Do Su Sd /\
         ~(Uo 0) /\ ~(Do 0) /\ ~(Su 0) /\
                                ~(Sd 0)   ) =
          (RWF_SLICE Ui Di Uo Do Su Sd /\
                                ~(Su 0) /\
                                ~(Sd 0)   )",
   REWRITE_TAC [ theorem `RWF` `RWF_SLICE_THM`
               ; CONV_RULE (DEPTH_CONV (CHANGED_CONV FORALL_AND_CONV))
                           Rwf_Slice_Spec
               ; xor
               ]
   THEN REPEAT GEN_TAC
   THEN EQ_TAC
   THEN STRIP_TAC
   THEN ASM_REWRITE_TAC[]
   THEN CONJ_TAC
   THEN X_GEN_TAC "t"
   THEN STRIP_ASSUME_TAC (SPEC "t" num_CASES)
   THEN ASM_REWRITE_TAC [ADD1]
  );;


%----------------------------------------------------------------------
|   Rwf_Spec_EQ  =  |- Rwf_Spec = RWF_SPEC
----------------------------------------------------------------------%
let Rwf_Spec_EQ =
 prove_thm
  (`Rwf_Spec_EQ`,
   "Rwf_Spec  =  RWF_SPEC",
   REPEAT (CHANGED_TAC (CONV_TAC (ONCE_DEPTH_CONV FUN_EQ_CONV)))
   THEN REWRITE_TAC [Rwf_Spec; definition `RWF` `RWF_SPEC` ]
  );;
```

```
let nRwf_Half_Slice_Wb =
 prove_thm
  (`nRwf_Half_Slice_Wb`,
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                        (Wb St phi2  /\
                                                         Wb St phi2' /\
                                                         Wb Uo phi2  /\
                                                         Wb xu phi1  /\
                                                         Wb Ux phi1    )",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC nRwf_Half_Slice
   THEN EQ_RES_TAC Clock_Sym
   THEN REPEAT_IMP_RES_TAC Latch_Wb
   THEN REPEAT_IMP_RES_TAC nNor3_Wb
   THEN REPEAT_IMP_RES_TAC nNor2_Wb
   THEN REPEAT_IMP_RES_TAC nXor_Wb
   THEN ASM_REWRITE_TAC []
  );;


let pRwf_Half_Slice_Wb =
 prove_thm
  (`pRwf_Half_Slice_Wb`,
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                       (Wb St phi2  /\
                                                        Wb St phi2' /\
                                                        Wb Uo phi2' /\
                                                        Wb xu phi1  /\
                                                        Wb Ux phi1    )",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC pRwf_Half_Slice
   THEN EQ_RES_TAC Clock_Sym
   THEN REPEAT_IMP_RES_TAC Latch_Wb
   THEN REPEAT_IMP_RES_TAC pNor3_Wb
   THEN REPEAT_IMP_RES_TAC nNor2_Wb
   THEN REPEAT_IMP_RES_TAC nXor_Wb
   THEN ASM_REWRITE_TAC []
  );;
```

```
let nRwf_Half_Slice_Def =
 prove_thm
  (`nRwf_Half_Slice_Def`,
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
    !t.                                                  (isHi phi1 t) ==>
       (Def St (t+2) /\ Def St (t+3) /\ Def St (t+4) /\ Def St (t+5) /\
        Def Uo (t+2) /\ Def Uo (t+3) /\ Def Uo (t+4) /\ Def Uo (t+5) /\
        Def xu (t  ) /\ Def xu (t+1) /\ Def xu (t+2) /\ Def xu (t+3) /\
        Def Ux (t  ) /\ Def Ux (t+1) /\ Def Ux (t+2) /\ Def Ux (t+3)  )",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC nRwf_Half_Slice
   THEN REPEAT_IMP_RES_TAC nXor_Def
   THEN REPEAT_IMP_RES_TAC nNor2_Def
   THEN REPEAT_IMP_RES_TAC nNor3_Def_phi2
   THEN REPEAT_IMP_RES_TAC nNand2_Def
   THEN REPEAT_IMP_RES_TAC Latch_Def_phi2
   THEN ASM_REWRITE_TAC []
  );;


let pRwf_Half_Slice_Def =
 prove_thm
  (`pRwf_Half_Slice_Def`,
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
    !t.                                                  (isHi phi1 t) ==>
       (Def St (t+2) /\ Def St (t+3) /\ Def St (t+4) /\ Def St (t+5) /\
        Def Uo (t+2) /\ Def Uo (t+3) /\ Def Uo (t+4) /\ Def Uo (t+5) /\
        Def xu (t  ) /\ Def xu (t+1) /\ Def xu (t+2) /\ Def xu (t+3) /\
        Def Ux (t  ) /\ Def Ux (t+1) /\ Def Ux (t+2) /\ Def Ux (t+3)  )",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC pRwf_Half_Slice
   THEN REPEAT_IMP_RES_TAC nXor_Def
   THEN REPEAT_IMP_RES_TAC nNor2_Def
   THEN REPEAT_IMP_RES_TAC pNor3_Def_phi2
   THEN REPEAT_IMP_RES_TAC nNand2_Def
   THEN REPEAT_IMP_RES_TAC Latch_Def_phi2
   THEN ASM_REWRITE_TAC []
  );;
```

```
let Rwf_Slice_Wb =
 prove_thm
  (`Rwf_Slice_Wb`,
   "!phi1 phi1' phi2 phi2'.  Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di Uo Do Su Sd.
       Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) ==>
                                                  (Wb Su phi2  /\
                                                   Wb Sd phi2' /\
                                                   Wb Uo phi2  /\
                                                   Wb Do phi2'   )",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC Rwf_Slice
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_Wb
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_Wb
   THEN ASM_REWRITE_TAC []
  );;


let Rwf_Slice_Def =
 prove_thm
  (`Rwf_Slice_Def`,
   "!phi1 phi1' phi2 phi2'.  Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di Uo Do Su Sd.
       Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) ==>
    !t.                                        (isHi phi1 t) ==>
       (Def Su (t+2) /\ Def Su (t+3) /\ Def Su (t+4) /\ Def Su (t+5) /\
        Def Sd (t+2) /\ Def Sd (t+3) /\ Def Sd (t+4) /\ Def Sd (t+5) /\
        Def Uo (t+2) /\ Def Uo (t+3) /\ Def Uo (t+4) /\ Def Uo (t+5) /\
        Def Do (t+2) /\ Def Do (t+3) /\ Def Do (t+4) /\ Def Do (t+5)  )",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC Rwf_Slice
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_Def
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_Def
   THEN ASM_REWRITE_TAC []
  );;
```

```
let Rwf_Wb =
 prove_thm
  ('Rwf_Wb',
   "!phi1 phi1' phi2 phi2'.  Clock(phi1,phi1',phi2,phi2') ==>
    !n Ui Di Uo Do Su Sd.
           Rwf n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd ==>
                                              (Wb (Su n) phi2  /\
                                               Wb (Sd n) phi2' /\
                                               Wb (Uo n) phi2  /\
                                               Wb (Do n) phi2'   )",
   REPEAT GEN_TAC
   THEN DISCH_TAC
   THEN INDUCT_TAC
   THEN PURE_REWRITE_TAC [Rwf]
   THEN REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN REPEAT_IMP_RES_TAC Rwf_Slice_Wb
   THEN ASM_REWRITE_TAC []
  );;


let Rwf_Def =
 prove_thm
  ('Rwf_Def',
   "!phi1 phi1' phi2 phi2'.  Clock(phi1,phi1',phi2,phi2') ==>
    !n Ui Di Uo Do Su Sd.
           Rwf n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd ==>
    !t.                                       (isHi phi1 t) ==>
 (Def(Su n)(t+2) /\ Def(Su n)(t+3) /\ Def(Su n)(t+4) /\ Def(Su n)(t+5) /\
  Def(Sd n)(t+2) /\ Def(Sd n)(t+3) /\ Def(Sd n)(t+4) /\ Def(Sd n)(t+5) /\
  Def(Uo n)(t+2) /\ Def(Uo n)(t+3) /\ Def(Uo n)(t+4) /\ Def(Uo n)(t+5) /\
  Def(Do n)(t+2) /\ Def(Do n)(t+3) /\ Def(Do n)(t+4) /\ Def(Do n)(t+5)  )",
   REPEAT GEN_TAC
   THEN DISCH_TAC
   THEN INDUCT_TAC
   THEN PURE_REWRITE_TAC [Rwf]
   THEN REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN REPEAT_IMP_RES_TAC Rwf_Slice_Def
   THEN ASM_REWRITE_TAC []
  );;
```

```
let nRwf_Half_Slice_Ux = prove_thm ('nRwf_Half_Slice_Ux',
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                      Wb Ui phi2  ==>
                                                      Wb Di phi2' ==>
                                                      Wb xd phi1  ==>
                                                      Wb Dx phi1  ==>
    !t.                                              (isHi phi1 t) ==>
                                                         Def Ui t  ==>
                                                         Def St t  ==>
              (ValAbs Ux (t+1)  =  ~ValAbs St t  xor  ValAbs Ui t ) /\
              (ValAbs Ux (t+2)  =  ~ValAbs St t  xor  ValAbs Ui t )  ",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC nRwf_Half_Slice
   THEN REPEAT_IMP_RES_TAC StatInv_SPEC
   THEN REPEAT_IMP_RES_TAC StatInv_Def
   THEN REPEAT_IMP_RES_TAC Latch_SPEC
   THEN REPEAT_IMP_RES_TAC Latch_Def
   THEN REPEAT_IMP_RES_TAC nXor_SPEC_1
   THEN REPEAT_IMP_RES_TAC Latch_Wb
   THEN REPEAT_IMP_RES_TAC nXor_SPEC_2
   THEN ASM_REWRITE_TAC [DE_MORGAN_THM; xor_CLAUSES] );;


let pRwf_Half_Slice_Ux = prove_thm ('pRwf_Half_Slice_Ux',
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                      Wb Ui phi2' ==>
                                                      Wb Di phi2  ==>
                                                      Wb xd phi1  ==>
                                                      Wb Dx phi1  ==>
    !t.                                              (isHi phi1 t) ==>
                                                         Def Ui t  ==>
                                                         Def St t  ==>
              (ValAbs Ux (t+1)  =  ~ValAbs St t  xor  ValAbs Ui t ) /\
              (ValAbs Ux (t+2)  =  ~ValAbs St t  xor  ValAbs Ui t )  ",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC pRwf_Half_Slice
   THEN REPEAT_IMP_RES_TAC StatInv_SPEC
   THEN REPEAT_IMP_RES_TAC StatInv_Def
   THEN REPEAT_IMP_RES_TAC Latch_SPEC
   THEN REPEAT_IMP_RES_TAC Latch_Def
   THEN REPEAT_IMP_RES_TAC nXor_SPEC_1
   THEN REPEAT_IMP_RES_TAC Latch_Wb
   THEN REPEAT_IMP_RES_TAC nXor_SPEC_2
   THEN ASM_REWRITE_TAC [DE_MORGAN_THM; xor_CLAUSES] );;
```

```
let nRwf_Half_Slice_xu = prove_thm ('nRwf_Half_Slice_xu',
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                      Wb Ui phi2  ==>
                                                      Wb Di phi2' ==>
                                                      Wb xd phi1  ==>
                                                      Wb Dx phi1  ==>
    !t.                                              (isHi phi1 t) ==>
                                                         Def Ui t  ==>
                                                         Def St t  ==>
              (ValAbs xu (t+1)  =  ValAbs St t  /\  ~ValAbs Ui t ) /\
              (ValAbs xu (t+2)  =  ValAbs St t  /\  ~ValAbs Ui t )  ",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC nRwf_Half_Slice
   THEN REPEAT_IMP_RES_TAC StatInv_SPEC
   THEN REPEAT_IMP_RES_TAC StatInv_Def
   THEN REPEAT_IMP_RES_TAC Latch_SPEC
   THEN REPEAT_IMP_RES_TAC Latch_Def
   THEN REPEAT_IMP_RES_TAC nNor2_SPEC_1
   THEN REPEAT_IMP_RES_TAC Latch_Wb
   THEN REPEAT_IMP_RES_TAC nNor2_SPEC_2
   THEN ASM_REWRITE_TAC [DE_MORGAN_THM] );;


let pRwf_Half_Slice_xu = prove_thm ('pRwf_Half_Slice_xu',
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                      Wb Ui phi2' ==>
                                                      Wb Di phi2  ==>
                                                      Wb xd phi1  ==>
                                                      Wb Dx phi1  ==>
    !t.                                              (isHi phi1 t) ==>
                                                        Def Ui t  ==>
                                                        Def St t  ==>
              (ValAbs xu (t+1)  =  ValAbs St t  /\  ~ValAbs Ui t ) /\
              (ValAbs xu (t+2)  =  ValAbs St t  /\  ~ValAbs Ui t )  ",
  REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
  THEN EQ_RES_THEN STRIP_ASSUME_TAC pRwf_Half_Slice
  THEN REPEAT_IMP_RES_TAC StatInv_SPEC
  THEN REPEAT_IMP_RES_TAC StatInv_Def
  THEN REPEAT_IMP_RES_TAC Latch_SPEC
  THEN REPEAT_IMP_RES_TAC Latch_Def
  THEN REPEAT_IMP_RES_TAC nNor2_SPEC_1
  THEN REPEAT_IMP_RES_TAC Latch_Wb
  THEN REPEAT_IMP_RES_TAC nNor2_SPEC_2
  THEN ASM_REWRITE_TAC [DE_MORGAN_THM] );;
```

```
let nRwf_Half_Slice_Uo =
 prove_thm
  (`nRwf_Half_Slice_Uo`,
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                        Wb Ui phi2  ==>
                                                        Wb Di phi2' ==>
    !t.                                                (isHi phi1 t) ==>
                                                        Def Ui (t+3) ==>
                                                        Def Ui (t+4) ==>
                                                        Def Di (t+3) ==>
                                                        Def Di (t+4) ==>
            (ValAbs Uo(t+3)  =  ValAbs Ui(t+3) /\
                               ~ValAbs St(t+3) /\
                               ~ValAbs Di(t+3)    ) /\
            (ValAbs Uo(t+4)  =  ValAbs Ui(t+4) /\
                               ~ValAbs St(t+4) /\
                               ~ValAbs Di(t+4)    )
   ",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC nRwf_Half_Slice
   THEN EQ_RES_TAC Clock_Sym
   THEN REPEAT_IMP_RES_TAC StatInv_Def
   THEN REPEAT_IMP_RES_TAC StatInv_Wb
   THEN REPEAT_IMP_RES_TAC nNand2_Def
   THEN REPEAT_IMP_RES_TAC Latch_Def_phi2
   THEN REPEAT_IMP_RES_TAC Latch_Wb
   THEN REPEAT_IMP_RES_TAC nNor3_SPEC_phi2
   THEN REPEAT_IMP_RES_TAC StatInv_SPEC
   THEN ASM_REWRITE_TAC [DE_MORGAN_THM]
  );;
```

```
let pRwf_Half_Slice_Uo =
 prove_thm
  (`pRwf_Half_Slice_Uo`,
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                        Wb Ui phi2' ==>
                                                        Wb Di phi2  ==>
    !t.                                                (isHi phi1 t) ==>
                                                        Def Ui (t+3) ==>
                                                        Def Ui (t+4) ==>
                                                        Def Di (t+3) ==>
                                                        Def Di (t+4) ==>
            (ValAbs Uo(t+3)  =  ValAbs Ui(t+3) /\
                               ~ValAbs St(t+3) /\
                               ~ValAbs Di(t+3)    ) /\
            (ValAbs Uo(t+4)  =  ValAbs Ui(t+4) /\
                               ~ValAbs St(t+4) /\
                               ~ValAbs Di(t+4)    )
   ",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC pRwf_Half_Slice
   THEN EQ_RES_TAC Clock_Sym
   THEN REPEAT_IMP_RES_TAC StatInv_Def
   THEN REPEAT_IMP_RES_TAC StatInv_Wb
   THEN REPEAT_IMP_RES_TAC nNand2_Def
   THEN REPEAT_IMP_RES_TAC Latch_Def_phi2
   THEN REPEAT_IMP_RES_TAC Latch_Wb
   THEN REPEAT_IMP_RES_TAC pNor3_SPEC_phi2
   THEN REPEAT_IMP_RES_TAC StatInv_SPEC
   THEN ASM_REWRITE_TAC [DE_MORGAN_THM]
  );;
```

```
let nRwf_Half_Slice_St =
 prove_thm
  ('nRwf_Half_Slice_St',
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                        Wb Ui phi2  ==>
                                                        Wb Di phi2' ==>
                                                        Wb xd phi1  ==>
                                                        Wb Dx phi1  ==>
    !t.                                                (isHi phi1 t) ==>
                                                         Def St t    ==>
                                                         Def Ui t    ==>
                                                         Def xd(t+1) ==>
                                                         Def xd(t+2) ==>
                                                         Def Dx(t+1) ==>
                                                         Def Dx(t+2) ==>
(ValAbs St(t+3) = (ValAbs St t \/ ~ValAbs Ui t  \/ ~ValAbs Dx(t+2)) /\
                  ((ValAbs St t xor ValAbs Ui t) \/ ~ValAbs xd(t+2))   )/\
(ValAbs St(t+4) = (ValAbs St t \/ ~ValAbs Ui t  \/ ~ValAbs Dx(t+2)) /\
                  ((ValAbs St t xor ValAbs Ui t) \/ ~ValAbs xd(t+2))   )
   ",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC nRwf_Half_Slice
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_Def
   THEN REPEAT_IMP_RES_TAC StatInv_Def
   THEN REPEAT_IMP_RES_TAC Latch_Def
   THEN REPEAT_IMP_RES_TAC pNand2_Def
   THEN REPEAT_IMP_RES_TAC pNand3_Def
   THEN REPEAT_IMP_RES_TAC nNand2_Def
   THEN REPEAT_IMP_RES_TAC StatInv_Wb
   THEN REPEAT_IMP_RES_TAC Latch_Wb
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_Wb
   THEN REPEAT_IMP_RES_TAC pNand3_Wb
   THEN REPEAT_IMP_RES_TAC pNand2_Wb
   THEN REPEAT_IMP_RES_TAC nNand2_Wb
   THEN REPEAT_IMP_RES_TAC StatInv_SPEC
   THEN REPEAT_IMP_RES_TAC Latch_SPEC
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_Ux
   THEN REPEAT_IMP_RES_TAC pNand2_SPEC_2
   THEN REPEAT_IMP_RES_TAC pNand3_SPEC_2
   THEN REPEAT_IMP_RES_TAC nNand2_SPEC_2
   THEN REPEAT_IMP_RES_TAC Latch_SPEC_phi2
   THEN ASM_REWRITE_TAC [DE_MORGAN_THM]
  );;
```

```
let pRwf_Half_Slice_St =
 prove_thm
  (`pRwf_Half_Slice_St`,
   "!phi1 phi1' phi2 phi2'.          Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di xu xd Ux Dx Uo St.
    pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
                                                        Wb Ui phi2' ==>
                                                        Wb Di phi2  ==>
                                                        Wb xd phi1  ==>
                                                        Wb Dx phi1  ==>
    !t.                                                (isHi phi1 t) ==>
                                                        Def St t    ==>
                                                        Def Ui t    ==>
                                                        Def xd(t+1) ==>
                                                        Def xd(t+2) ==>
                                                        Def Dx(t+1) ==>
                                                        Def Dx(t+2) ==>
(ValAbs St(t+3) = (ValAbs St t \/ ~ValAbs Ui t  \/ ~ValAbs Dx(t+2)) /\
                  ((ValAbs St t xor ValAbs Ui t) \/ ~ValAbs xd(t+2))   )/\
(ValAbs St(t+4) = (ValAbs St t \/ ~ValAbs Ui t  \/ ~ValAbs Dx(t+2)) /\
                  ((ValAbs St t xor ValAbs Ui t) \/ ~ValAbs xd(t+2))   )
   ",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC pRwf_Half_Slice
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_Def
   THEN REPEAT_IMP_RES_TAC StatInv_Def
   THEN REPEAT_IMP_RES_TAC Latch_Def
   THEN REPEAT_IMP_RES_TAC pNand2_Def
   THEN REPEAT_IMP_RES_TAC pNand3_Def
   THEN REPEAT_IMP_RES_TAC nNand2_Def
   THEN REPEAT_IMP_RES_TAC StatInv_Wb
   THEN REPEAT_IMP_RES_TAC Latch_Wb
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_Wb
   THEN REPEAT_IMP_RES_TAC pNand3_Wb
   THEN REPEAT_IMP_RES_TAC pNand2_Wb
   THEN REPEAT_IMP_RES_TAC nNand2_Wb
   THEN REPEAT_IMP_RES_TAC StatInv_SPEC
   THEN REPEAT_IMP_RES_TAC Latch_SPEC
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_Ux
   THEN REPEAT_IMP_RES_TAC pNand2_SPEC_2
   THEN REPEAT_IMP_RES_TAC pNand3_SPEC_2
   THEN REPEAT_IMP_RES_TAC nNand2_SPEC_2
   THEN REPEAT_IMP_RES_TAC Latch_SPEC_phi2
   THEN ASM_REWRITE_TAC [DE_MORGAN_THM]
  );;
```

```
let Rwf_Slice_SPEC =
 prove_thm
  ('Rwf_Slice_SPEC',
   "!phi1 phi1' phi2 phi2'.  Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di Uo Do Su Sd.
       Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) ==>
                                              Wb Ui phi2  ==>
                                              Wb Di phi2' ==>
    !t.                                     (isHi phi1 t) ==>
                                              Def Su t    ==>
                                              Def Sd t    ==>
                                              Def Ui t    ==>
                                              Def Ui(t+3) ==>
                                              Def Ui(t+4) ==>
                                              Def Di t    ==>
                                              Def Di(t+3) ==>
                                              Def Di(t+4) ==>
(ValAbs Uo(t+4) = ValAbs Ui(t+4) /\ ~ValAbs Su(t+4) /\ ~ValAbs Di(t+4))/\
(ValAbs Do(t+4) = ValAbs Di(t+4) /\ ~ValAbs Sd(t+4) /\ ~ValAbs Ui(t+4))/\
(ValAbs Su(t+4) =
   (ValAbs Su t \/ ~ValAbs Ui t  \/ (ValAbs Sd t xor ValAbs Di t) ) /\
  ((ValAbs Su t xor ValAbs Ui t) \/ ~ValAbs Sd t  \/ ValAbs Di t  )   )/\
(ValAbs Sd(t+4) =
   (ValAbs Sd t \/ ~ValAbs Di t  \/ (ValAbs Su t xor ValAbs Ui t) ) /\
  ((ValAbs Sd t xor ValAbs Di t) \/ ~ValAbs Su t  \/ ValAbs Ui t  )   )
   ",
   REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
   THEN EQ_RES_THEN STRIP_ASSUME_TAC Rwf_Slice
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_Def
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_Def
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_Wb
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_Wb
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_Ux
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_xu
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_Uo
   THEN REPEAT_IMP_RES_TAC nRwf_Half_Slice_St
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_Ux
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_xu
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_Uo
   THEN REPEAT_IMP_RES_TAC pRwf_Half_Slice_St
   THEN FILTER_ASM_REWRITE_TAC (can (match "x=y"))
                               [xor_CLAUSES; DE_MORGAN_THM]
 );;
```

```
let Rwf_Slice_imp_SLICE =
 prove_thm
  (`Rwf_Slice_imp_SLICE`,
   "!Ui Di Uo Do Su Sd.
     Rwf_Slice_Spec Ui Di Uo Do Su Sd  ==>
                              ~(Uo 0)  ==>
                              ~(Do 0)  ==>
                              ~(Su 0)  ==>
                              ~(Sd 0)  ==>  SLICE (MK_int_mod2a (Ui,Di))
                                                  (MK_int_mod2a (Uo,Do))
                                                  (MK_int_mod2a (Su,Sd))
   ",
   REPEAT STRIP_TAC
   THEN EQ_RES_TAC Rwf_Slice_Spec_EQ
   THEN IMP_RES_TAC (theorem `RWF` `RWF_SLICE_imp_SLICE`)
   );;


let Rwf_Def_when_phi1 =
 prove_thm
  (`Rwf_Def_when_phi1`,
   "!phi1 phi1' phi2 phi2'.
                      Clock(phi1,phi1',phi2,phi2') ==>
    !n Ui Di Uo Do Su Sd.
      Rwf n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd ==>
                  ((!t. ((Def(Su n) o $+3) when (isHi phi1)) t   ) /\
                   (!t. ((Def(Sd n) o $+3) when (isHi phi1)) t   ) /\
                   (!t. ((Def(Uo n) o $+3) when (isHi phi1)) t   ) /\
                   (!t. ((Def(Do n) o $+3) when (isHi phi1)) t   ) /\
                   (!t. ((Def(Su n)      ) when (isHi phi1))(t+1)) /\
                   (!t. ((Def(Sd n)      ) when (isHi phi1))(t+1)) /\
                   (!t. ((Def(Uo n)      ) when (isHi phi1))(t+1)) /\
                   (!t. ((Def(Do n)      ) when (isHi phi1))(t+1))  ) ",
  REPEAT (FILTER_STRIP_TAC "Rwf")
  THEN DISCH_TAC
  THEN PURE_REWRITE_TAC [when]
  THEN BETA_TAC
  THEN IMP_RES_THEN (\th. REWRITE_TAC [ th ; fun_comp ; SPEC "3" ADD_SYM ]
                     ) Clock_TimeOf_isHi_plus4
  THEN CONV_TAC AND_FORALL_CONV
  THEN X_GEN_TAC "t"
  THEN IMP_RES_THEN (ASSUME_TAC o (SPEC "t"))
                     Clock_isHi_TimeOf_isHi
  THEN REPEAT_IMP_RES_TAC Rwf_Def
  THEN ASM_REWRITE_TAC []
 );;
```

```
set_goal([],
   "!phi1 phi1' phi2 phi2'.  Clock(phi1,phi1',phi2,phi2') ==>
    !Ui Di Uo Do Su Sd.
       Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) ==>
                                              Wb Ui phi2  ==>
                                              Wb Di phi2' ==>
                           ((Def Su) when (isHi phi1)) 0  ==>
                           ((Def Sd) when (isHi phi1)) 0  ==>
                (!t.       ((Def Ui) when (isHi phi1)) t) ==>
                (!t.       ((Def Di) when (isHi phi1)) t) ==>
                (!t. ((Def Ui o $+3) when (isHi phi1)) t) ==>
                (!t. ((Def Di o $+3) when (isHi phi1)) t) ==>
    let UI = ((ValAbs Ui) when (isHi phi1))
    in
    let DI = ((ValAbs Di) when (isHi phi1))
    in
    let UO = ((ValAbs Uo) when (isHi phi1))
    in
    let DO = ((ValAbs Do) when (isHi phi1))
    in
    let SU = ((ValAbs Su) when (isHi phi1))
    in
    let SD = ((ValAbs Sd) when (isHi phi1))
    in
    Rwf_Slice_Spec UI DI UO DO ($~ o SU) ($~ o SD)
   ");;



expand (PURE_REWRITE_TAC [LET_DEF; when]
   THEN BETA_TAC
   THEN REPEAT STRIP_TAC
   THEN PURE_REWRITE_TAC [Rwf_Slice_Spec]
   THEN BETA_TAC
   THEN INDUCT_TAC
   THENL [ ALL_TAC ; POP_ASSUM (K ALL_TAC) ]
  );;
```

```
expand (POP_ASSUM (   MP_TAC o (SPEC "0"     )) %____________Case t = 0 %
   THEN POP_ASSUM (   MP_TAC o (SPEC "0"     ))
   THEN POP_ASSUM (\th. MP_TAC (SPEC "0"   th)
                    THEN MP_TAC (SPEC "0+1" th))
   THEN POP_ASSUM (\th. MP_TAC (SPEC "0"   th)
                    THEN MP_TAC (SPEC "0+1" th))
   THEN IMP_RES_THEN
          (\th. REWRITE_TAC [th; fun_comp; SPEC "3" ADD_SYM])
          Clock_TimeOf_isHi_plus4
   THEN IMP_RES_THEN ( ASSUME_TAC o (SPEC "0") )
                     Clock_isHi_TimeOf_isHi
   THEN REPEAT DISCH_TAC
   THEN BETA_TAC
   THEN REPEAT_IMP_RES_TAC Rwf_Slice_SPEC
   THEN IMP_RES_THEN
          (\th. FILTER_ASM_REWRITE_TAC
                   (can (match "x:*=y"))
                   [th; xor_CLAUSES; DE_MORGAN_THM] )
          Clock_TimeOf_isHi_plus4
  );;

expand (POP_ASSUM (   MP_TAC o (SPEC "t+1"     )) %________Case (SUC t) %
   THEN POP_ASSUM (   MP_TAC o (SPEC "t+1"     ))
   THEN POP_ASSUM (\th. MP_TAC (SPEC "t+1"   th)
                   THEN MP_TAC (SPEC "t+1+1" th))
   THEN POP_ASSUM (\th. MP_TAC (SPEC "t+1"   th)
                   THEN MP_TAC (SPEC "t+1+1" th))
   THEN IMP_RES_THEN
          (\th. ASSUME_TAC (SPEC "t" th) THEN MP_TAC (SPEC "t+1" th))
          Clock_isHi_TimeOf_isHi
   THEN IMP_RES_THEN (\th. REWRITE_TAC [ th
                                       ; fun_comp
                                       ; SPEC "3" ADD_SYM
                                       ; ADD1
                                       ; ADD_ASSOC
                                       ]
                     ) Clock_TimeOf_isHi_plus4
   THEN REPEAT_IMP_RES_TAC Rwf_Slice_Def
   THEN REPEAT DISCH_TAC
   THEN BETA_TAC
   THEN REPEAT_IMP_RES_TAC Rwf_Slice_SPEC
   THEN IMP_RES_THEN (\th. FILTER_ASM_REWRITE_TAC
                             (can (match "x:*=y"))
                             [th; xor_CLAUSES; DE_MORGAN_THM]
                     ) Clock_TimeOf_isHi_plus4
  );;

let Rwf_Slice_when_phi1 = save_top_thm `Rwf_Slice_when_phi1`;;
```

```
let SLICE_Correct = SUBS [SYM Rwf_Spec_EQ]
                         (theorem `RWF` `SLICE_CORRECT`);;

let Rwf_Spec_CORRECT =  SUBS [SYM Rwf_Spec_EQ]
                             (theorem `RWF` `RWF_SPEC_CORRECT`);;

let Rwf_Slice_when_phi1' =
    BETA_RULE (REWRITE_RULE [LET_DEF] Rwf_Slice_when_phi1);;

set_goal([],
   "!phi1 phi1' phi2 phi2'.
                         Clock(phi1,phi1',phi2,phi2') ==>
    !n Ui Di Uo Do Su Sd.
         Rwf n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd ==>
                                          Wb Ui phi2  ==>
                                          Wb Di phi2' ==>
       (!n.  ((    Def    (Su n)) when (isHi phi1)) 0) ==>
       (!n.  ((    Def    (Sd n)) when (isHi phi1)) 0) ==>
       (!n.  ((    Def    (Uo n)) when (isHi phi1)) 0) ==>
       (!n.  ((    Def    (Do n)) when (isHi phi1)) 0) ==>
       (!n.  ((    ValAbs (Su n)) when (isHi phi1)) 0) ==>
       (!n.  ((    ValAbs (Sd n)) when (isHi phi1)) 0) ==>
       (!n. ~((    ValAbs (Uo n)) when (isHi phi1)) 0) ==>
       (!n. ~((    ValAbs (Do n)) when (isHi phi1)) 0) ==>
       (!t.  ((Def  Ui          ) when (isHi phi1)) t) ==>
       (!t.  ((Def  Di          ) when (isHi phi1)) t) ==>
       (!t.  ((Def  Ui    o $+3 ) when (isHi phi1)) t) ==>
       (!t.  ((Def  Di    o $+3 ) when (isHi phi1)) t) ==>
       let UI = ValAbs Ui when (isHi phi1)
       in
       let DI = ValAbs Di when (isHi phi1)
       in
       let ip = MK_int_mod2a (UI, DI)
       in
       let UO = ValAbs (Uo n) when (isHi phi1)
       in
       let DO = ValAbs (Do n) when (isHi phi1)
       in
       let op = MK_int_mod2a (UO, DO)
       in
       let SU = (\x. $~ o (ValAbs (Su x) when (isHi phi1)))
       in
       let SD = (\x. $~ o (ValAbs (Sd x) when (isHi phi1)))
       in
       let St = (\x. MK_int_mod2a (SU x, SD x) )
       in
       Rwf_Spec n ip op (VAL n St)
  ");;
```

```
expand (REPEAT GEN_TAC
   THEN DISCH_TAC
   THEN PURE_REWRITE_TAC [LET_DEF]
   THEN BETA_TAC
   THEN BETA_TAC
   THEN INDUCT_TAC
  );;


let Strong_Induction =
 TAC_PROOF
  (([], "!P. P 0 /\ (!n. P(n+1)) ==> !n. P n"),
   GEN_TAC
   THEN STRIP_TAC
   THEN INDUCT_TAC
   THEN ASM_REWRITE_TAC [ADD1]
  );;


% Base Case %
expand (REWRITE_TAC [VAL; SYM (SPEC_ALL SLICE_Correct); Rwf]
   THEN REPEAT (FILTER_STRIP_TAC "Def")
   THEN DISCH_THEN (\th. ASSUME_TAC th THEN ASSUME_TAC (SPEC "0" th))
   THEN DISCH_THEN (\th. ASSUME_TAC th THEN ASSUME_TAC (SPEC "0" th))
   THEN DISCH_THEN (\th. ASSUME_TAC th THEN ASSUME_TAC (SPEC "0" th))
   THEN DISCH_THEN (\th. ASSUME_TAC th THEN ASSUME_TAC (SPEC "0" th))
   THEN DISCH_THEN (\th. ASSUME_TAC th THEN ASSUME_TAC (SPEC "0" th))
   THEN DISCH_THEN (\th. ASSUME_TAC th THEN ASSUME_TAC (SPEC "0" th))
   THEN DISCH_THEN (\th. ASSUME_TAC th THEN ASSUME_TAC (SPEC "0" th))
   THEN DISCH_THEN (\th. ASSUME_TAC th THEN ASSUME_TAC (SPEC "0" th))
   THEN REPEAT STRIP_TAC
   THEN BETA_TAC
   THEN REPEAT_IMP_RES_TAC Rwf_Slice_when_phi1'
   THEN REPEAT_IMP_RES_TAC Rwf_Slice_imp_SLICE
   THEN POP_ASSUM (\th.  REPEAT_IMP_RES_TAC
                           (REWRITE_RULE [fun_comp] th) )
  );;
```

```
% Inductive Case %
expand (POP_ASSUM
         (\th.  PURE_REWRITE_TAC [Rwf; Rwf_Spec_CORRECT]
           THEN REPEAT (GEN_TAC ORELSE DISCH_THEN STRIP_ASSUME_TAC)
           THEN EXISTS_TAC op
           THEN REPEAT_IMP_RES_TAC th
         )
   THEN POP_ASSUM (\th. REWRITE_TAC [th; SYM (SPEC_ALL SLICE_Correct)])
   THEN BETA_TAC
   THEN POP_ASSUM (K ALL_TAC)
   THEN POP_ASSUM (K ALL_TAC)
   THEN POP_ASSUM (K ALL_TAC)
   THEN POP_ASSUM (K ALL_TAC)
   THEN POP_ASSUM
         (\th1.  POP_ASSUM
          \th2.  POP_ASSUM
          \th3.  POP_ASSUM
          \th4.  POP_ASSUM
          \th5.  POP_ASSUM
          \th6.  POP_ASSUM
          \th7.  POP_ASSUM
          \th8.  ASSUME_TAC (SPEC "SUC n" th8)
            THEN ASSUME_TAC (SPEC "SUC n" th7)
            THEN ASSUME_TAC (SPEC     "n" th6)
            THEN ASSUME_TAC (SPEC     "n" th5)
            THEN ASSUME_TAC (SPEC "SUC n" th4)
            THEN ASSUME_TAC (SPEC "SUC n" th3)
            THEN ASSUME_TAC (SPEC "SUC n" th2)
            THEN ASSUME_TAC (SPEC "SUC n" th1)
         )
   THEN IMP_RES_THEN IMP_RES_TAC Rwf_Def_when_phi1
   THEN IMP_RES_THEN IMP_RES_TAC Strong_Induction
   THEN REPEAT_IMP_RES_TAC Rwf_Wb
   THEN REPEAT_IMP_RES_TAC Rwf_Slice_when_phi1'
   THEN REPEAT_IMP_RES_TAC Rwf_Slice_imp_SLICE
   THEN POP_ASSUM (REPEAT_IMP_RES_TAC o (REWRITE_RULE [fun_comp]))
      )
where op = "MK_int_mod2a ( (ValAbs (Uo n)) when (isHi phi1) ,
                           (ValAbs (Do n)) when (isHi phi1) )";;


let Rwf_when_phi1 = save_top_thm `Rwf_when_phi1`;;
```

```
#print_theory `Rwf`;;
The Theory Rwf
Parents --  HOL     gates-lib     RWF
Constants --
  nRwf_Half_Slice
    ":(num -> tri) #
      ((num -> tri) #
       ((num -> tri) #
        ((num -> tri) #
         ((num -> tri) #
          ((num -> tri) #
           ((num -> tri) #
            ((num -> tri) #
             ((num -> tri) # ((num -> tri) # (num -> tri)))))))))) ->
      bool"
  pRwf_Half_Slice
    ":(num -> tri) #
      ((num -> tri) #
       ((num -> tri) #
        ((num -> tri) #
         ((num -> tri) #
          ((num -> tri) #
           ((num -> tri) #
            ((num -> tri) #
             ((num -> tri) # ((num -> tri) # (num -> tri)))))))))) ->
      bool"
  Rwf_Slice
    ":(num -> tri) #
      ((num -> tri) #
       ((num -> tri) #
        ((num -> tri) #
         ((num -> tri) #
          ((num -> tri) #
           ((num -> tri) #
            ((num -> tri) # ((num -> tri) # (num -> tri)))))))))) ->
      bool"
```

```
  Rwf
    ":num ->
      ((num -> tri) ->
       ((num -> tri) ->
        ((num -> tri) ->
         ((num -> tri) ->
          ((num -> tri) ->
           ((num -> tri) ->
            ((num -> (num -> tri)) ->
             ((num -> (num -> tri)) ->
              ((num -> (num -> tri)) -> ((num -> (num -> tri)) -> bool)))))))))"
  Rwf_Half_Slice_Spec
    ":(num -> bool) ->
      ((num -> bool) ->
       ((num -> bool) ->
        ((num -> bool) ->
         ((num -> bool) ->
          ((num -> bool) -> ((num -> bool) -> ((num -> bool) -> bool)))))))"
  Rwf_Slice_Spec
    ":(num -> bool) ->
      ((num -> bool) ->
       ((num -> bool) ->
        ((num -> bool) -> ((num -> bool) -> ((num -> bool) -> bool)))))"
  Rwf_Spec
    ":num ->
      ((num -> int_mod2) ->
       ((num -> int_mod2) -> ((num -> int) -> bool)))"
Definitions --
  nRwf_Half_Slice
    |- !phi1 phi1' phi2 phi2' Ui Di xu xd Ux Dx Uo St.
        nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) =
        (?p1 p2 p3 p4 p5 p6.
          StatInv(Ui,p1) /\
          Latch(phi1,phi1',St,p2) /\
          Latch(phi1,phi1',p1,p3) /\
          nXor(phi1,phi1',p2,p3,Ux) /\
          nNor2(phi1',p2,p3,xu) /\
          pNand3(phi1,p2,p3,Dx,p4) /\
          pNand2(phi1,Ux,xd,p5) /\
          nNand2(phi1',p4,p5,p6) /\
          Latch(phi2,phi2',p6,St) /\
          nNor3(phi2',p1,St,Di,Uo))
```

```
pRwf_Half_Slice
  |- !phi1 phi1' phi2 phi2' Ui Di xu xd Ux Dx Uo St.
      pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) =
      (?p1 p2 p3 p4 p5 p6.
        StatInv(Ui,p1) /\
        Latch(phi1,phi1',St,p2) /\
        Latch(phi1,phi1',p1,p3) /\
        nXor(phi1,phi1',p2,p3,Ux) /\
        nNor2(phi1',p2,p3,xu) /\
        pNand3(phi1,p2,p3,Dx,p4) /\
        pNand2(phi1,Ux,xd,p5) /\
        nNand2(phi1',p4,p5,p6) /\
        Latch(phi2,phi2',p6,St) /\
        pNor3(phi2,p1,St,Di,Uo))
Rwf_Slice
  |- !phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd.
      Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) =
      (?xu xd Ux Dx.
        nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,Su) /\
        pRwf_Half_Slice(phi1,phi1',phi2,phi2',Di,Ui,xd,xu,Dx,Ux,Do,Sd))
Rwf_DEF
  |- Rwf =
     PRIM_REC
     (\phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd.
       Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo 0,Do 0,Su 0,Sd 0))
     (\g00004 n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd.
       g00004 phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd /\
       Rwf_Slice
       (phi1,phi1',phi2,phi2',Uo n,Do n,Uo(SUC n),Do(SUC n),Su(SUC n),
        Sd(SUC n)))
Rwf_Half_Slice_Spec
  |- !Ui Di xu xd Ux Dx Uo St.
      Rwf_Half_Slice_Spec Ui Di xu xd Ux Dx Uo St =
      (!t.
        (Ux t = (St t) xor (Ui t)) /\
        (xu t = ~St t /\ ~Ui t) /\
        (Uo t = Ui t /\ St t /\ ~Di t) /\
        (St(t + 1) =
         St t /\ Ui t /\ Dx t \/ (St t) xor (Ui t) /\ xd t))
```

```
  Rwf_Slice_Spec
    |- !Ui Di Uo Do Su Sd.
        Rwf_Slice_Spec Ui Di Uo Do Su Sd =
        (!t.
          (Uo(t + 1) = Ui(t + 1) /\ Su(t + 1) /\ ~Di(t + 1)) /\
          (Do(t + 1) = Di(t + 1) /\ Sd(t + 1) /\ ~Ui(t + 1)) /\
          (Su(t + 1) =
           Su t /\ Ui t /\ (Sd t) xor (Di t) \/
           (Su t) xor (Ui t) /\ ~Sd t /\ ~Di t) /\
          (Sd(t + 1) =
           Sd t /\ Di t /\ (Su t) xor (Ui t) \/
           (Sd t) xor (Di t) /\ ~Su t /\ ~Ui t))
  Rwf_Spec
    |- !n ip op s.
        Rwf_Spec n ip op s =
        (s 0 = INT 0) /\
        (!t.
          ((NUM((int_mod2_INT ip t) plus (s t))) < (2 EXP (n + 1)) =>
           ((op t = zero) /\ (s(SUC t) = (int_mod2_INT ip t) plus (s t))) |
           ((op t = ip t) /\ (s(SUC t) = INT 0))))

Theorems --
  Rwf
    |- (!phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd.
         Rwf 0 phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd =
         Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo 0,Do 0,Su 0,Sd 0)) /\
       (!n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd.
         Rwf(SUC n)phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd =
         Rwf n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd /\
         Rwf_Slice
         (phi1,phi1',phi2,phi2',Uo n,Do n,Uo(SUC n),Do(SUC n),Su(SUC n),
          Sd(SUC n)))
  Rwf_Half_Slice_Spec_EQ  |- Rwf_Half_Slice_Spec = RWF_HALF_SLICE
  Rwf_Slice_Spec_EQ
    |- !Ui Di Uo Do Su Sd.
        Rwf_Slice_Spec Ui Di Uo Do Su Sd /\
        ~Uo 0 /\
        ~Do 0 /\
        ~Su 0 /\
        ~Sd 0 =
        RWF_SLICE Ui Di Uo Do Su Sd /\ ~Su 0 /\ ~Sd 0
  Rwf_Spec_EQ  |- Rwf_Spec = RWF_SPEC
```

```
nRwf_Half_Slice_Wb
  |- !phi1 phi1' phi2 phi2'.
       Clock(phi1,phi1',phi2,phi2') ==>
       (!Ui Di xu xd Ux Dx Uo St.
         nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
         Wb St phi2 /\
         Wb St phi2' /\
         Wb Uo phi2 /\
         Wb xu phi1 /\
         Wb Ux phi1)
pRwf_Half_Slice_Wb
  |- !phi1 phi1' phi2 phi2'.
       Clock(phi1,phi1',phi2,phi2') ==>
       (!Ui Di xu xd Ux Dx Uo St.
         pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
         Wb St phi2 /\
         Wb St phi2' /\
         Wb Uo phi2' /\
         Wb xu phi1 /\
         Wb Ux phi1)
Rwf_Slice_Wb
  |- !phi1 phi1' phi2 phi2'.
       Clock(phi1,phi1',phi2,phi2') ==>
       (!Ui Di Uo Do Su Sd.
         Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) ==>
         Wb Su phi2 /\ Wb Sd phi2' /\ Wb Uo phi2 /\ Wb Do phi2')
nRwf_Half_Slice_Def
  |- !phi1 phi1' phi2 phi2'.
       Clock(phi1,phi1',phi2,phi2') ==>
       (!Ui Di xu xd Ux Dx Uo St.
         nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
         (!t. isHi phi1 t ==>
              Def St(t + 2) /\
              Def St(t + 3) /\
              Def St(t + 4) /\
              Def St(t + 5) /\
              Def Uo(t + 2) /\
              Def Uo(t + 3) /\
              Def Uo(t + 4) /\
              Def Uo(t + 5) /\
              Def xu t /\
              Def xu(t + 1) /\
              Def xu(t + 2) /\
              Def xu(t + 3) /\
              Def Ux t /\
              Def Ux(t + 1) /\
              Def Ux(t + 2) /\
              Def Ux(t + 3)))
```

```
pRwf_Half_Slice_Def
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di xu xd Ux Dx Uo St.
        pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
        (!t.
          isHi phi1 t ==>
          Def St(t + 2) /\
          Def St(t + 3) /\
          Def St(t + 4) /\
          Def St(t + 5) /\
          Def Uo(t + 2) /\
          Def Uo(t + 3) /\
          Def Uo(t + 4) /\
          Def Uo(t + 5) /\
          Def xu t /\
          Def xu(t + 1) /\
          Def xu(t + 2) /\
          Def xu(t + 3) /\
          Def Ux t /\
          Def Ux(t + 1) /\
          Def Ux(t + 2) /\
          Def Ux(t + 3)))
Rwf_Slice_Def
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di Uo Do Su Sd.
        Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) ==>
        (!t.
          isHi phi1 t ==>
          Def Su(t + 2) /\
          Def Su(t + 3) /\
          Def Su(t + 4) /\
          Def Su(t + 5) /\
          Def Sd(t + 2) /\
          Def Sd(t + 3) /\
          Def Sd(t + 4) /\
          Def Sd(t + 5) /\
          Def Uo(t + 2) /\
          Def Uo(t + 3) /\
          Def Uo(t + 4) /\
          Def Uo(t + 5) /\
          Def Do(t + 2) /\
          Def Do(t + 3) /\
          Def Do(t + 4) /\
          Def Do(t + 5)))
```

```
Rwf_Wb
  |- !phi1 phi1' phi2 phi2'.
       Clock(phi1,phi1',phi2,phi2') ==>
       (!n Ui Di Uo Do Su Sd.
         Rwf n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd ==>
         Wb(Su n)phi2 /\ Wb(Sd n)phi2' /\ Wb(Uo n)phi2 /\ Wb(Do n)phi2')
Rwf_Def
  |- !phi1 phi1' phi2 phi2'.
       Clock(phi1,phi1',phi2,phi2') ==>
       (!n Ui Di Uo Do Su Sd.
         Rwf n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd ==>
         (!t.
           isHi phi1 t ==>
           Def(Su n)(t + 2) /\
           Def(Su n)(t + 3) /\
           Def(Su n)(t + 4) /\
           Def(Su n)(t + 5) /\
           Def(Sd n)(t + 2) /\
           Def(Sd n)(t + 3) /\
           Def(Sd n)(t + 4) /\
           Def(Sd n)(t + 5) /\
           Def(Uo n)(t + 2) /\
           Def(Uo n)(t + 3) /\
           Def(Uo n)(t + 4) /\
           Def(Uo n)(t + 5) /\
           Def(Do n)(t + 2) /\
           Def(Do n)(t + 3) /\
           Def(Do n)(t + 4) /\
           Def(Do n)(t + 5)))
nRwf_Half_Slice_Ux
  |- !phi1 phi1' phi2 phi2'.
       Clock(phi1,phi1',phi2,phi2') ==>
       (!Ui Di xu xd Ux Dx Uo St.
         nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
         Wb Ui phi2 ==>
         Wb Di phi2' ==>
         Wb xd phi1 ==>
         Wb Dx phi1 ==>
         (!t.
           isHi phi1 t ==>
           Def Ui t ==>
           Def St t ==>
           (ValAbs Ux(t + 1) = ~(ValAbs St t) xor (ValAbs Ui t)) /\
           (ValAbs Ux(t + 2) = ~(ValAbs St t) xor (ValAbs Ui t))))
```

```
pRwf_Half_Slice_Ux
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di xu xd Ux Dx Uo St.
        pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
        Wb Ui phi2' ==>
        Wb Di phi2 ==>
        Wb xd phi1 ==>
        Wb Dx phi1 ==>
        (!t.
          isHi phi1 t ==>
          Def Ui t ==>
          Def St t ==>
          (ValAbs Ux(t + 1) = ~(ValAbs St t) xor (ValAbs Ui t)) /\
          (ValAbs Ux(t + 2) = ~(ValAbs St t) xor (ValAbs Ui t))))
nRwf_Half_Slice_xu
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di xu xd Ux Dx Uo St.
        nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
        Wb Ui phi2 ==>
        Wb Di phi2' ==>
        Wb xd phi1 ==>
        Wb Dx phi1 ==>
        (!t.
          isHi phi1 t ==>
          Def Ui t ==>
          Def St t ==>
          (ValAbs xu(t + 1) = ValAbs St t /\ ~ValAbs Ui t) /\
          (ValAbs xu(t + 2) = ValAbs St t /\ ~ValAbs Ui t)))
pRwf_Half_Slice_xu
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di xu xd Ux Dx Uo St.
        pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
        Wb Ui phi2' ==>
        Wb Di phi2 ==>
        Wb xd phi1 ==>
        Wb Dx phi1 ==>
        (!t.
          isHi phi1 t ==>
          Def Ui t ==>
          Def St t ==>
          (ValAbs xu(t + 1) = ValAbs St t /\ ~ValAbs Ui t) /\
          (ValAbs xu(t + 2) = ValAbs St t /\ ~ValAbs Ui t)))
```

```
nRwf_Half_Slice_Uo
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di xu xd Ux Dx Uo St.
        nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
        Wb Ui phi2 ==>
        Wb Di phi2' ==>
        (!t.
          isHi phi1 t ==>
          Def Ui(t + 3) ==>
          Def Ui(t + 4) ==>
          Def Di(t + 3) ==>
          Def Di(t + 4) ==>
          (ValAbs Uo(t + 3) =
           ValAbs Ui(t + 3) /\ ~ValAbs St(t + 3) /\ ~ValAbs Di(t + 3)) /\
          (ValAbs Uo(t + 4) =
           ValAbs Ui(t + 4) /\ ~ValAbs St(t + 4) /\ ~ValAbs Di(t + 4))))
pRwf_Half_Slice_Uo
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di xu xd Ux Dx Uo St.
        pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
        Wb Ui phi2' ==>
        Wb Di phi2 ==>
        (!t.
          isHi phi1 t ==>
          Def Ui(t + 3) ==>
          Def Ui(t + 4) ==>
          Def Di(t + 3) ==>
          Def Di(t + 4) ==>
          (ValAbs Uo(t + 3) =
           ValAbs Ui(t + 3) /\ ~ValAbs St(t + 3) /\ ~ValAbs Di(t + 3)) /\
          (ValAbs Uo(t + 4) =
           ValAbs Ui(t + 4) /\ ~ValAbs St(t + 4) /\ ~ValAbs Di(t + 4))))
```

```
nRwf_Half_Slice_St
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di xu xd Ux Dx Uo St.
        nRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
        Wb Ui phi2 ==>
        Wb Di phi2' ==>
        Wb xd phi1 ==>
        Wb Dx phi1 ==>
        (!t.
          isHi phi1 t ==>
          Def St t ==>
          Def Ui t ==>
          Def xd(t + 1) ==>
          Def xd(t + 2) ==>
          Def Dx(t + 1) ==>
          Def Dx(t + 2) ==>
          (ValAbs St(t + 3) =
           (ValAbs St t \/ ~ValAbs Ui t \/ ~ValAbs Dx(t + 2)) /\
           ((ValAbs St t) xor (ValAbs Ui t) \/ ~ValAbs xd(t + 2))) /\
          (ValAbs St(t + 4) =
           (ValAbs St t \/ ~ValAbs Ui t \/ ~ValAbs Dx(t + 2)) /\
           ((ValAbs St t) xor (ValAbs Ui t) \/ ~ValAbs xd(t + 2)))))
pRwf_Half_Slice_St
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di xu xd Ux Dx Uo St.
        pRwf_Half_Slice(phi1,phi1',phi2,phi2',Ui,Di,xu,xd,Ux,Dx,Uo,St) ==>
        Wb Ui phi2' ==>
        Wb Di phi2 ==>
        Wb xd phi1 ==>
        Wb Dx phi1 ==>
        (!t.
          isHi phi1 t ==>
          Def St t ==>
          Def Ui t ==>
          Def xd(t + 1) ==>
          Def xd(t + 2) ==>
          Def Dx(t + 1) ==>
          Def Dx(t + 2) ==>
          (ValAbs St(t + 3) =
           (ValAbs St t \/ ~ValAbs Ui t \/ ~ValAbs Dx(t + 2)) /\
           ((ValAbs St t) xor (ValAbs Ui t) \/ ~ValAbs xd(t + 2))) /\
          (ValAbs St(t + 4) =
           (ValAbs St t \/ ~ValAbs Ui t \/ ~ValAbs Dx(t + 2)) /\
           ((ValAbs St t) xor (ValAbs Ui t) \/ ~ValAbs xd(t + 2)))))
```

```
Rwf_Slice_SPEC
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di Uo Do Su Sd.
        Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) ==>
        Wb Ui phi2 ==>
        Wb Di phi2' ==>
        (!t.
          isHi phi1 t ==>
          Def Su t ==>
          Def Sd t ==>
          Def Ui t ==>
          Def Ui(t + 3) ==>
          Def Ui(t + 4) ==>
          Def Di t ==>
          Def Di(t + 3) ==>
          Def Di(t + 4) ==>
          (ValAbs Uo(t + 4) =
           ValAbs Ui(t + 4) /\ ~ValAbs Su(t + 4) /\ ~ValAbs Di(t + 4)) /\
          (ValAbs Do(t + 4) =
           ValAbs Di(t + 4) /\ ~ValAbs Sd(t + 4) /\ ~ValAbs Ui(t + 4)) /\
          (ValAbs Su(t + 4) =
           (ValAbs Su t \/
            ~ValAbs Ui t \/
            (ValAbs Sd t) xor (ValAbs Di t)) /\
           ((ValAbs Su t) xor (ValAbs Ui t) \/
            ~ValAbs Sd t \/
            ValAbs Di t)) /\
          (ValAbs Sd(t + 4) =
           (ValAbs Sd t \/
            ~ValAbs Di t \/
            (ValAbs Su t) xor (ValAbs Ui t)) /\
           ((ValAbs Sd t) xor (ValAbs Di t) \/
            ~ValAbs Su t \/
            ValAbs Ui t))))
Rwf_Slice_imp_SLICE
  |- !Ui Di Uo Do Su Sd.
      Rwf_Slice_Spec Ui Di Uo Do Su Sd ==>
      ~Uo 0 ==>
      ~Do 0 ==>
      ~Su 0 ==>
      ~Sd 0 ==>
      SLICE
      (MK_int_mod2a(Ui,Di))
      (MK_int_mod2a(Uo,Do))
      (MK_int_mod2a(Su,Sd))
```

```
Rwf_Def_when_phi1
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!n Ui Di Uo Do Su Sd.
        Rwf n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd ==>
        (!t. (((Def(Su n)) o ($+ 3)) when (isHi phi1))t) /\
        (!t. (((Def(Sd n)) o ($+ 3)) when (isHi phi1))t) /\
        (!t. (((Def(Uo n)) o ($+ 3)) when (isHi phi1))t) /\
        (!t. (((Def(Do n)) o ($+ 3)) when (isHi phi1))t) /\
        (!t. ((Def(Su n)) when (isHi phi1))(t + 1)) /\
        (!t. ((Def(Sd n)) when (isHi phi1))(t + 1)) /\
        (!t. ((Def(Uo n)) when (isHi phi1))(t + 1)) /\
        (!t. ((Def(Do n)) when (isHi phi1))(t + 1)))
Rwf_Slice_when_phi1
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!Ui Di Uo Do Su Sd.
        Rwf_Slice(phi1,phi1',phi2,phi2',Ui,Di,Uo,Do,Su,Sd) ==>
        Wb Ui phi2 ==>
        Wb Di phi2' ==>
        ((Def Su) when (isHi phi1))0 ==>
        ((Def Sd) when (isHi phi1))0 ==>
        (!t. ((Def Ui) when (isHi phi1))t) ==>
        (!t. ((Def Di) when (isHi phi1))t) ==>
        (!t. (((Def Ui) o ($+ 3)) when (isHi phi1))t) ==>
        (!t. (((Def Di) o ($+ 3)) when (isHi phi1))t) ==>
        let UI = (ValAbs Ui) when (isHi phi1)
        in
        let DI = (ValAbs Di) when (isHi phi1)
        in
        let UO = (ValAbs Uo) when (isHi phi1)
        in
        let DO = (ValAbs Do) when (isHi phi1)
        in
        let SU = (ValAbs Su) when (isHi phi1)
        in
        let SD = (ValAbs Sd) when (isHi phi1)
        in
        Rwf_Slice_Spec UI DI UO DO($~ o SU)($~ o SD))
```

```
Rwf_when_phi1
  |- !phi1 phi1' phi2 phi2'.
      Clock(phi1,phi1',phi2,phi2') ==>
      (!n Ui Di Uo Do Su Sd.
        Rwf n phi1 phi1' phi2 phi2' Ui Di Uo Do Su Sd ==>
        Wb Ui phi2 ==>
        Wb Di phi2' ==>
        (!n. ((Def(Su n)) when (isHi phi1))0) ==>
        (!n. ((Def(Sd n)) when (isHi phi1))0) ==>
        (!n. ((Def(Uo n)) when (isHi phi1))0) ==>
        (!n. ((Def(Do n)) when (isHi phi1))0) ==>
        (!n. ((ValAbs(Su n)) when (isHi phi1))0) ==>
        (!n. ((ValAbs(Sd n)) when (isHi phi1))0) ==>
        (!n. ~((ValAbs(Uo n)) when (isHi phi1))0) ==>
        (!n. ~((ValAbs(Do n)) when (isHi phi1))0) ==>
        (!t. ((Def Ui) when (isHi phi1))t) ==>
        (!t. ((Def Di) when (isHi phi1))t) ==>
        (!t. (((Def Ui) o ($+ 3)) when (isHi phi1))t) ==>
        (!t. (((Def Di) o ($+ 3)) when (isHi phi1))t) ==>
        let UI = (ValAbs Ui) when (isHi phi1)
        in
        let DI = (ValAbs Di) when (isHi phi1)
        in
        let ip = MK_int_mod2a(UI,DI)
        in
        let UO = (ValAbs(Uo n)) when (isHi phi1)
        in
        let DO = (ValAbs(Do n)) when (isHi phi1)
        in
        let op = MK_int_mod2a(UO,DO)
        in
        let SU = \x. $~ o ((ValAbs(Su x)) when (isHi phi1))
        in
        let SD = \x. $~ o ((ValAbs(Sd x)) when (isHi phi1))
        in
        let St = \x. MK_int_mod2a(SU x,SD x)
        in
        Rwf_Spec n ip op(VAL n St))

************************************
() : void

#
```