# Formal Verification of Asynchronous Data-Path Circuits 

 byDavid T. Weih<br>B.Sc., University of British Columbia, 1992<br>\title{ A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF<br><br>Master of Science in THE FACULTY OF GRADUATE STUDIES<br><br>(Department of Computer Science) }<br>We accept this thesis as conforming to the required standard

## The University of British Columbia

April 1996
(c) David T. Weih, 1996

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, 1 agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada
Date April $26 \quad 1996$

## Abstract

Asynchronous designs are typically modelled with non-deterministic next-state relations. When a deterministic model is available, specialised verification techniques can be used to automatically verify relatively large designs. This thesis demonstrates that verification techniques developed for deterministic models can be applied to speed-independent self-timed circuits. We introduce local formulas, and show that the validity of a local formula is independent of the order in which the components of a speed-independent circuit perform their operations. Local formulas provide a natural way to specify the input/output behaviour of a circuit and are well-suited for specifying data-paths. We demonstrate the practicality of our approach by describing the verification of three designs: a FIFO, an adder, and a vector multiplier chip.

## Contents

Abstract ..... ii
Contents ..... iii
List of Figures ..... v
Acknowledgements ..... vi
Dedication ..... viii
1 Introduction ..... 1
1.1 Formal Verification Techniques ..... 2
1.1.1 Theorem Proving ..... 3
1.1.2 OBDD Based Model Checking ..... 4
1.1.3 Symbolic Trajectory Evaluation ..... 5
1.1.4 The Voss Prover ..... 6
1.2 Automatic Verification of Asynchronous Data-Path Circuits ..... 7
1.3 Related Work ..... 8
1.4 Overview of Thesis ..... 10
2 Theory ..... 12
2.1 The Circuit Model ..... 12
2.2 Definitions ..... 16
2.3 A Specification for the FIFO ..... 23
2.4 Properties of Speed-Independent Circuits ..... 25
2.5 Discussion of Hypotheses ..... 30
3 Local and Global Formulas ..... 32
3.1 Local Formulas ..... 33
3.2 Global Formulas and Period Functions ..... 35
3.3 Unspecifiable Behaviour ..... 39
4 Implementation ..... 40
4.1 VHDL Circuit Descriptions ..... 40
4.2 Symbolic Trajectory Evaluation ..... 42
4.3 Model Checking for Speed-Independence ..... 46
5 Vector Multiplier ..... 48
6 Conclusions ..... 53
6.1 Future Work ..... 54
Bibliography ..... 57
Appendix A Proofs ..... 62
Appendix B Source Code ..... 74

## List of Figures

2.1 A Self-Timed FIFO ..... 13
2.2 Splicing Traces ..... 21
3.1 4 Bit Adders ..... 33
3.2 Adder Environment ..... 34
3.3 Pipelined Asynchronous Adder ..... 37
3.4 Global Formula for Pipelined Adder ..... 38
3.5 Global Formula for Flat Adder ..... 38
4.1 VHDL Code for Source Component ..... 43
4.2 VHDL Code for Sink Component ..... 44
4.3 VHDL Code for FIFO Cell ..... 45
5.1 Bit Slice of the MAS Block ..... 50
6.1 Vector Multiplier: Abstract State Machine ..... 55

## Acknowledgements

I owe a great debt.to Mark Greenstreet, who was a tremendously helpful supervisor. Some of the material in this thesis was prepared in collaboration with Mark as a paper for IEE Proceedings - Computer and Digital Techniques. The theoretical framework presented in Chapter 2 owes its clarity to his suggestions.

I would also like to acknowledge the ISD group for making hardware verification a lively and social pursuit. In particular, Mike Donat and Mark Aagaard let me practice explaining my ideas to them, while Andy Martin essentially took a two day holiday from his own work to help me prove a theorem.

Finally, I would like to thank my family and other friends who supported me in many ways during my thesis. My parents and grandparents encouraged me and provided some very practical support. Mike Wilson and Jen Weih gave me with the opportunity to occasionally escape from work. And last but not least, deep thanks to Andrea, without whom nothing would have been the same.

David T. Wein

The University of British Columbia
April 1996

To my beloved wife-to-be,
Andrea Wadman

## Chapter 1

## Introduction

Many techniques have been developed for formal hardware verification. There has been a need for this variety because each method has strengths and weaknesses which restrict it to a particular subclass of verification tasks. The verification method described in this thesis has capabilities not found in existing techniques. In particular, this method allows automatic verification of asynchronous data-path circuits.

Synchronous and asynchronous designs differ primarily in the timing models used to describe their behaviour. In a pure synchronous design, computation is performed by combinational logic and storage is performed by clocked latches. In this framework, the state of the latches after a clock event is a deterministic function of the values held in the latches before the clock event. Specialised verification methods, such as symbolic trajectory evaluation (STE), exploit the fact that synchronous designs have deterministic next state functions which allow a symbolic "next state" operator to be implemented efficiently. This allows large synchronous designs to be formally verified with a high degree of automation [Bea93, Dar94].

Asynchronous designs, on the other hand, do not use a clock to determine the
timing of state changing operations. Because the timing relations between circuit components cannot be completely determined, asynchronous designs are usually modelled using non-deterministic next-state relations. Symbolic model checkers have been used to verify small designs; however, the state-space explosion problem prevents fully automated verification on the same scale that has been achieved for synchronous designs.

This thesis presents a formal framework in which asynchronous circuits can be verified using synchronous models. The key observation is that for speedindependent circuits without output choice the sequence of operations performed by any component depends only on the original state of the circuit and any input values applied. The exact relative ordering of component actions doesn't matter. This motivates specifying the desired behaviour in terms of input and output sequences for each component, which is a natural specification for data-path circuits. Because the synchronous implementation of the circuit corresponds to choosing a specific ordering for component actions, verifying the input/output sequences of the components in a synchronous model guarantees that the asynchronous circuit will produce equivalent sequences.

### 1.1 Formal Verification Techniques

In order to set this work in context, a brief overview of the major categories of verification techniques is provided. Much of the information contained in this section is drawn from [Gup92] and [Seg92]. Most methods can be divided into two categories, theorem proving and model checking. Because of their relevance to this work, two particular methods are included in this discussion: symbolic trajectory evaluation (STE) which is based on an efficient OBDD-based model checker; and
the Voss Prover which combines STE with a theorem proving framework.

### 1.1.1 Theorem Proving

The process of verification by theorem proving typically proceeds by describing both the specification and the structure of the hardware in terms of formal logic and then formally showing a relationship (such as implication or equivalence) between them. Automated theorem provers (such as HOL [MG93]) have been developed to assist the creation of the required proof. In order to grasp the size of this task, it is important to realize that a formal proof consists of a finite sequence of statements, where each statement is either one of the fundamental axioms provided by the theorem prover or can be derived from the previous statements by a primitive rule of inference. The collection of these axioms and inference rules is typically quite small (5 axioms and 8 inference rules for HOL) requiring a very large number of steps in order to prove theorems. For example, a proof showing that the combination of a NAND gate and an inverter is equivalent to an AND gate takes a sequence of 365 statements.

Also, while these theorem provers are partially automated, in order to prove theorems of any size, human guidance is required. This interaction may be required approximately once for every 1000 proof steps. Despite the large multiplier, we can see from the number of lines required for even a trivial example that considerable human assistance is required for larger proofs. A further drawback is that providing this assistance requires that users possess a high level of training in formal logic.

However, theorem provers are extremely powerful, and there are many examples of their use in verifying both hardware and software systems [ $\mathrm{OR}^{+} 95$ ]. Theorem proving has two main advantages which help make it a powerful technique. First, being based on a general mathematical logic (e.g. higher order logic in HOL), theo-
rem provers can be used in a wide variety of different situations. Second, it is well suited to hierarchical methods. This is extremely important when dealing with large systems and systems (such as hardware) which contain various levels with different levels of detail (for example, verifying an arithmetic unit made up of functional blocks which are made up of gates).

### 1.1.2 OBDD Based Model Checking

Model checking is a technique which has strengths and weaknesses that are almost exactly the reverse of those of theorem proving. Model checking proceeds by describing the system as a finite state machine. That is, for a system with $d$ boolean valued signals it creates the state space $2^{d}$ and a state transition relation $R \subseteq 2^{d} \times 2^{d}$. Given an initial state predicate, $Q_{0}$, and a desired property, $p$, verification proceeds by exhaustively searching the states reachable from $Q_{0}$ under $R$ for states where $p$ is false. Thus, it provides a verification of the behaviour of the system described by the model provided, as opposed to theorem proving which shows functional or structural equivalence between the specification and implementation, both of which are described in the same language. One advantage of the model checking approach is that a single specification may be used to verify different implementations with the same intended functionality simply by repeating the decision procedure. This can be contrasted with theorem proving where the proof will typically need to be reformulated to match the new implementation.

This process typically has a completely automatic decision procedure, allowing users with little understanding of the underlying theory to use the system to verify properties. It can also provide counterexamples to failed assertions, which is a significant aid to designers whose fundamental goal is to create a correct system
(rather than simply to determine the correctness of the current design).
There are significant drawbacks to model checking. The first is that the specifications are restricted to being simple enumerations of the desired properties, and it can be difficult to determine whether the specification completely captures the desired properties of the system. The second problem is the fact that the complete model must be described before the system may be verified. The state is "described" rather than constructed, because symbolic techniques using OBDDs [Bry86] have been developed which represent the model and its transition relation as functions from sets of states to sets of states which allow the verification of systems with large state spaces (for example $\left[\mathrm{BC}^{+} 94\right]$ with more than $10^{120}$ states for a synchronous pipelined ALU and $10^{50}$ states for an asynchronous stack). However, the state space must still be described in some manner, which limits the size of systems that can be verified. However, model checking is well-suited to verifying control circuitry, which will typically have a complex state relation but relatively few states.

### 1.1.3 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation (STE) is a model checking based technique which traces its roots back to symbolic simulation. The inclusion of OBDDs significantly increased the power of symbolic simulation while retaining the ability to model low-level properties such as bounded delays and charge sharing. To reduce the impact of large internal state, a third logic value $X$ is added to indicate an unknown value for a node. This allows the system state to be placed in a lattice, and efficient algorithms have been developed for verifying large designs in this framework. In this method, the stimulus to the system and the required behaviour must be described in a restricted form of temporal logic. STE is a process for showing that the trace
of states simulated implies the trace of states described by the consequent.
This is a relatively efficient verification method, allowing the verification of large circuits, such as a 32 -bit pipelined RISC processor [Dar94]. Like model checking, the specifications are typically behavioural, allowing reuse of specifications for different implementations, and the user does not need to have a complete understanding of the details of the internal model in order to use the tool effectively.

However, to realize efficient symbolic simulation STE depends on the existence of a deterministic next state function. It is also limited by the properties of the underlying OBDDs. For example, there is no variable ordering which allows multiplication of bit vectors to be represented by a diagram of a size which is less than exponential in the number of bits. This is a "hard" limit in the capabilities of STE, since (like model checking) it works directly on the circuits states and does not lend itself to hierarchical methods. Furthermore, the lattice structure which allows efficient verification also restricts the class of formulas which can be checked.

### 1.1.4 The Voss Prover

The Voss Prover combines the powerful low-level capabilities of STE with a simple theorem prover [HS95]. This allows STE results to be combined together to show the correctness of circuits too large for STE to verify on its own.

Theorem proving is limited by the extremely small steps which it must take in deriving proofs and STE is limited by the OBDDs which form its underlying representation. The Voss Prover solves these problems by using the verification results provided by trajectory evaluation as components which are manipulated to form proofs. A fully programmable script language is provided for writing proofs. This language includes abstract representations for data (such as integers) which
free the proof from the boolean representation provided by trajectory evaluation, and allows human intervention to take place at an extremely high level. As opposed to HOL, where each proof step must either be an axiom or the application of an inference rule, the Voss Prover has specialised rules for composing STE results built in, as well as the ability to include limited domain knowledge. This makes proofs shorter and easier for the operator to understand. An example is the verification of a pipelined IEEE 64-bit floating-point multiplier [AS95].

### 1.2 Automatic Verification of Asynchronous Data-Path Circuits

The goal of this research was to develop a method for automatically verifying asynchronous data-path circuits. With this in mind, let us review some key properties of the above verification techniques.

## Theorem Proving

Despite its strengths, theorem proving is not suitable for our purposes because it is not automatic. There are partially-automated theorem provers, but they still require significant manual effort by users with considerable mathematical sophistication.

## Model Checking

Model checking is automatic, but general model checking on its own it is not sufficient because its complexity can be exponential in the size of the circuit being verified. This causes a state-space explosion for data-path circuits which typically have a large number of state-holding nodes.

## Symbolic Trajectory Evaluation

This specialised form of model checking takes advantage of a deterministic next-state function to allow efficient verification. It is still automatic, but can not handle the non-determinism of an asynchronous circuit.

## Voss Prover

While the Voss Prover is a powerful extension of STE in many respects, it still does not handle non-deterministic next state relations.

From the above summary we can see that theorem proving does not provide an automatic decision procedure, that we can not use model checking on its own, because of the large state space required to describe data-path circuits, nor can we simply use trajectory evaluation or the Voss Prover, because the next state relation of an asynchronous circuit is non-deterministic.

For this work model checking was used to show speed-independence, which was used to justify transforming the asynchronous circuits into equivalent synchronous models. Then STE was used to verify the synchronous models. The theoretical justification for this transformation is presented in chapter 2. Although the implementation described here uses OBDD based model checking and STE, the theoretical framework presented could be applied to other methods as well.

### 1.3 Related Work

This work is motivated by the observation that the current state of a speed-independent circuit depends only on the initial state of the circuit, its inputs, and how many times each component has performed an action. Muller and Bartky [MB59] used similar ideas when they described circuits whose states could be
modelled on a semi-modular lattice. The handling of traces is influenced by Udding's thesis [Udd84], where he classifies systems based on the reordering of traces. While he uses this as a method to describe input and output choice, we use the equivalence of reorderings of traces to simplify verification. Using a synchronous model for an asynchronous circuit can be seen as an extreme example of the reduction of the next state relation described in [Pel93].

The circuit model presented excludes designs with arbiters, and we note that many published examples of asynchronous designs meet this requirement. The Caltech Asynchronous Processor [Mar89a] and the self-timed divider of Williams [Wi191] are other examples of self-timed designs that do not use arbiters. A notable exception to this is the Counter Flow Pipeline Processor [SSM94] in which arbiters are used to coordinate the interactions of adjacent pipeline stages. Another well-known published design which is not tractable by this technique is the micropipeline [Sut89], which combines delay-insensitive control circuitry with a bounded-delay data-path. A more general theoretical framework than the one presented in this paper would be required to verify either of these last two designs.

This work requires that designs be shown to be speed-independent before they can be verified. Here this step is performed using model checking and the composition of specifications [AL90], but there are several design methodologies which ensure that the circuit produced is speed-independent (or delay-insensitive, which is stronger).

The survey [Hau95] provides an overview of design methods for delay-insensitive circuits. Petri net-type methods [Mur89] are used as a basis for many of these methods. One is I-nets [MFR85], which can describe systems which are not delay-insensitive, but where delay-insensitivity can be shown by the Foam

Rubber Wrapper constraint [MFR85]. Another method is Signal Transition Graphs [CLW85], where various levels of restriction can be placed on the graphs in return for more efficient automatic synthesis of a circuit. Finally, Change Diagrams $\left[\mathrm{KK}^{+} 92\right]$ are similar to STGs, but avoid some of the restrictions while still maintaining efficient verification of design properties and circuit synthesis.

Martin has implemented a compilation technique [Mar89b] that uses a language similar to Communicating Sequential Processes and compiles it into speedindependent circuits. Van Berkel [vB92] uses the language Tangram in a similar manner. As opposed to Martin and Van Berkel who create their own speedindependent components as required, Brunvald and Sproull [BS89] use a subset of Occam (a language based on CSP) and provide a delay-insensitive module implementing each of the language constructs. Ebergen [Ebe91] uses traces to describe both the specification and the design, and can automatically reduce the specification to a design built up of small number of basic components (such as wire, fork, Celement, etc.). Lastly, Spars $\varnothing$ and Staunstrup [SS93] describe a method for designing multi-ring circuits from building blocks which are speed-independent by construction. A vector multiplier from their paper is verified as the major example in this thesis. A speed-independent circuit produced by any these techniques is amenable to verification using the method presented in this thesis.

### 1.4 Overview of Thesis

Chapter 2 describes the theoretical framework and results which justify the use of synchronous verification tools for asynchronous circuits. Chapter 3 provides a detailed discussion of the specification method used. Chapter 4 describes how the process was enacted for the examples, and Chapter 5 presents a large example circuit
which was verified to show the capabilities of this method. Chapter 6 provides conclusions and suggests possible areas for future work.

## Chapter 2

## Theory

This chapter introduces the theoretical underpinnings of this work. The key theorems are stated in section 2.4 along with sketches of their proofs. Detailed proofs are provided in appendix A. To motivate the theorems, a FIFO is used as a simple example. The circuit model is introduced in section 2.1, and the necessary definitions are laid out in section 2.2. These are used to give the specification for the FIFO in section 2.3. Finally, section 2.5 provides a brief discussion relating the hypotheses required in the theoretical results to circuit design.

### 2.1 The Circuit Model

We model a circuit as a set of nodes $V$, a state transition relation $R$, a set of circuit components $C$, a set of environment components $E$, and an initial state predicate $Q_{0}$. Each node corresponds to a boolean-valued signal. A state is an assignment of values to nodes; we write $S$ to denote the circuit's state space, where $S=2^{V}$. For $v$ in $V$ and $s$ in $S$, we write $v(s)$ to denote the value of node $v$ in state $s$. The components provide a decomposition of $V$ and $R$ described shortly.


Figure 2.1: A Self-Timed FIFO

Figure 2.1 depicts a simple four-stage FIFO that is used as an example throughout this chapter. The FIFO is one-bit wide and uses a dual-rail coding [Sei79], where the signal carries its own completion information. In this case, a single bit $x$ is encoded by a pair of wires ( $x . t, x . f$ ), with ( $T, F$ ) representing true, $(F, T)$ representing false, and $(F, F)$ as an extra value empty which is used to separate data values. Thus, the transition from $(F, F)$ to $(T, F)$ encodes the arrival of a true value. The acknowledgement part of the handshake cycle takes place on the request line $r$. This is raised when a stage has passed an empty value, and lowered after it has passed a data value.

This circuit has fifteen nodes, twelve circuit components, and two environment components. We assume that it is initialised to the all-empty state.

$$
\begin{align*}
V & =\{\mathrm{x}[i] \cdot \mathrm{t}, \mathrm{x}[i] \cdot \mathrm{f}, \mathrm{x}[i] \cdot \mathrm{r} \mid i \in\{0 \ldots 4\}\} \\
C & =\{c 1 t, c 1 f, n 1, c 2 t, c 2 f, n 2, c 3 t, c 3 f, n 3, c 4 t, c 4 f, n 4\} \\
E & =\{s r c, s n k\}  \tag{2.1}\\
Q_{0} & =\bigwedge_{i=0, \ldots .4} \neg \mathrm{x}[i] \cdot \mathrm{t} \wedge \neg \mathrm{x}[i] \cdot \mathrm{f} \wedge \mathrm{x}[i] \cdot \mathrm{r}
\end{align*}
$$

Each component $c$ in $C \cup E$ has a set of inputs $I_{c}$ and a set of outputs $O_{c}$, where $I_{c}$ and $O_{c}$ are subsets of $V$. Furthermore, we assume that each node is connected to exactly one output. In other words, the outputs of the components in
$C \cup E$ partition $V$. The behaviour of component $c$ is given by the state transition relation $R_{c}$. To simplify the presentation, we exclude vacuous transitions: i.e. if $\left(s, s^{\prime}\right)$ is a state transition in $R_{c}$, then $s \neq s^{\prime}$. Only outputs of $c$ change in a transition in $R_{c}$,

$$
\begin{equation*}
\forall\left(s, s^{\prime}\right) \in R_{c} . \forall v \in V .\left(v \in O_{c}\right) \vee\left(v(s)=v\left(s^{\prime}\right)\right) \tag{2.2}
\end{equation*}
$$

Furthermore, the behaviour of a component depends only on the value of its inputs and outputs. If $s^{1}$ and $s^{2}$ are two states and $c$ is a component such that all inputs and outputs of $c$ have the same value in both $s^{1}$ and $s^{2}$, then if $c$ can perform an action in $s^{1}$ it can perform an equivalent action from $s^{2}$. More formally,

$$
\begin{align*}
\forall s^{1}, s^{2}, s^{3} \in S . & \left(\left(\forall v \in I_{c} \cup O_{c} \cdot v\left(s^{1}\right)=v\left(s^{2}\right)\right) \wedge\left(s^{1}, s^{3}\right) \in R_{c}\right)  \tag{2.3}\\
& \Rightarrow \exists s^{4} \in S .\left(\left(s^{2}, s^{4}\right) \in R_{c} \wedge\left(\forall v \in O_{c} \cdot v\left(s^{3}\right)=v\left(s^{4}\right)\right)\right)
\end{align*}
$$

A component is enabled in state $s$ if there is another state $s^{\prime}$ such that $\left(s, s^{\prime}\right)$ is in $R_{c}$.

$$
\begin{equation*}
\operatorname{enabled}(c, s) \equiv \exists s^{\prime} .\left(s, s^{\prime}\right) \in R_{c} \tag{2.4}
\end{equation*}
$$

It follows from equation 2.3 that if there are two states, such that $c$ is enabled in one but not the other, then these states must differ in the value of some input or output of $c$.

$$
\forall s, s^{\prime} \in S .\left(\left(\operatorname{enabled}(c, s) \wedge \neg \operatorname{enabled}\left(c, s^{\prime}\right)\right) \Rightarrow\left(\exists v \in I_{c} \cup O_{c} . v(s) \neq v\left(s^{\prime}\right)\right)\right)
$$

Continuing the FIFO example, the input set, $I_{c 1 t}$ of C-element c1t from figure 2.1 is $\{\mathrm{x}[0] . \mathrm{t}, \mathrm{x}[1] . \mathrm{r}\}$ and the output set, $O_{c 1 t}$ is $\{\mathrm{x}[1] . \mathrm{t}\}$. State transition
relations for components $c 1 t, s r c$, and $s n k$, are given below,

$$
\begin{aligned}
& R_{c 1 t}=\left\{\left(s^{1}, s^{2}\right) \in S \times S \mid\right. \\
& \left(\mathrm{x}[1] \cdot \mathrm{r}\left(s^{1}\right)=\mathrm{x}[0] \cdot \mathrm{t}\left(s^{1}\right)\right) \wedge\left(\mathrm{x}[1] . \mathrm{t}\left(s^{1}\right) \neq \mathrm{x}[0] . \mathrm{t}\left(s^{1}\right)\right) \\
& \wedge\left(\mathrm{x}[1] . \mathrm{t}\left(s^{2}\right)=\mathrm{x}[0] . \mathrm{t}\left(s^{1}\right)\right) \\
& \wedge \forall v \in(V-\{\mathrm{x}[1] \cdot \mathrm{t}\}) \cdot v\left(s^{2}\right)=v\left(s^{1}\right) \\
& \text { \} } \\
& R_{s r c}=\left\{\left(s^{1}, s^{2}\right) \in S \times S \mid\right. \\
& \left.\left(\mathrm{x}[0] . \mathrm{t}\left(s^{1}\right) \vee \mathrm{x}[0] . \mathrm{f}\left(s^{1}\right)\right)\right) \neq\left(\mathrm{x}[0] . \mathrm{r}\left(s^{1}\right)\right. \\
& \left.\wedge\left(\mathrm{x}[0] . \mathrm{t}\left(s^{2}\right) \vee \mathrm{x}[0] . \mathrm{f}\left(s^{2}\right)\right)\right)=\left(\mathrm{x}[0] \cdot \mathrm{r}\left(s^{1}\right)\right. \\
& \wedge \neg\left(\mathrm{x}[0] . \mathrm{t}\left(s^{2}\right) \wedge \mathrm{x}[0] . \mathrm{f}\left(s^{2}\right)\right) \\
& \wedge \forall v \in(V-\{\mathbf{x}[0] . \mathrm{t}, \mathrm{x}[0] . \mathrm{f}\}) . v\left(s^{2}\right)=v\left(s^{1}\right) \\
& \text { \} } \\
& R_{s n k}=\left\{\left(s^{1}, s^{2}\right) \in S \times S \mid\right. \\
& \left(\mathrm{x}[4] . \mathrm{r}\left(s^{1}\right)=\left(\mathrm{x}[4] . \mathrm{t}\left(s^{1}\right) \vee \mathrm{x}[4] . \mathrm{f}\left(s^{1}\right)\right)\right) \\
& \wedge\left(x[4] . r\left(s^{2}\right) \neq\left(\mathrm{x}[4] . \mathrm{t}\left(s^{1}\right) \vee \mathrm{x}[4] . \mathrm{f}\left(s^{1}\right)\right)\right) \\
& \wedge \quad \forall v \in(V-\{\mathrm{x}[4] \cdot \mathrm{r}\}) \cdot v\left(s^{2}\right)=v\left(s^{1}\right) \\
& \text { \} }
\end{aligned}
$$

where $S=2^{V}$ with $V$ from equation 2.1.
Given a set of nodes $V$, a set of circuit components $C$, and a set of environment components $E$, we define the state transition relation for the circuit and its environment as the union of the state transition relations for each component. Let $R$ denote this state transition relation.

$$
\begin{equation*}
R=\bigcup_{c \in C \cup E} R_{c} \tag{2.6}
\end{equation*}
$$

If more than one component is enabled in state $s$, then $s$ can have several possible
successors. This non-determinism corresponds to unspecified delays for the components. Furthermore, each successor corresponds to the action of a single component. This gives an "interleaved semantics" for the circuit. Section 2.4 shows that allowing simultaneous actions of different components does not affect the properties which can be specified in local formulas.

To verify properties of a circuit, we consider assertions about reachable states. The set of reachable states is given by the circuit's initial state predicate $Q_{0}$ and its state transition relation $R$. A trace is a sequence of states such that the first state in the sequence satisfies $Q_{0}$ and each pair of successive states in the sequence is in $R$. We will write $\operatorname{traces}\left(R, Q_{0}\right)$ to denote the set of all traces which can be formed using the transition relation $R$ and whose initial state satisfies the predicate $Q_{0}$. If $t$ is a trace, we write $t^{0}$ to denote the initial (first) state of $t$, and $t^{k}$ to denote the $k+1^{s t}$ element of the sequence. A trace is either infinite or ends in a terminal state, where $s \in S$ is terminal if there is no state $s^{\prime}$ such that $\left(s, s^{\prime}\right) \in R$. A state $s$ is reachable if there is a trace $t$ such that $s$ appears in $t$. The predicate reachable $\left(R, Q_{0}, s\right)$ denotes that $s$ is reachable by a circuit and environment with combined state transition relation $R$, starting from a state satisfying $Q_{0}$.

### 2.2 Definitions

This thesis considers speed-independent circuits whose circuit components have no output choice and whose environment components have stable inputs. A circuit is speed-independent if once a component is enabled to change its output(s), it remains enabled to perform that change until it performs some action. A component has output choice if it can be enabled to perform one of two or more mutually exclusive actions. A component has stable inputs if once it is enabled, the values
of its input nodes remain unchanged until it performs an action. Discussion is restricted to circuits that are speed-independent, with circuit components that do not have output choice, and where the environment components have stable inputs. A circuit which has these properties will be called an eligible circuit.

Definition 1 (Speed-Independence) Given a circuit ( $V, R, C, E, Q_{0}$ ), the circuit and its environment are speed-independent iff

$$
\begin{aligned}
& \forall c \in C \cup E . \forall\left(s^{1}, s^{2}\right) \in R_{c} . \forall s^{3} \in S . \\
& \quad\left(\left(s^{1}, s^{3}\right) \in R-R_{c}\right) \wedge \operatorname{reachable}\left(R, Q_{0}, s^{1}\right) \Rightarrow \\
& \quad\left(\exists s^{4} \in S .\left(\left(s^{3}, s^{4}\right) \in R_{c}\right) \wedge\left(\forall v \in O_{c} \cdot\left(v\left(s^{2}\right)=v\left(s^{4}\right)\right)\right)\right)
\end{aligned}
$$

Definition 2 (Output Choice) Given a circuit ( $V, R, C, E, Q_{0}$ ) and a component $c \in C \cup E, c$ has output choice iff

$$
\exists\left(s, s^{1}\right),\left(s, s^{2}\right) \in R_{c} . \text { reachable }\left(R, Q_{0}, s\right) \wedge\left(s^{1} \neq s^{2}\right)
$$

Definition 3 (Stable Inputs) Given a circuit ( $V, R, C, E, Q_{0}$ ) and a component $c \in C \cup E, c$ has stable inputs iff

$$
\begin{aligned}
& \forall\left(s, s^{\prime}\right) \in R .\left(\operatorname{reachable}\left(R, Q_{0}, s\right) \wedge \operatorname{enabled}(c, s)\right) \Rightarrow \\
& \left(\left(s, s^{\prime}\right) \in R_{c}\right) \vee\left(\forall v \in I_{c} . v(s)=v\left(s^{\prime}\right)\right)
\end{aligned}
$$

Note that if $\left(s, s^{\prime}\right) \notin R_{c}$ then $c$ 's outputs cannot change either, by the definition of outputs given in equation 2.2.

Definition 4 (Eligible Circuits) A circuit $\left(V, R, C, E, Q_{0}\right)$ is eligible iff it is speedindependent, no component in $C$ has output choice, and every component in $E$ has stable inputs.

Section 4.3 describes how the FIFO from figure 2.1 can be shown to be speed-independent and that all of its components have stable inputs. Only the src
component has output choice: from a state where $x[0] . t$ and $x[0] . f$ are false and $x[0] . r$ is true, the source may set either $x[0] . t$ or $x[0] . f$ to true, but not both (see equation 2.5). The FIFO is an example of an eligible circuit.

The next five definitions describe technical properties of traces that are needed for the proofs in section 2.4 and appendix A. A trace is weakly fair if every persistently enabled component eventually performs an action. The occur and index functions give the number and locations of occurrences of actions by a particular component in a trace. Two traces are choice consistent if each environment component outputs the same sequence of values given the same sequence of inputs. Splicing is an operation that combines two traces while preserving properties of local formulas and weak-fairness.

Definition 5 (Weakly Fair Trace) Given a circuit ( $V, R, C, E, Q_{0}$ ), a trace $t$ is weakly fair iff

$$
\begin{aligned}
& \forall c \in C \cup E . \forall i \in N . \text { enabled }\left(c, t^{i}\right) \Rightarrow \\
& \quad \exists j \in N .(j \geq i) \wedge\left(\left(t^{j}, t^{j+1}\right) \in R_{c}\right) \vee \neg \operatorname{enabled}\left(c, t^{j}\right)
\end{aligned}
$$

Note that for a weakly fair trace of a speed-independent circuit, any enabled component eventually performs an action.

Definition 6 (Occurrence of a Transition) Given a component c, a trace $t$, and a natural number $k$, occur $(c, t, m)$ denotes the number of times that $c$ occurs in $t^{0}, \ldots, t^{m}$, formally defined as follows.

$$
\operatorname{occur}(c, t, 0) \equiv 0
$$

$$
\operatorname{occur}(c, t, m) \equiv \operatorname{occur}(c, t, m-1) \quad \text { if } m>0 \text { and } \forall v \in O_{c} \cdot v\left(t^{m-1}\right)=v\left(t^{m}\right)
$$

$$
\operatorname{occur}(c, t, m) \equiv \operatorname{occur}(c, t, m-1)+1 \quad \text { if } m>0 \text { and } \exists v \in O_{c} \cdot\left(v\left(t^{m-1}\right) \neq v\left(t^{m}\right)\right.
$$

Definition 7 (Index of a Transition) Given a component c, a trace $t$, and a natural number $k$, index $(c, t, k)$ denotes the location of the $k^{\text {th }}$ transition of component $c$ in trace $t$. That is,

$$
\operatorname{index}(c, t, k) \equiv \min \{m \in N \mid \operatorname{occur}(c, t, m)=k\}
$$

Let $t$ be a trace of a circuit that includes a component $c$. We write $t_{c}^{1}$ to denote the state immediately before the first time component $c$ effects a state transition in $t$ and $t_{c}^{1}$.post to denote the state immediately after this transition. More generally, $t_{c}^{k}=t^{\text {index }(c, t, k)-1}$ and $t_{c}^{k} \cdot p o s t=t^{\text {index }(c, t, k)}$. Since the components' output sets partition $V$, there can be no change on any $v \in O_{c}$ from index $(c, t, k)$ to index $(c, t, k+1)-1$. This means that assertions about $t_{c}^{k}$.post can be expressed using $t_{c}^{k+1}$. As a result, assertions about a node's inputs at the time of its $k^{\text {th }}$ action are written about the state $t_{c}^{k}$ while assertions about the outputs of the $k^{\text {th }}$ action are commonly written about the state $t_{c}^{k+1}$. The notation $v_{c}^{k}(t)$ denotes $v\left(t_{c}^{k}\right)$, and $\operatorname{valid}\left(v_{c}^{k}, t\right)$ indicates that component $c$ is enabled at least $k$ times in $t$.

Definition 8 (Choice Consistent Traces) Given an eligible circuit $\left(V, R, C, E, Q_{0}\right)$, let $c$ be a component in $E$ and let $t$ and $u$ be traces. The actions of $c$ are choice consistent in $t$ and $u\left(w r i t t e n t \cong_{c} u\right)$ if, given the same
history of inputs, $c$ produces the same outputs in both traces. More formally,

$$
t \cong_{c} u \equiv\left(\left(t^{0}=u^{0}\right) \wedge \forall k \in N . \operatorname{PreEquiv}(c, t, u, k)\right) \Rightarrow \operatorname{PostEquiv}(c, t, u, k)
$$

where

$$
\begin{gathered}
\operatorname{PreEquiv}(c, t, u, k) \equiv \forall v \in I_{c} \cup O_{c} . \forall j \in\{1 \ldots k\} \\
\operatorname{valid}\left(v_{c}^{j}, t\right)=\operatorname{valid}\left(v_{c}^{j}, u\right) \\
\wedge \quad \operatorname{valid}\left(v_{c}^{j}, t\right) \Rightarrow\left(v\left(t_{c}^{j}\right)=v\left(u_{c}^{j}\right)\right) \\
\operatorname{PostEquiv}(c, t, u, k) \equiv \forall v \in O_{c} . \forall j \in\{1 \ldots k\} \\
\\
\quad\left(\operatorname{valid}\left(v_{c}^{j}, t\right) \Rightarrow\left(v\left(t_{c}^{j} \cdot p o s t\right)=v\left(u_{c}^{j} . p o s t\right)\right)\right)
\end{gathered}
$$

If $D$ is a set of components, $t \cong \cong_{D} u$ denotes $\forall d \in D . t \cong_{d} u$. Consistency is an equivalence relation.

If $t$ is a trace, each pair of successive states in $t$ can be associated with the component that effected the transition. Given two choice consistent traces $t$ and $u$, splice $(t, u, m)$ produces a sequence where the first $m$ transitions are effected as in $t$, the corresponding transitions are deleted from $u$, and the remainder of $u$ is appended to the prefix from $t$. Theorem 1 will show that splice $(t, u, m)$ is a trace. Figure 2.2 shows two traces and their first six splices (where $c^{i}$ denotes the $i^{\text {th }}$ action by component $c$ ).

Definition 9 (Splicing Traces) Given an eligible circuit ( $V, R, C, E, Q_{0}$ ), let $t$ and $u$ be traces with $t^{0}=u^{0}$ and $t \cong_{E} u$. Let $m$ be a natural number such that $t^{m}$ exists. The $m^{\text {th }}$ splice of $t$ and $u$ is defined inductively. Let $c_{m}$ be the component which makes its $n^{\text {th }}$ action going from the state $t^{m-1}$ to the state $t^{m}$, and let $i_{m}$ be the index of the point where $c_{m}$ makes its $n^{\text {th }}$ action in splice $(t, u, m-1)$, if $c_{m}$ ever

| $u$ | $:$ | $c^{1}$ | $a^{1}$ | $d^{1}$ | $c^{2}$ | $d^{2}$ | $b^{1}$ | $c^{3}$ | $a^{2}$ | $b^{2}$ | $d^{3}$ | $a^{3}$ | $c^{4}$ | $\ldots$ |
| ---: | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| splice $(t, u, 0)$ | $:$ | $c^{1}$ | $a^{1}$ | $d^{1}$ | $c^{2}$ | $d^{2}$ | $b^{1}$ | $c^{3}$ | $a^{2}$ | $b^{2}$ | $d^{3}$ | $a^{3}$ | $c^{4}$ | $\ldots$ |
| splice $(t, u, 1)$ | $:$ | $a^{1}$ | $c^{1}$ | $d^{1}$ | $c^{2}$ | $d^{2}$ | $b^{1}$ | $c^{3}$ | $a^{2}$ | $b^{2}$ | $d^{3}$ | $a^{3}$ | $c^{4}$ | $\ldots$ |
| splice $(t, u, 2):$ | $a^{1}$ | $b^{1}$ | $c^{1}$ | $d^{1}$ | $c^{2}$ | $d^{2}$ | $c^{3}$ | $a^{2}$ | $b^{2}$ | $d^{3}$ | $a^{3}$ | $c^{4}$ | $\ldots$ |  |
| splice $(t, u, 3):$ | $a^{1}$ | $b^{1}$ | $c^{1}$ | $d^{1}$ | $c^{2}$ | $d^{2}$ | $c^{3}$ | $a^{2}$ | $b^{2}$ | $d^{3}$ | $a^{3}$ | $c^{4}$ | $\ldots$ |  |
| splice $(t, u, 4):$ | $a^{1}$ | $b^{1}$ | $c^{1}$ | $d^{1}$ | $c^{2}$ | $d^{2}$ | $c^{3}$ | $a^{2}$ | $b^{2}$ | $d^{3}$ | $a^{3}$ | $c^{4}$ | $\ldots$ |  |
| splice $(t, u, 5):$ | $a^{1}$ | $b^{1}$ | $c^{1}$ | $d^{1}$ | $a^{2}$ | $c^{2}$ | $d^{2}$ | $c^{3}$ | $b^{2}$ | $d^{3}$ | $a^{3}$ | $c^{4}$ | $\ldots$ |  |
| $\vdots$ |  | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ |
|  | $t:$ | $a^{1}$ | $b^{1}$ | $c^{1}$ | $d^{1}$ | $a^{2}$ | $b^{2}$ | $c^{2}$ | $d^{2}$ | $a^{3}$ | $b^{3}$ | $c^{3}$ | $d^{3}$ |  |

Figure 2.2: Splicing Traces
makes such an action. Formally,

$$
\begin{aligned}
& \left(t^{m-1}, t^{m}\right) \in R_{c_{m}} \\
& n=\operatorname{occur}\left(c_{m}, \text { splice }(t, u, m-1), m-1\right)+1 \\
& i_{m}=\text { index }\left(c_{m}, \text { splice }(t, u, m-1), n\right) \text { if it exists }
\end{aligned}
$$

If $i_{m}$ exists, then splice $(t, u, m)$ is the sequence that satisfies the following equations

$$
\begin{array}{rlrl}
\text { splice }(t, u, 0) & \equiv u & & \\
v\left(\text { splice }(t, u, m)^{k}\right) & \equiv v\left(t^{k}\right) & & \text { if } k<m \text { and } v \in V \\
v\left(\text { splice }(t, u, m)^{k}\right) & \equiv v\left(t^{m}\right) & & \text { if } m \leq k<i_{m}, \text { and } v \in O_{c_{m}} \\
v\left(\text { splice }(t, u, m)^{k}\right) & \equiv v\left(\text { splice }(t, u, m-1)^{k-1}\right) & \text { if } m \leq k<i_{m}, \text { and } v \notin O_{c_{m}} \\
v\left(\text { splice }(t, u, m)^{k}\right) & \equiv v\left(\text { splice }(t, u, m-1)^{k}\right) & & \text { if } i_{m} \leq k
\end{array}
$$

if $i_{m}$ does not exist then splice $(t, u, m)$ is not defined.

The following property gives a sufficient condition for splice $(t, u, m)$ to be defined, and it is shown in theorem 1 of section 2.4 that splices are traces.

Property 1 (Splicing) Given an eligible circuit ( $V, R, C, E, Q_{0}$ ), let $t$ and $u$ be traces with $t^{0}=u^{0}, t \cong_{E} u$. If $t$ is weakly fair, then splice $(t, u, m)$ exists.

The combination of weak fairness and speed-independence ensures that $i_{m}$ exists.

The next four definitions describe the properties of traces that are used for specification and verification. Local formulas refer to the values of the inputs and outputs of a component when the component performs its $k^{t h}$ action, specifying properties of bounded prefixes of traces. Two traces are equivalent if they have the same truth values for all local formulas. Global formulas are predicates over states. Periodic global formulas and shifted local formulas are used to specify safety properties of unbounded traces.

Definition 10 (Local Formula) A local formula is either

$$
\begin{array}{lll} 
& v_{c}^{k} & \text { where } v \in I_{c} \cup O_{c}, k \in N \text {, and } c \text { is a component with stable inputs } \\
\text { or } & \neg f & \text { where } f \text { is a local formula } \\
\text { or } & f \wedge g & \text { where } f \text { and } g \text { are local formulas. }
\end{array}
$$

A local formula $L$ is valid for trace $t$ if $\operatorname{valid}\left(v_{c}^{k}, t\right)$ is true for each term of the form $v_{c}^{k}$ in $L$. If $L$ is valid for trace $t$, then it holds if the values of the nodes of the circuit in trace $t$ satisfy the formula. The predicates $\operatorname{valid}(L, t)$ and $L(t)$ denote that $L$ is valid for trace $t$ and that $L$ holds for trace $t$ respectively, and the notation $\operatorname{comp}(L)$ indicates the set of components which appear in the local formula. Finally, $\left(R, Q_{0}\right) \models L$ denotes that $L$ is valid and holds for every weakly fair trace of the circuit with state transition relation $R$ and initial state predicate $Q_{0}$.

Definition 11 (Equivalent Traces) Two traces $t$ and $u$ with $t^{0}=u^{0}$ are equivalent iff for all local formulas $L$

$$
(\operatorname{valid}(L, t)=\operatorname{valid}(L, u)) \bigwedge(\operatorname{valid}(L, t) \Rightarrow(L(t)=L(u)))
$$

The notation $t \cong u$ denotes that the traces $t$ and $u$ are equivalent. Because environment components of an eligible circuit have stable inputs, $t \cong u \Rightarrow t \cong_{E} u$.

Definition 12 (Global Formula) A global formula is a predicate over states.
Definition 13 (Shifted local formula) Let $L$ be a local formula and $\operatorname{comp}(L)$ be the set of components that appear in terms of $L$. Let $i$ be a natural number and $p$ be a function from $\operatorname{comp}(L)$ to the naturals. The shift of $L$ by itimes $p$ is obtained by replacing every term of the form $v_{c}^{k}$ in $L$ with $v_{c}^{i * p(c)+k}$.

The notation $\operatorname{shift}(L, p, i)$ denotes the shift of $L$ by $i$ times $p$.
Definition 14 (Periodic Global Formula) Let ( $V, R, C, E, Q_{0}$ ) be an eligible circuit. Let $G$ be a global formula, let $C^{\prime} \subseteq C \cup E$ be a set of components, and let $p$ be a function from $C^{\prime}$ to the natural numbers, with $p(c)>0$ for all $c \in C^{\prime} . G$ has period $p$ if only components in $C^{\prime}$ are enabled in states that satisfy $G$, and for every trace whose initial state satisfies $G$ there is a choice consistent trace that returns to a state satisfying $G$ after each component $c$ in $C^{\prime}$ has performed $p(c)$ actions. More formally, $G$ is periodic with period $p$ iff

$$
\begin{gathered}
\forall s \in S . \forall c \in(C \cup E)-C^{\prime} . G(s) \Rightarrow \neg \operatorname{enabled}(c, s) \\
\wedge \quad \forall t \in \operatorname{traces}(R, G) . \exists u \in \operatorname{traces}(R, G) . \exists m \in N . \\
\\
\left(t \cong_{E} u\right) \wedge G\left(u^{m}\right) \wedge\left(\forall c \in C^{\prime} . \operatorname{occur}(c, u, m)=p(c)\right)
\end{gathered}
$$

The predicate periodic $\left(G, C^{\prime}, p\right)$ indicates that the global formula $G$ is periodic with period $p$ on the components in the set $C^{\prime}$.

### 2.3 A Specification for the FIFO

Local formulas provide a natural way to specify the input/output behaviour of speedindependent circuits. For example, a natural specification for the FIFO shown in
figure 2.1 is that the infinite sequence of values output by the source must be the same as the sequence received by the sink. This is expressed by the formula

$$
\forall i \in N .\left((\mathbf{x}[0] \cdot t)_{s r c}^{i} \cdot p o s t=\mathrm{x}[4] \cdot \mathrm{t}_{s n k}^{i}\right) \wedge\left((\mathrm{x}[0] \cdot \mathrm{f})_{s r c}^{i} \cdot p o s t=\mathrm{x}[4] \cdot \mathrm{f}_{s n k}^{i}\right)
$$

which is equivalent to (dropping the post notation)

$$
\begin{equation*}
\forall i \in N .\left(\mathrm{x}[0] \cdot \mathrm{t}_{s r c}^{i+1}=\mathrm{x}[4] \cdot \mathrm{t}_{s n k}^{i}\right) \wedge\left(\mathrm{x}[0] \cdot \mathrm{f}_{s r c}^{i+1}=\mathrm{x}[4] \cdot \mathrm{f}_{s n k}^{i}\right) \tag{2.7}
\end{equation*}
$$

Recall from the discussion after definition 7 that $x[0] . t_{s r c}^{i+1}$ refers the value of $\mathrm{x}[0] . \mathrm{t}$ immediately before the src component performs its $i+1^{\text {st }}$ operation. This is the value that was output by the source at its $i^{\text {th }}$ operation. On the other hand, x [4]. $\mathrm{t}_{s n k}^{i}$ is the value that was input to the $s n k$ component at its $i^{\text {th }}$ operation.

Local formulas can only specify properties with a finite number of terms. Periodic global formulas extend the result to infinite sequences. For the FIFO example, it is convenient to assume that the FIFO is initially reset to a state where all stages hold empty values (as in equation 2.1), and consider a period function that returns the FIFO to this state. Because the source alternates between sending empty and full values, the source must perform two operations per period, as must the sink. This motivates the following choices of $G$ and $p$.

$$
\begin{align*}
p(s r c) & \equiv 2 \\
p(s n k) & \equiv 2  \tag{2.8}\\
G & \equiv Q_{0}
\end{align*}
$$

where $Q_{0}$ is given in equation 2.1. Note that this specification does not say that the FIFO periodically returns to an empty state for all traces. It merely requires that there exists at least one trace for which the FIFO has this behaviour. Let $L$ be the
local formula defined below.

$$
\begin{align*}
L \equiv & \left(x[0] \cdot \mathrm{t}_{s r c}^{2}=\mathrm{x}[4] \cdot \mathrm{t}_{s n k}^{1}\right) \wedge\left(\mathrm{x}[0] \cdot \mathrm{f}_{s r c}^{2}=\mathrm{x}[4] \cdot \mathrm{f}_{s n k}^{1}\right)  \tag{2.9}\\
& \wedge\left(\mathrm{x}[0] \cdot \mathrm{t}_{s r c}^{3}=\mathrm{x}[4] \cdot \mathrm{t}_{s n k}^{2}\right) \wedge\left(\mathrm{x}[0] \cdot \mathrm{f}_{s r c}^{3}=\mathrm{x}[4] \cdot \mathrm{f}_{s n k}^{2}\right)
\end{align*}
$$

Intuitively, the first line states that the data is propagated correctly, and the second states that the empty is propagated correctly. Theorem 3 in the next section shows that establishing that $(R, G) \models L$ and that $G$ has period $P$ is sufficient to establish $\forall i \in N .(R, G) \models \operatorname{shift}(L, p, i)$, which for $G, L$, and $p$ as defined above is equivalent to the specification given in formula 2.7. Since the formula $L$ asserts that the first value output by the circuit is equal to the first value supplied, showing $\operatorname{shift}(L, p, i)$ shows that for any $i$, the $i^{\text {th }}$ value output by the circuit is equal to the $i^{\text {th }}$ value supplied, which verifies correct FIFO behaviour.

### 2.4 Properties of Speed-Independent Circuits

The splice operation has two properties that provide the foundation for this work. First, the splice of two weakly fair traces is a weakly fair trace. Second, splicing preserves the values of local formulas. These properties motivate the following theorem.

Theorem 1 (Splicing) Given an eligible circuit ( $V, R, C, E, Q_{0}$ ), let $t$ and $u$ be weakly fair traces with $m$ a natural number such that $t^{m}$ exists. If $t^{0}=u^{0}$ and $t \cong_{E} u$, then splice $(t, u, m)$ is a weakly fair trace and splice $(t, u, m) \cong u$.

Proof (sketch, see appendix A for details): The key observation is that the actions of concurrently enabled components commute without changing the values of local formulas or affecting the property of weak fairness. Let $\alpha$ be a trace, $i$ be a natural
number, and $c$ and $d$ be components such that $\left(\alpha^{i}, \alpha^{i+1}\right) \in R_{c},\left(\alpha^{i+1}, \alpha^{i+2}\right) \in R_{d}$, and $d$ is enabled in state $\alpha^{i}$. Let $\beta$ be the sequence such that

$$
\begin{aligned}
\beta^{k} & =\alpha^{k} & & \text { if } k \neq i+1 \\
v\left(\beta^{i+1}\right) & =v\left(\alpha^{i+2}\right) & & \text { if } v \in O_{d} \\
v\left(\beta^{i+1}\right) & =v\left(\alpha^{i}\right) & & \text { if } v \notin O_{d}
\end{aligned}
$$

Note that any component that has output choice must be an environment component and therefore has stable inputs. This and speed-independence ensure that $\left(\beta^{i}, \beta^{i+1}\right) \in R_{d}$ and $\left(\beta^{i+1}, \beta^{i+2}\right) \in R_{c}$. Thus, $\beta$ is a trace. Given an arbitrary component $e$ with stable inputs, it is shown below that $v_{e}^{k}$ has the same validity and value in traces $\alpha$ and $\beta$ by case analysis on index $(e, \alpha, k)$, and that if $\alpha$ is weakly fair then $\beta$ is weakly fair as well.

The theorem is proven by induction on $m$. The base case is $m=0$, which is discharged by noting that splice $(t, u, 0)=u$. For $m>0$, note that the component that effects the transition from $t^{m-1}$ to $t^{m}$ must be enabled in state splice $(t, u, m-1)^{m-1}$. By the induction hypothesis, $\operatorname{splice}(t, u, m-1)$ is weakly fair and $t \cong_{E}$ splice $(t, u, m-1)$; thus, this component must eventually perform an action, and any environment choices must be identical in both traces. Therefore, the sequence $\operatorname{splice}(t, u, m)$ can be obtained from trace splice $(t, u, m-1)$ by a finite number of commuting operations, and the claims of the theorem follow from the properties of commuting.

The key property of speed-independent circuits that we exploit is that if two traces for a circuit start in the same state and the environment provides the same inputs, then all components with stable inputs have identical input and output sequences in both traces. This equivalence is formalised by observing that both
traces satisfy the same local formulas.

Theorem 2 (Equivalence of Traces) Given an eligible circuit ( $V, R, C, E, Q_{0}$ ), if $t$ and $u$ are weakly fair traces with $t^{0}=u^{0}$ and $t \cong{ }_{E} u$, then $t \cong u$.

Proof: Because every local formula $L$ can be represented by terms of the form $v_{c}^{k}$ and the operators $\neg$ and $\wedge$, if every term of the form $v_{c}^{k}$ has the same validity and value in $t$ and $u$ then, by induction on the structure of $L$, the theorem holds.

Let $v_{c}^{k}$ be a term of $L$, where $c$ is a component with stable inputs. If $v_{c}^{k}$ is valid in trace $t$, let $\alpha=\operatorname{splice}(t, u$, index $(c, t, k))$. By the definition of splicing, $v_{c}^{k}$ is valid in both $t$ and $\alpha$, and $v_{c}^{k}(t)=v_{c}^{k}(\alpha)$. By theorem $1, v_{c}^{k}(\alpha)=v_{c}^{k}(u)$. This establishes $v_{c}^{k}(t)=v_{c}^{k}(u)$ as required. A symmetric argument applies if $v_{c}^{k}$ is valid in $u$. Finally, the case where $v_{c}^{k}$ is not valid in either $t$ or $u$ also satisfies the claim of the theorem.

Theorem 2 provides the basis for verifying specifications of the form $\left(R, Q_{0}\right) \vDash L$. If the circuit satisfies the hypotheses of theorem 2 , it is only necessary to verify a single trace for each initial state and sequence of inputs from the environment to show that all possible traces starting from an allowable initial state and with suitable environment behaviour will satisfy $L$. Chapter 4 describes how this can be shown using STE. However, a local formula only specifies properties of a bounded number of actions of each component. Infinite traces can be verified by using periodic global formulas and shifted local formulas (as defined in section 2.2).

Theorem 3 (Recurring Local Formulas) Given an eligible circuit $\left(V, R, C, E, Q_{0}\right)$, let $G$ be a global formula and $L$ be a local formula such that $Q_{0} \Rightarrow G$ and $(R, G) \models L$. Let $p$ be a mapping from $\operatorname{comp}(L)$ to the natural numbers.

Then,

$$
\operatorname{periodic}(G, \operatorname{comp}(L), p) \Rightarrow \forall i \geq 0 .(R, G) \models \operatorname{shift}(L, p, i)
$$

Proof: Let $t$ be a trace of $\left(R, Q_{0}\right)$. The proof proceeds by induction on $i$, with induction hypothesis

$$
(R, G) \models \operatorname{shift}(L, p, i)
$$

$\wedge \forall t \in \operatorname{traces}(R, G) . \exists u \in \operatorname{traces}(R, G) . \exists m \in N$.
$(\quad u \cong t$
$\wedge \quad \forall c \in \operatorname{comp}(L) . \operatorname{occur}(c, u, m)=(i+1) * p(c)$
$\wedge \forall c \in(C \cup E)-\operatorname{comp}(L) . \neg \operatorname{enabled}\left(c, u^{m}\right)$
$\left.\wedge G\left(u^{m}\right)\right)$
The base case is $i=0$. From $\operatorname{shift}(L, p, 0)=L$ and $(R, G) \vDash L$ we see that $(R, G) \models \operatorname{shift}(L, p, 0)$. The second half of the conjunction follows from the definition of periodic global formulas. The induction step relies on properties of the subtrace $u^{\prime}=u^{m}, u^{m+1}, \ldots$ where $m$ is provided by the induction hypothesis for $i-1$. Since $u^{\prime}$ is in $\operatorname{traces}(R, G)$ we can see that $L$ holds for $u^{\prime}$, and since $\operatorname{occur}(c, u, m)=i * p(c)$, this is equivalent to $(R, G) \models \operatorname{shift}(L, p, i)$. The second half of the conjunction follows from the definition of periodic global formulas as applied to $u^{\prime}$. This completes the proof.

When using symbolic trajectory evaluation for verification, it is convenient to perform the actions of all enabled components as a single state transition. The next theorem shows that the validity and value of local formulas is preserved under
such an execution model. Let $R$ be a state transition relation, and let

$$
\left.\begin{array}{rl}
R^{+}= & \left\{\left(s^{1}, s^{2}\right) \in S \times S \mid\right. \\
& \forall c \in C \cup E . \\
& \exists s^{\prime} \in S \cdot\left(\left(s^{1}, s^{\prime}\right) \in R_{c}\right) \wedge\left(\forall v \in O_{c} \cdot v\left(s^{2}\right)=v\left(s^{\prime}\right)\right)  \tag{2.11}\\
& \vee \neg \operatorname{enabled}\left(c, s^{1}\right) \wedge\left(\forall v \in O_{c} \cdot v\left(s^{2}\right)=v\left(s^{1}\right)\right)
\end{array}\right\}
$$

Transition ( $s^{1}, s^{2}$ ) is in $R^{+}$iff $s^{2}$ is the state reached by performing an action of each component that is enabled in state $s^{1}$ in a single state transition. Note that the definition of occur does not require that components act one at a time.

Theorem 4 (Combined Actions) Given an eligible circuit ( $V, R, C, E, Q_{0}$ ), for all local formulas $L$,

$$
\left(R, Q_{0}\right) \models L \quad \Leftrightarrow \quad\left(R^{+}, Q_{0}\right) \models L
$$

Proof (sketch, see appendix A for details): Let $u$ be a trace of $\left(V, R^{+}, C, E, Q_{0}\right)$. We construct a trace $t$ of $\left(V, R, C, E, Q_{0}\right)$ such that $t \cong u$. The key idea is that if $u$ includes a transition $\left(t^{i}, t^{i+1}\right)$ where $n$ components perform actions, then we can effect the same change in $t$ with $n$ separate transitions. Because the original circuit is speed-independent, each component that was enabled in state $u^{i}$ is enabled when its turn arrives. Furthermore, because any component that occurs in a term of $L$ must have stable input, all input nodes of the component have the same value when the component performs its action in $t$ as in $u$. This guarantees that $L$ has the same value and validity in $t$ as in $u$.

To see the converse, consider a weakly-fair trace $t$ in $\left(V, R, C, E, Q_{0}\right)$. The proof proceeds by creating two traces, $u$ in $\operatorname{traces}\left(R^{+}, Q_{0}\right)$ and $\alpha$ in $\operatorname{traces}\left(R, Q_{0}\right)$, such that $\alpha$ and $u$ have the same relationship as the one described in the preceding
paragraph for $t$ and $u$, and $t \cong \alpha$. The traces are formed by considering the set $D^{1}$ of components which are enabled in $t^{0}$. The trace $u$ is formed by the simultaneous action of all components in $D^{1}$, while $\alpha$ is the trace formed by having them all as the first $\left|D^{1}\right|$ actions in $\alpha$. The process continues with the set $D^{2}$ of components enabled in $u^{1}$.

### 2.5 Discussion of Hypotheses

While the requirements of the theorems in section 2.4 may seem technical and restrictive, many of them have natural intuitions and are satisfied by common design methods. One of the basic requirements is that the circuit to be verified be eligible. This means that the circuit must be speed-independent, that none of its components have output choice, and that all of its environment components have stable inputs. Speed-independence has been chosen as a requirement because it is a readily testable guarantee of freedom from timing hazards. One key point for the next two properties is that for the proof of theorem 1 to work every component must either have stable inputs or no output choice. The formalism also only permits the verification of local formulas which describe the inputs and outputs of components with stable inputs. This makes the choice to restrict components with output choice to the environment reasonable, as it allows us to represent arbitrary data values as inputs to the circuit and guarantees that we will be able to make assertions about any node which connects the circuit to its environment. On the other hand, it does prevent us from handling circuits which contain arbiters.

Now consider theorem 2. In many ways it provides the foundation of this work. Its main requirement is that the two traces $t$ and $u$ be choice equivalent.

Although the definition of of choice equivalent traces is somewhat complicated, the meaning is simply that the environment provides the same stimulus to the circuit in both traces. Theorem 3 requires that the global formula $G$ be periodic, which means that we could choose an order for components to act such that we arrive back at $G$. Thus, the theorem's conclusion is that we can build arbitrarily long representative traces by returning repeatedly to states which satisfy $G$. Finally, theorem 4 requires that we have a transition relation $R^{+}$which contains transitions formed by simultaneously performing all enabled transitions in $R$.

## Chapter 3

## Local and Global Formulas

The combined use of global and local formulas is essential to the power of this verification method. They have different expressive abilities, and understanding their function and use is required before applying this theory to practical problems. In this chapter, the two different asynchronous implementations of a 4-bit adder as shown in figure 3.1 are used to illustrate a discussion of the two types of formulas. The local formulas describing correct function are discussed in section 3.1 and the global formulas and period functions describing the adders' periodic behaviour are discussed in section 3.2. Finally, section 3.3 describes a form of behaviour which cannot be verified using this technique.

Both designs use the same dual-rail codes as the FIFO described in section 2.1 and are speed-independent. The design on the left calculates the carry for each input pair and carry-in only after all three inputs are valid. This satisfies the stable inputs condition. In order to create a pipeline and allow a possible throughput of one operand per component delay, the adding components are embedded along the diagonal in a block of buffers (FIFO cells). In the second (flat) adder, the carry unit


Figure 3.1: 4 Bit Adders
can produce a result as it receives two inputs of the same value, creating a datadependent throughput. This is still speed-independent, even though the carry unit does not have stable inputs. This realizes a logarithmic average-case latency [Sei79]. Both adders are verified using the same environment, shown in figure 3.2.

### 3.1 Local Formulas

Local formulas are a form of modal logic specifically designed to describe invariant properties of speed-independent asynchronous circuits. Given the same initial state and inputs from the environment, a speed-independent circuit will produce the same sequence of values on the nodes attached to components with stable inputs. However, because the circuit is asynchronous, it is not possible to know the timing of these values. Local formulas are assertions about properties which are independent of the relative timing of components. If a local formula only refers to the inputs and outputs of a circuit, then it is also independent of the internal circuit structure. This allows the same specification to be used with two different implementations, such as the two adders described above.


Figure 3.2: Adder Environment

Recall from definition 10 that local formulas are boolean expressions formed from terms of the form $v_{c}^{k}$, where $v$ is an input or output node of a component $c$ with stable inputs. The index $k$ denotes that we are describing the state of the node $v$ at the $k^{\text {th }}$ action of $c$. Since environment nodes are required to have the property of stable inputs, and they are the place where the actions of the system are apparent to the outside world, they are natural candidates for acting as the components $c$ in local formulas. Also recall that $v_{c}^{k}$, refers to the value of node $v$ at the instant before the $k^{\text {th }}$ action by $c$. Accordingly, the output on node $v$ for $c$ 's first action is expressed as $v_{c}^{2}$. Finally, local formulas are typically used to describe a single step of a system's function, which in this case is a single set of operands.

With this in hand, we can describe the desired functionality of the 4-bit adder. For the sake of conciseness, $\left[\mathrm{A3}_{s r c a 3}^{2}, \mathrm{~A} 2_{s r c a 2}^{2}, A 1_{s r c a 1}^{2}, \mathrm{AO} 0_{s r c a 0}^{2}\right]$ is used to repre-
sent the bit vector supplied by the source components for the $a$ operand with their first action, and binary addition is used as shorthand for the logical notation using $\wedge$ and $\neg$.

$$
\begin{align*}
& {\left[\mathrm{S}_{3 n k s 4}^{1}, \mathrm{S3}_{s n k s 9}^{1}, \mathrm{~S}_{3 n k s 2}^{1}, \mathrm{S1}_{s n k s 1}^{1}, \mathrm{SO}_{s n k s 0}^{1}\right]} \\
& =\left[0, \mathrm{A3}_{s r c a 3}^{2}, \mathrm{A2}_{\text {srca2 }}^{2}, \mathrm{~A}_{s r c a 1}^{2}, \mathrm{AO}_{s r c a}^{2}\right]+\left[0, \mathrm{B3}_{\text {srcb3 }}^{2}, \mathrm{B2}_{\text {srcb2 }}^{2}, \mathrm{B1}_{s r c b 1}^{2}, \mathrm{BO}_{s r c b 0}^{2}\right] \\
& +\left[0,0,0,0, \text { Cin }_{s r c c i n}^{2}\right] \tag{3.1}
\end{align*}
$$

Since it refers only to nodes through their association with environment components, this specification does not reflect the internal structure of the adder. This allows the same specification to be used for any implementation of the adder.

### 3.2 Global Formulas and Period Functions

Global formulas serve as an "invariant", to which the system will return after one of the cycles described by the period function $p$. These formulas are not genuine invariants, however, for a number of reasons. First, they are not expected to hold throughout the device's computation. Second, the asynchronous implementation is not required to return to a state where this formula holds. However we can think of the system as returning to the state $G$ because if there is one trace for which the global formula recurs with period $p$, then by the equivalence of traces, conclusions drawn for this trace can be applied to all traces with the same inputs from the environment.

Before further discussion of the conceptual basis and use of global formulas and periodic functions, let us recall their definitions. A global formula $G$ is a predicate over the instantaneous state of the system, and the period function $p$ is a function from a set $C^{\prime} \subset C$ to the counting numbers such that only components in
$C^{\prime}$ are enabled in a state satisfying $G$ and for every trace whose initial state satisfies $G$ there is a choice consistent trace that returns to a state satisfying $G$ after each component $c$ in $C^{\prime}$ has performed $p(c)$ actions. Also recall that the set $C^{\prime}$ of components whose behaviour is described by $p$ must contain all of (and typically only) the components which appear in the local formulas. Thus, the period function $p$, like the local formula, should not change with different implementations of the device.

Most devices can be thought of as having clear cycles of operation. For the FIFO this is the insertion and removal of a single piece of data; for the adder it is the production of one sum from two operands. If we consider an environment which supplies one set of data and then waits until the outputs have been produced, we can see that the system returns to a particular state. This is the state which the global formulas describe. Note that this state may never reappear after the initial situation because the real environment may supply data before the outputs have been extracted. However, we know that we could have waited for the outputs to appear, which means that each set of data can be considered as if it were the first. As established by theorem 2 , having a representative trace which exhibits correct behaviour ensures that all traces behave correctly. We consider the trace which returns to the state described by the global formula after each computation, and it serves as our representative trace.

There is a high level of interdependence between different nodes in asynchronous circuits, due to the handshaking protocols. As a result, any unspecified values in the initial state will quickly propagate through the entire system. Thus, the global formulas typically describe the state of every node, and are dependent on the implementation details. If we consider the two adders, both of their global formulas describe them in an empty state. However, since the pipelined adder sys-


Figure 3.3: Pipelined Asynchronous Adder
tem contains more components, its global formula will specify that they are empty, in addition to the components which it shares with the flat adder. In this case the global formulas are similar, but in general there is no requirement that they match on anything other than specifying the initial state of the nodes connected to the environment.

Figure 3.3 shows the internal structure of the pipelined adder. The flat adder has the same structure with all of the FIFO elements removed. For simplicity's sake, consider a generic predicate empty which takes lists of components as its argument and is true if any components with a data output have their output set to the dual rail code for empty, any components with acknowledge lines have this signal set to show an empty state, and any internal state is set in a manner which does not enable any transitions. The global formulas for the two adders can then be expressed by figures 3.4 and 3.5 below.

```
empty [srcci, srca0, srcb0, srca1, srcb1, srca2, srcb2, srca3, srcb3,
    fif001a, fif001b, fif002a, fif002b, fif003a, fifo03b, fifo10s,
    fifo12a, fifo12b, fifo13a, fifo13b, fifo20s, fifo21s, fifo23a,
    fifo23b, fifo30s, fifo31s, fifo32s, maj0, maj1, maj2, majl3,
    vjoinv0, vjoinv1, vjoinv2, vjoinv3, xor0, xor1, xor2, xorl3,
    snk0, snk1, snk2, snk3, snk4]
```

Figure 3.4: Global Formula for Pipelined Adder

```
empty [srcci, srca0, srcb0, srca1, srcb1, srca2, srcb2, srca3, srcb3,
    maj0, maj1, maj2, majl3, vjoinv0, vjoinv1, vjoinv2, vjoinv3,
    xor0, xor1, xor2, xorl3, snk0, snk1, snk2, snk3, snk4]
```

Figure 3.5: Global Formula for Flat Adder

The period function provides a connection between the global formula and
the local formula. Although it refers to the actions required to return to the global formula, which is implementation dependent, like the local formula it is implementation independent. Since it only needs to describe the periods of components described in the local formula, typically it will describe the number of actions of the environment components at the inputs and outputs of the circuit. This can be seen from the adder examples. The cycle of operation for the adders is receiving two bit arrays of values followed by empty values to separate them from any subsequent data, and producing another array representing the sum (also followed by empty values. Thus, for both adders the period function is 2 for all of the input and output components.

### 3.3 Unspecifiable Behaviour

This method allows us to specify and verify behaviours which can be described by formulas whose terms are the values of nodes immediately before the $k^{\text {th }}$ action of components. We can neither describe nor verify properties which correspond to the relative timing of the actions of distinct components. For example, structural properties of a circuit may maintain invariants (such as the fact that the $n+1^{s t}$ input to a $n$ stage FIFO may not be accepted until there has been at least one output action) which can neither be described nor verified by this method. However, the goal was to verify data-path circuits. For such circuits, assertions about the sequences of values are a natural method for specifying behaviour.

## Chapter 4

## Implementation

This section describes how the results from chapter 2 can be applied using existing verification tools to verify self-timed designs. The overall scheme for the verification is:

- The structure and behaviour of the circuit and its environment are described using VHDL.
- The specification is written as an initial state predicate, a local formula, a global formula, and a period function. These formulas are verified for the circuit using symbolic trajectory evaluation.
- Speed-independence and stable-input assumptions are verified by model checking small clusters of components separately.


### 4.1 VHDL Circuit Descriptions

[^0]interconnection of these components. The VHDL model provides $R^{+}$as described in section 2.4.

Augmenting the models for environment components facilitates verification. In particular, added registers provide data corresponding to the environment's output choice and record the outputs of the circuit. This matches the specifications which describe the input/output behaviour of the circuit. Since trajectory evaluation only allows assertions about data values at particular times, we add a counter which increments each time a data value is received and use it to direct data values into an array. This bridges the gap between asynchronous behaviour and synchronous specifications. Thus the occurrence number of an action will correspond to its position in the array of recorded values. Counters can also be added to record the number of actions taken by the components, for the purposes of establishing that the period function $p$ is correct and to disable the environment components after the period has completed. Since only components which are in the domain $C$ of $p$ may be enabled in states satisfying the global formula $G$, a correctly functioning circuit will eventually halt in a state satisfying $G$ if all of the components in $C$ are disabled after the number of actions permitted by $p$.

These registers and counters are auxiliary circuitry in the environment components. In order to fully verify the circuit, this auxiliary circuitry must not inappropriately restrain the environment components. It is important that the choice variables stored in the source components' arrays do not reduce the output choice of these components. However, these variables are symbolic and simply serve as reference names for the values for the purposes of verifying a general formula with only one simulation run. It is also important that the restriction of the components to a fixed number of actions by the counters does not prevent them from correctly
representing general source components. If we were not taking advantage of the periodic behaviour of the circuit, we would require an infinite number of registers initialised to arbitrary boolean variables in order to emulate the output choice of the environment. However, because we know that verifying a finite number of steps is sufficient to show general correctness, only a small number of registers and boolean variables are required.

Three examples of VHDL code excerpts are provided: figure 4.1 shows code for a source which supplies a a data value and then stops; figure. 4.2 is a sink unit which records the last data value to arrive; and figure 4.3 shows the code for a single FIFO cell. This code is for an FIFO which passes data values in inverted form. Note that some of the internal variables (such as index, and done) must be set by the STE antecedent.

### 4.2 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation (STE) is a verification technique for circuits with deterministic next state functions which verifies specifications written in a restricted logic called "trajectory formulas". Although restrictive, trajectory formulas readily express requirements of the form "the value of a node $n$ at time $t$ is the function $f$ of the values of the nodes $\left[n_{1}, \ldots, n_{k}\right]$ at times $\left[t_{1}, \ldots, t_{k}\right]$ ". Using choice variables to control the output choice of the source components and counter indexed arrays to store inputs to the sink components allows us to represent local formulas using this structure.

The specification that a global formula $G$ has period $p$ can also be expressed in this form. After augmenting the models for the environment components with "period counters" as described in section 4.1, we verify that all trajectories that

```
ENTITY source IS
PORT (
    sr : IN BIT; -- request signal from first FIFO cell
    outT : OUT BIT; -- dual rail TRUE
    outF : OUT BIT; -- dual rail FALSE
    data : IN BIT -- auxiliary value to fix output choice
    );
END source;
ARCHITECTURE data_flow OF source IS
    SIGNAL cTOut : REG_BIT REGISTER;
    SIGNAL cFout : REG_BIT REGISTER;
    SIGNAL dindex : REG_BIT REGISTER; -- One data passing action
    SIGNAL eindex : REG_BIT REGISTER; -- One empty passing action
BEGIN
-- Transition to supply empties when next cell has data
clr : BLOCK ( NOT eindex AND NOT sr AND (cTout OR cFout))
BEGIN
    cTout <= GUARDED "0";
    cFout <= GUARDED "O";
    eindex <= GUARDED "1";
END BLOCK clr;
-- Stuff for dealing with supplying data and toggling index
dref0 : BLOCK ( (NOT (cTout OR cFout )) AND sr AND NOT dindex )
BEGIN
    cTout <= GUARDED data;
    cFout <= GUARDED NOT data;
    dindex <= GUARDED "1";
END BLOCK;
-_--------------------------
outT <= cTout;
outF <= cFout;
END data_flow;
```

Figure 4.1: VHDL Code for Source Component

```
ENTITY sink IS
PORT (
    r : OUT BIT; -- request signal to last FIFO cell
    inT : IN BIT; -- dual rail input for inverted TRUE
    inF : IN BIT -- dual rail input for inverted FALSE
    );
END sink;
ARCHITECTURE data_flow OF sink IS
    SIGNAL regT : REG_BIT REGISTER;
    SIGNAL regF : REG_BIT REGISTER;
    SIGNAL gr : REG_BIT REGISTER;
    SIGNAL done : REG_BIT REGISTER;
BEGIN
-- Transition to acknowledge an empty
clr : BLOCK ( NOT ( inT OR inF ) )
BEGIN
    gr <= GUARDED "0";
END BLOCK clr;
-- Stuff for dealing with valid data
val : BLOCK ( ( inT OR inF ) AND NOT done)
BEGIN
    regT <= GUARDED inT;
    regF <= GUARDED inF;
    gr <= GUARDED "1";
    done <= GUARDED "1";
END BLOCK;
r <= gr;
END data_flow;
```

Figure 4.2: VHDL Code for Sink Component

```
ENTITY fifocell IS
PORT (
    inT : IN BIT; -- dual rail TRUE input
    inF : IN BIT; -- dual rail FALSE input
    outT : INOUT BIT; -- dual rail TRUE output
    outF : INOUT BIT; -- dual rail FALSE output
    r : OUT BIT; -- request signal from previous FIFO cell
    sr : IN BIT -- request signal from next FIFO cell
    );
END fifocell;
ARCHITECTURE data_flow OF fifocell IS
    SIGNAL gTout : REG_BIT REGISTER;
    SIGNAL gFout : REG_BIT REGISTER;
BEGIN
ct : BLOCK (( inT = sr ) AND ( gTout XOR inT ))
BEGIN
    gTout <= GUARDED inT;
END BLOCK;
cf : BLOCK (( inF = sr ) AND ( gFout XOR inF ))
BEGIN
    gFout <= GUARDED inF;
END BLOCK;
-- NOR gate
r <= NOT ( gTout OR gFout );
-- required because guarded variables can't be outputs
outT <= gTout;
outF <= gFout;
END data_flow;
```

Figure 4.3: VHDL Code for FIFO Cell
start in a state that satisfies $G$ end in a state which also satisfies $G$ and that for each environment variable $e$, the corresponding counter ends with value $p(e)$. The form of trajectory formulas makes this very easy when most components return to an empty state at the end of the period. More care (and therefore more time) is required if data values can remain in the circuit at the end of the period.

### 4.3 Model Checking for Speed-Independence

Verification using local formulas is based upon the assumption that the circuit is eligible. This means that speed-independence must be verified for the circuit as well as the the stable-inputs property required for components appearing in local formulas. In this approach, the designer identifies small sub-circuits (typically single components) whose interfaces observe common self-timed handshaking protocols, and where the speed-independence of the sub-circuit depends only on its environment observing the handshaking protocol, not upon the particular values applied. Therefore, each sub-circuit is model checked with generic sources and sinks attached to the interfaces. These generic environment components have non-deterministic nextstate relations that ensure that they observe the handshaking protocol, but allow them to apply arbitrary data values. In this context two properties are verified using a model checker [LGS94a]. First, the sub-circuit is shown to be speed-independent (and to have stable inputs, if required) in this generic environment. Second, the sub-circuit is shown to observe the handshaking protocol as well, by showing that its behaviour on the lines to each neighbour is a refinement of the behaviour between a sink and a source (the simplest circuit which implements the correct handshaking protocol). That this demonstrates the speed-independence of the complete circuit can be seen either from the discussion in [AL90] on composing specifications, or
from the protocol verification described in [SM95]. Because the sub-circuits are small, model checking is fast and completely automatic. The property of stable inputs can also be expressed as a predicate and verified using model checking.

As an example, consider the FIFO shown in figure 2.1. The design is partitioned into six sub-circuits: the source, the sink, and the four FIFO stages where stage 1 consists of components $c 1 t, c 1 f$, and $n 1$, and likewise for stages 2,3 , and 4. The descriptions of the $s r c$ and $s n k$ components in equation 2.5 provide generic sources and sinks for the four-phase handshake protocol of the FIFO. The FIFO stage to be verified is composed with a generic source connected to its data input and a generic sink connected to its output. Model checking shows that this composition is a speed-independent circuit and that at its input interface the FIFO stage only performs actions that could be made by a generic sink and that at its output interface the stage only performs actions that could be made by a generic source.

## Chapter 5

## Vector Multiplier

For an example with a significant amount of data path complexity, we chose the vector multiplier described in [SS93]. The multiplier uses an iterative serial-parallel algorithm implementing the "paper and pencil" method for multiplication, with the accumulating sum stored in carry-save form until it is to be output. The key functional unit is the MAS (Multiply-Accumulate-Shift) block. With each bit of the serial operand, the parallel operand and the carry word are added to the accumulating sum (the parallel operand is first ANDed with the serial bit), and then the parallel operand and the carry word are shifted to the left, while the sum bits circulate in the MAS bit-slices. When the result is to be produced, the sum is converted from carry-save form by extending the serial operand with enough zeroes to ensure that the carry word is completely cleared. A detailed diagram of the MAS bit-slice is shown in figure 5.1. Unlabelled devices in the diagram represent asynchronous latches (fifo cells) or sinks. There is a source (labelled with a 0 ) which produces a stream of dual-rail false values. The switches are asymmetric: when the control line is a dual-rail false, data is passed directly through; when the control line is true, the
right input is passed to the left output and no operation is performed on the other input and output. This circuit was modelled at the level of functional blocks. That is, behavioural VHDL was used to describe the net behaviour of each component, rather than building them up from gates or transistors.

The two control signals organise the different time scales in the multiplier, which functions in nested cycles at three levels. The lowest level is the cycle described above which occurs once per serial bit. At each iteration of this cycle, both C1 and C2 receive a single T value, followed by an empty. The middle level consists of actions which take place once per complete serial operand, such as loading a new parallel operand into the MAS block, and here $\mathbf{C 1}$ receives an F , while $\mathbf{C} 2$ still receives a T. The highest cycle hinges on an operand Last, which is supplied with each operand pair and tells the multiplier whether or not to convert the accumulated sum from carry save into binary representation and produce it as a result. This results in extending the serial operand with a number of zeroes and then producing $F$ 's on both C1 and C2.

These multiple cycles complicate the formalism developed in the preceding chapters. Theorem 3 requires a period function $p$ in order to show that we can verify unbounded traces. However, in this example there is no clear $p$. If we consider the mid-level cycle which occurs once per operand pair, we see that an an output is only produced by the vector multiplier when the Last input is true. This means. that the the number of actions taken by the output components is data dependent. To avoid this problem we use the outermost cycle (Last true to Last true) with a fixed number of operand pairs to define the period. However, this also fails to produce a well defined function $p$, as we do not know the number of operand pairs which will be supplied to the multiplier between Last's. Thus, we can show that


Figure 5.1: Bit Slice of the MAS Block (from [SS93])
for a particular $n$, the multiplier will function if $n$ input pairs are provided before the output is produced. Since the initial and final states satisfy the same $G$, we can also assert that the system works for sequences of a particular $n$. Table 5.1 below shows verification times for small $n$ 's. The rapid increase verification times with the number of pairs is a result of the OBDD representations of the accumulated sum. A OBDD's size increases considerably when its value depends on the interaction of many variables, and the most significant bits of the accumulated sum depend in a complicated fashion on all of the bits of each operand pair.

| Device | STE Time (in s) |
| :--- | ---: |
| Fifo (7 stages) | 3 |
| Adder (4 bit) | 4 |
| Vector Multiplier (1 pair) | 14 |
| Vector Multiplier (2 pairs) | 22 |
| Vector Multiplier (3 pairs) | 602 |
| Vector Multiplier (4 pairs) | stopped after 340 min |

Table 5.1: STE Verification Times

Initially, we had hoped to use the middle level (once per serial operand) cycles to define $p$, and then to describe the possible behaviours of holding the sum in carry save form or exporting it after conversion in a complicated local formula $L$. The difficulty arises due to the fact that the multiplier may or may not produce the result, which means that we need to include the component which outputs the results, but it may undergo 2 or 0 transitions depending on the value of the input

## Last.

This example had the most complicated periodic global formula $G$. Because of rings contained in the circuit, not all components return to an empty state. Correctly determining $G$ was the most time consuming aspect of verifying the vector
multiplier, although this included finding a suitable initial state for the system. Part of the difficulty was caused by the existence of periodic global formulas which do not correctly initialise the system (and result in incorrect function).

Part of the difficulty in describing the global formula comes from the restrictions for the trajectory formulas used in STE. Disjunction is not easily expressed in STE, and as a result the global formulas have to be described exactly as a function of the inputs and initial state. This requires a precise understanding of the circuit and its cycles. For example, for the vector multiplier the data values wait in different points of the circuit depending on the value of the Last variables associated with the last two operand pairs. Including disjunction would allow the global formula to be a general predicate which includes all of the possible final states, although it still would not handle the fact that the output behaviour depends on the input data.

## Chapter 6

## Conclusions

This thesis presents an approach to verifying speed-independent circuits that exploits the observation that the sequence of operations performed by each component of the circuit is independent of the order in which the operations are performed. The specifications have three components: a local formula, a period function, and a periodic global formula. The local formula and period function capture the desired input/output behaviour of the circuit and are usually simple and intuitive to write. The periodic global formula describes a set of states that are equivalent to the initial state with respect to the handshaking protocol. Typically, the global formula is not an "interesting" part of the specification, but it is needed for efficient verification. While it may be complicated and depend on user knowledge, the knowledge which is required is a detailed understanding of the circuit's function. So, while the system is not automatic, it requires information which the system designer is likely to have (as opposed to requiring a deep understanding of formal logic).

We have implemented the verification techniques described in this thesis using a combination of model checking and symbolic trajectory evaluation. Model
checking is used to show that the design is speed-independent. These tests are performed separately for small sub-circuits in the design to avoid the state-space explosion problem. Symbolic trajectory evaluation is used to verify the local formulas and the periodicity of the global formula. This are done using a deterministic circuit model that allows large designs to be handled efficiently. Although symbolic trajectory evaluation was used for the given examples, the framework of local and global formulas provides a general basis for applying verification methods designed for deterministic (i.e. synchronous) models to speed-independent circuits. Using the Voss Prover (STE combined with a specialised theorem prover) would significantly increase the size of circuit which can be verified by this method.

As examples of this technique; several speed-independent data-path designs have been verified, including a simple FIFO, two designs for an adder, and a vector multiplier chip.

### 6.1 Future Work

The inability of this theoretical framework to completely verify the vector multiplier (that is, verifying correct behaviour with an arbitrary number of input pairs before producing an output) points out where shortcomings could be addressed. We believe that the theory can be extended to allow multiple global formulas $\left\{G_{1}, G_{2}, \ldots, G_{n}\right\}$ with local formulas describing the input and output sequences which move the system from one to the other. This creates an abstract finite state machine where the arcs are verified local formulas describing transitions from $G_{i}$ to $G_{j}$. With this added structure the vector multiplier can be thought of as having 2 recurring states: the empty state after producing an output; and the state where it has a partial sum stored and is waiting for a new input. Figure 6.1 represents the corresponding
abstract state machine.


Figure 6.1: Vector Multiplier: Abstract State Machine

A simple theorem prover could be used to manipulate results and record proof obligations. If the check for speed-independence and trajectory evaluation were included as tools for showing base-level facts, results could be combined using specialised rules in a manner analogous to the Voss Prover [HS95]. A further extension would be a formal framework in which subsystems of the circuit could be verified independently. For the vector multiplier this would allow us to verify the multiplication in the MAS block one bit at a time (as in [AS95]), permitting the verification of systems with much larger data-paths.

This research could also be extended into other models for asynchronous circuits. Delay-insensitive circuits, which allow both components and wires to have arbitrary delays, are a subcategory of speed-independent circuits, and can already be verified using this method. More general classes of asynchronous circuits would require more structure, but any hazard-free circuit should be possible to verify using a synchronous model. For example, hazard-free timed circuits are still deterministic. One possibility would be to show that they are a refinement of a speed-
independent specification using model checking [LGS94b], or the separation of events in time $\left[\mathrm{HB}^{+} 94\right]$, before verifying synchronously.

Finally, some circuits are not speed-independent at their lowest level, but have a deterministic model to which they should correspond. An example is the Counter-Flow Pipeline Processor [SSM94], which contains arbiters and is therefore not speed-independent. There are two approaches which may allow us to work with these circuits. Arbiters could be modelled by speed-independent components with a choice variable supplied by the environment that determines the direction of the grant in the event of two requests. Correct behaviour can then be shown regardless of the choice variables supplied, which shows correct function given any permitted behaviour by the arbiter. The other approach would be to observe invariants in system behaviour which allow a formal equivalence to be shown between the circuit and a deterministic model which could be verified using STE. The Counter-Flow Pipeline Processor might be accessible using this method. Invariants could be shown based on the pipeline rules [SSM94] which show that each instruction executes in an environment where it appears that all previous instructions have executes. However, this approach would require sophisticated users to develop specialised invariants for each circuit.

## Bibliography

[AL90] Martín Abadi and Leslie Lamport. Composing specifications. Technical Report 66, Digital Equipment Corporation, Systems Research Center, October 1990.
[AS95] Mark D. Aagaard and Carl-Johan Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In Proceedings of the 1995 International Conference on Computer Aided Design, November 1995 .
[ $\left.\mathrm{BC}^{+} 94\right]$ J.R. Burch, E.M. Clarke, et al. Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design, 13(4):401-424, April 1994.
[Bea93] Derek L Beatty. A methodology for formal hardware verification, with application to microprocessors. PhD thesis, Computer Science Deparatment, Carnegie Mellon University, 1993.
[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, C-35(8):677-691, August 1986.
[BS89] E. Brunvand and R.F. Sproull. Translating concurrent programs into delay-insensitive circuits. In Proceedings of ICCD, pages 262-265, 1989.
[CLW85] T.A. Chu, C.K.C. Leung, and T.S. Wanuga. A design methodology for concurrent VLSI systems. In Proceedings of ICCD, pages 407-410, 1985.
[Dar94] Mohammad Darwish. Formal verification of a 32-bit pipelined RISC processor. Master's thesis, Department of Electrical Engineering, University of British Columbia, 1994.
[Ebe91] J.C. Ebergen. A formal approach to designing delay-insensitive circuits. Distributed Computing, 5(3):107-119, July 1991.
[Gup92] Aarti Gupta. Formal hardware verification methods: A survey. Formal Methods in System Design, 1(2/3):151-258, October 1992.
[Hau95] S. Hauck. Asynchronous design methodologies: An overview. Proceedings of the IEEE, 83(1), 1995.
[ $\mathrm{HB}^{+94]}$ Henrik Hulgaard, Steven M. Burns, et al. An algorithm for exact bounds on the time separation of events in concurrent systems. Technical Report 94-02-02, Department of Computer Science, University of Washington, Seattle, 1994.
[HS95] Scott Hazelhurst and Carl-Johan H. Seger. A simple theorem prover based on symbolic trajectory evaluation and BDDs. IEEE Transactions on Computer-Aided Design, 14(4):413-422, April 1995.
[KK $\left.{ }^{+} 92\right]$ M.A. Kishinevsky, A.Y. Kondratyev, et al. On self-timed behaviour verification. In Proceedings of TAU'92, March 1992.
[LGS94a] Trevor W.S. Lee, Mark R. Greenstreet, and Carl-Johan Seger. Automatic verification of asynchronous circuits. IEEE Design and Test, 12(1):24-31, Spring 1994.
[LGS94b] Trevor W.S. Lee, Mark R. Greenstreet, and Carl-Johan Seger. Automatic verification of refinement. In Proceedings of the 1994 International Conference on Computer Design, Boston, October 1994.
[Mar89a] Alain J. Martin. The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation. In M. Leeser and G. Brown, editors, Hardware Specification, Verification and Synthesis: Mathematical Aspect, LNCS 408, pages 244-259. Springer-Verlag, 1989.
[Mar89b] Alain J. Martin. Programming in VLSI: From communicating processes to delay insensitive circuits. In C.A.R. Hoare, editor, University of Texas Year of Programming Institute on Concurrent Programming. AddisonWesley, 1989.
[MB59] David E. Muller and W.S. Bartky. A theory of asynchronous circuits. In Proc. Int. Symp. Theory Switching, pages 204-243, 1959.
[MFR85] C.E. Molnar, T.P. Fang, and F.U. Rosenberger. Synthesis of delayinsensitive modules. In Henry Fuchs, editor, Proc. 1985 Chapel Hill Conference on VLSI. Computer Science Press, 1985.
[MG93] Thomas F. Melham and Michael J. C Gordon. Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, New York, 1993.
[Mur89] T. Murata. Petri nets: Properties, analysis, and applications. Proceedings of the IEEE, 77(4):541-580, 1989.
[ $\mathrm{OR}^{+} 95$ ] S. Owre, J. Rushby, et al. Formal verification for fault tolerant architectures: Prolegomena to the design of PVS. IEEE Transactions on Software Engineering, 21(2):107-125, February 1995.
[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In Proceedings of CAV:93, LNCS 697, pages 409-423, 1993.
[Seg92] Carl-Johan H. Seger. Introduction to formal hardware verification. Technical Report 92-13, UBC, November 1992.
[Sei79] Charles L. Seitz. System timing. In Introduction to VLSI Systems (Carver Mead and Lynn Conway), chapter 7, pages 218-262. Addison Wesley, 1979.
[SM95] Jørgen Staunstrup and Niels Mellergaard. Localized verification of modular designs. Formal Methods in System Design, 1995.
[SS93] Jens Sparsø and Jørgen Staunstrup. Delay-insensitive multi-ring structures. INTEGRATION, 15(3):313-340, October 1993.
[SSM94] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The counterflow pipeline processor architecture. IEEE Design and Test, 11(3):4859, Fall 1994.
[Sut89] Ivan E. Sutherland. Micropipelines. Communications of the ACM, 32(6):720-738, June 1989. Turing Award lecture.
[Udd84] Jan T. Udding. Classification and Composition of Delay-Insensitive Circuits. PhD thesis, Eindhoven University of Technology, 1984.
[vB92] K. van Berkel. Handshake circuits: an intermediary between communicating processes and VLSI. Technische Universiteit Eindhoven, 1992.
[Wil91] Ted E. Williams. Self-Timed Rings and their Application to Division. PhD thesis, Stanford University, May 1991.

## Appendix A

## Proofs

The actions of simultaneously enabled components commute without changing the values of local formulas. This fact is the keystone of many of the theorems in this paper, and is formalised by the following lemma.

Lemma 1 (Commuting Actions) Given an eligible circuit ( $V, R, C, E, Q_{0}$ ), a weakly fair trace $\alpha$, and a natural number $i$ such that $\alpha^{i+2}$ exists, let $c$ and $d$ be the components such that $\left(\alpha^{i}, \alpha^{i+1}\right) \in R_{c},\left(\alpha^{i+1}, \alpha^{i+2}\right) \in R_{d}$, and $d$ is enabled in state $\alpha^{i}$. If $\beta$ is defined to be the sequence of states such that

$$
\begin{aligned}
\beta^{k} & =\alpha^{k} & & \text { if } k \neq i+1 \\
v\left(\beta^{i+1}\right) & =v\left(\alpha^{i+2}\right) & & \text { if } v \in O_{d} \\
v\left(\beta^{i+1}\right) & =v\left(\alpha^{i}\right) & & \text { if } v \notin O_{d}
\end{aligned}
$$

then $\beta$ is a weakly fair trace, and $\alpha \cong \beta$.

Proof: To provide structure for this proof it has been divided into sections. Here is an outline of the proof obligations and their structure.

1. First, we must show that $\beta$ is a trace. This requires two steps.

- We need to show that $\left(\beta^{i}, \beta^{i+1}\right) \in R_{d}$. There are two cases.
- $d$ has no output choice
- $d$ has output choice
- Next we show that $\left(\beta^{i+1}, \beta^{i+2}\right) \in R_{c}$.

2. We have to show that $\alpha \cong \beta$ by showing that $v_{e}^{k}(\alpha)=v_{e}^{k}(\beta)$ for any component $e$ with stable inputs. This is broken up into two cases.

- $e \in C \cup E-\{c, d\}$
- $e \notin C \cup E-\{c, d\}$

3. Finally, we show that $\beta$ is weakly fair

The proof will follow this outline.
The following is a review of some facts which will be used repeatedly and without further justification throughout this proof. First observe that if $d=c$ then $\alpha=\beta$, and therefore $\alpha \cong \beta$ and $\beta$ is a weakly fair trace. Thus, we will assume that $d \neq c$ for the rest of the proof. Since $\alpha$ is a trace, we know that reachable $\left(\alpha^{i}\right)$ is true. By the definition of $\beta$, for all $k \neq i+1$ we know that $\beta^{k}=\alpha^{k}$, and from now on we will write $\alpha^{i}$ and $\alpha^{i+2}$ for $\beta^{i}$ and $\beta^{i+2}$. Finally, since $d$ is enabled in state $\alpha^{i}$, we know that there exists at least one transition $\left(\alpha^{i}, s\right) \in R_{d}$. We will use $s$ to denote the final state of this transition.

## Showing that $\beta$ is a trace

Showing that $\left(\alpha^{i}, \beta^{i+1}\right) \in R_{d}$

## Case 1: $d$ has no output choice

By speed-independence and the fact that $\left(\alpha^{i}, \alpha^{i+1}\right) \in R_{c}$, we can see that there exists an element of $R_{d}$ which starts at $\alpha^{i+1}$ and produces the same outputs as the transition $\left(\alpha^{i}, s\right)$. That is,

$$
\exists s^{\prime} \in S .\left(\alpha^{i+1}, s^{\prime}\right) \in R_{d} \wedge\left(\forall v \in O_{d} \cdot v(s)=v\left(s^{\prime}\right)\right)
$$

The absence of output choice implies

$$
\forall\left(s, s^{1}\right),\left(s, s^{2}\right) \in R_{d} \cdot\left(s^{1}=s^{2}\right)
$$

In particular, $\alpha^{i+2}=s^{\prime}$, implying that

$$
\forall v \in O_{d} \cdot \dot{v}(s)=v\left(\alpha^{i+2}\right)
$$

which, by the definition of $\beta$ shows that $\beta^{i+1}=s$, and therefore $\left(\alpha^{i}, \beta^{i+1}\right) \in R_{d}$.

## Case2:d has output choice

Here we know that $d$ has stable inputs. Since $\left(\alpha^{i}, \alpha^{i+1}\right) \in R_{c}$, this yields

$$
\forall v \in I_{d} \cup O_{d} \cdot v\left(\alpha^{i}\right)=v\left(\alpha^{i+1}\right)
$$

Because $d$ 's inputs and outputs are the same in both $\alpha^{i}$ and $\alpha^{i+1}, d$ must be able to perform the same actions.

$$
\exists s^{\prime} \in S .\left(\left(\alpha^{i}, s^{\prime}\right) \in R_{d} \wedge \forall v \in O_{d} \cdot v\left(s^{\prime}\right)=v\left(\alpha^{i+2}\right)\right)
$$

Because $\left(\alpha^{i}, s^{\prime}\right) \in R_{d}$, we know that

$$
\forall v \notin O_{d} \cdot v\left(s^{\prime}\right)=v\left(\alpha^{i}\right)
$$

and can see that $s^{\prime}=\beta^{i+1}$, and therefore $\left(\beta^{i}, \beta^{i+1}\right) \in R_{d}$.

Showing that $\left(\beta^{i+1}, \alpha^{i+2}\right) \in R_{c}$
By the definition of speed-independence and the fact that $c$ was enabled in $\alpha^{i}$, we know that

$$
\exists s^{\prime} \in S .\left(\left(\beta^{i+1}, s^{\prime}\right) \in R_{c}\right) \wedge\left(\forall v \in O_{c} \cdot v\left(s^{\prime}\right)=v\left(\alpha^{i+2}\right)\right)
$$

Since only outputs of component can change in a transition in $R_{c}$, we can see that the $s^{\prime}$ described above is equal to $\alpha^{i+2}$ and therefore $\left(\beta^{i+1}, \alpha^{i+2}\right) \in R_{c}$.

## Showing that $\alpha \cong \beta$

We will show that $v_{e}^{k}(\alpha)=v_{c}^{k}(\beta)$ for any component $e$ with stable inputs.

Case 1: $e \notin\{c, d\}$
Note that for all numbers $k$ where $k \neq i+1$ we have $\alpha^{k}=\beta^{k}$, and that the index of $e$ is never $i+1$. This implies

$$
\forall e \in C \cup E-\{c, d\} . \forall v \in I_{e} \cup O_{e} . \forall k \in N . \operatorname{occur}(e, \alpha, k)=\operatorname{occur}(e, \beta, k)
$$

and therefore

$$
\begin{aligned}
\forall e \in & C \cup E-\{c, d\} . \forall k \in N \\
& \operatorname{valid}\left(v_{e}^{k}, \alpha\right)=\operatorname{valid}\left(v_{e}^{k}, \beta\right) \\
\wedge & \operatorname{valid}\left(v_{e}^{k}, \alpha\right) \Rightarrow\left(v_{e}^{k}(\alpha)=v_{e}^{k}(\beta)\right)
\end{aligned}
$$

which is the definition of $\alpha \cong \beta$.
Case 2: $e \in\{c, d\}$
Now suppose that $e=c$ and that $c$ has stable inputs. If $c$ does not have stable inputs, then $v_{c}^{k}$ cannot appear in a local formula and we are done.

Let

$$
n_{c}=\operatorname{occur}(c, \alpha, i)=\operatorname{occur}(c, \beta, i)
$$

Considering the definition of $\beta$, we can see that

$$
\begin{align*}
& \forall k \in N . k \neq n_{c}+1 \Rightarrow \operatorname{index}(c, \alpha, k)=\operatorname{index}(c, \beta, k) \\
\wedge \quad & \operatorname{index}\left(c, \alpha, n_{c}+1\right)=i+1  \tag{A.1}\\
\wedge & \quad \text { index }\left(c, \beta, n_{c}+1\right)=i+2
\end{align*}
$$

and therefore

$$
\forall k \in N . \forall v \in O_{c}
$$

$$
\left(k \neq n_{c}+1\right) \wedge \operatorname{valid}\left(v_{e}^{k}, \alpha\right) \Rightarrow\left(v\left(\alpha^{i n d e x}(c, \alpha, k)\right)=v\left(\beta^{i n d e x}(c, \beta, k)\right)\right)
$$

Now suppose that $k=n_{c}+1$. Recall from the discussion in section 2.2 that assertions about the outputs of a component after its $k^{t h}$ action are written as $v\left(t_{c}^{k+1}\right)$ (or equivalently $v\left(t_{c}^{k}\right.$.post $)$ ), while assertions about the inputs to a component at the time of its $k^{\text {th }}$ action must be written as $v\left(t_{c}^{k}\right)$. This means that

$$
\begin{aligned}
& \forall v \in O_{c} \cdot v_{c}^{k+1}(\alpha)=v\left(\alpha^{\text {index }\left(c, \alpha, n_{c}+2\right)-1}\right)=v\left(\alpha^{\text {index }\left(c, \alpha, n_{c}+1\right)}\right)=v\left(\alpha^{i+1}\right) \\
& \forall v \in O_{c} \cdot v_{c}^{k+1}(\beta)=v\left(\beta^{\text {index }\left(c, \beta, n_{c}+2\right)-1}\right)=v\left(\beta^{\text {index }\left(c, \beta, n_{c}+1\right)}\right)=v\left(\alpha^{i+2}\right)
\end{aligned}
$$

but by the definition of $\beta$ and the fact that $d \neq c$ we know that

$$
\forall v \in O_{c} \cdot\left(v\left(\alpha^{i+1}\right)=v\left(\alpha^{i+2}\right)\right)
$$

which shows that for $v \in O_{c}$ we have $v_{c}^{k}(\alpha)=v_{c}^{k}(\beta)$. The final case is $v \in I_{c}$, where we can see that

$$
\begin{aligned}
& \forall v \in I_{c} . v_{c}^{k}(\alpha)=v\left(\alpha^{\text {index }\left(c, \alpha, n_{c}+1\right)-1}\right)=v\left(\alpha^{i}\right) \\
& \forall v \in I_{c} . v_{c}^{k}(\beta)=v\left(\beta^{i n d e x\left(c, \beta, n_{c}+1\right)-1}\right)=v\left(\beta^{i+1}\right)
\end{aligned}
$$

Since we assumed that $c$ has stable inputs and we know that $c$ was enabled in state $\alpha^{i}$, we know that

$$
\forall v \in I_{c} \cdot v\left(\beta^{i}\right)=v\left(\beta^{i+1}\right)
$$

and that therefore for $v \in I_{c}$ we have $v_{c}^{k}(\alpha)=v_{c}^{k}(\beta)$. The argument for $e=d$ is similar, and can be created from the above argument by changing equation A. 1 to read

$$
\begin{align*}
& \forall k \in N . k \neq n_{d}+1 \Rightarrow \operatorname{index}(d, \alpha, k)=\operatorname{index}(d, \beta, k) \\
\wedge & \operatorname{index}\left(d, \alpha, n_{d}+1\right)=i+2  \tag{A.2}\\
\wedge & \operatorname{index}\left(d, \beta, n_{d}+1\right)=i+1
\end{align*}
$$

This shows that $\alpha \cong \beta$.

## Arguing that $\beta$ is weakly fair

The argument proceeds by contradiction. Consider a component e. Suppose that $\beta$ is not weakly fair, and that there is an $k$ such that $e$ is enabled in $\beta^{k}$ and that $e$ remains enabled for all $j \geq k$. Pick a number $n>k$ such that $n>i+2$. We know that $e$ is enabled in $\beta$ for all $j>n$. However, for all $j>i+2$ we know that $\alpha^{j}=\beta^{j}$, and so $e$ is enabled in $\alpha$ for all $j \geq n$ as well. This is a contradiction because we know that $\alpha$ is weakly fair.

This concludes the proof.

The next lemma uses repeated commutation operations to move an action an arbitrary number of steps.

Lemma 2 (Splicing Step) Given an eligible circuit ( $V, R, C, E, Q_{0}$ ), If $t$ and $u$ are weakly fair traces and $m$ is a natural number such that $t^{m}$ exists, splice( $t, u, m-$
$1) \cong u$, and splice $(t, u, m-1) \cong_{E} t$, then splice $(t, u, m) \cong \operatorname{splice}(t, u, m-1)$ and splice $(t, u, m)$ is a weakly fair trace.

Proof: By applying definition 9 (splicing) and the fact that $t$ and $u$ are weakly fair traces, we can see that splice $(t, u, m)$ is the sequence of states

$$
\begin{aligned}
\text { splice }(t, u, m)^{j} & =t^{j} & & \text { if } j<m \\
v\left(\text { splice }(t, u, m)^{j}\right) & =v\left(t^{k}\right) & & \text { if }(m \leq j<k) \wedge v \in O_{c} \\
v\left(\text { splice }(t, u, m)^{j}\right) & =v\left(u^{j}\right) & & \text { if }(m \leq j<k) \wedge v \notin O_{c} \\
\text { splice }(t, u, m)^{j} & =u^{j} & & \text { if } k \leq j
\end{aligned}
$$

where $c$ is the component such that $\left(t^{m-1}, t^{m}\right) \in R_{c}, n_{c}=\operatorname{occur}(c, u, m-1)$, and $k=\operatorname{index}\left(c, u, n_{c}+1\right)$.

First, we assert that $k$ exists. This is a consequence of the fact that $u$ is weakly fair and $c$ is enabled in state $u^{m-1}$. The proof proceeds by induction on $k$. The base case is $k=m$. Because the circuit is eligible, only components in $E$ have output choice, so if $c \notin E$ then

$$
\forall s \in S .\left(t^{m-1}, s\right) \in R_{c} \Rightarrow s=t^{m}
$$

which combined with $t^{m-1}=\dot{u}^{m-1}$ gives us $u^{m}=t^{m}$. If $c \in E$, we know that $\cong_{c} u$. By $t^{m-1}=u^{m-1}$ and the definition of choice consistency, we can see that

$$
\begin{aligned}
\forall v \in O_{c} \cdot v\left(t^{m}\right) & =v\left(u^{m}\right) \\
\wedge \quad \forall v \notin O_{c} \cdot v\left(t^{m}\right) & =v\left(t^{m-1}\right)=v\left(u^{m-1}\right)=v\left(u^{m}\right)
\end{aligned}
$$

which is enough to see that $u^{m}=t^{m}$.
For the induction step, we know that index $\left(c, u, n_{c}+1\right)=k$, and that if we can create a weakly fair trace $\alpha$ such that $\alpha \cong \operatorname{splice}(t, u, m)$ and $\operatorname{index}(c, \alpha, 1)=k-1$ then by the induction hypothesis we would have a proof of our claim. However,
by speed-independence we know that that component $c$ is enabled in all states $t^{m-1}, \ldots, t^{k-1}$, which allows us to apply lemma 1 and exchange the actions of $c$ and the previous component. This creates the required trace $\alpha$.

Theorem 1 (Splicing) Given an eligible circuit $\left(V, R, C, E, Q_{0}\right)$, let $t$ and $u$ be weakly fair traces with $m$ a natural number such that $t^{m}$ exists. If $t^{0}=u^{0}$ and $t \cong_{E} u$, then splice $(t, u, m)$ is a weakly fair trace and splice $(t, u, m) \cong u$.

Proof: The proof proceeds by induction. The base case is $m=0$ which is discharged by observing that $u=\operatorname{splice}(t, u, 0)$. For the induction step we have splice $(t, u, m-1) \cong_{E} t$, splice $(t, u, m-1)$ is a weakly fair trace, and $u \cong \operatorname{splice}(t, u, m-$ $1)$. Since $t$ and splice $(t, u, m-1)$ satisfy the conditions of lemma 2 we can see that $\operatorname{splice}(t, u, m)$ is a weakly fair trace and that $\operatorname{splice}(t, u, m) \cong \operatorname{splice}(t, u, m-1)$. Since $\operatorname{splice}(t, u, m-1) \cong_{E} t$ and splice $(t, u, m-1) \cong u$, this gives splice $(t, u, m) \cong_{E} t$ and $\operatorname{splice}(t, u, m) \cong u$.

When discussing traces from $\operatorname{traces}(R, G)$ and $\operatorname{traces}\left(R^{+}, \mathrm{G}\right)$ it is convenient to describe two traces $t \in \operatorname{traces}(R, G)$ and $u \in \operatorname{traces}\left(R^{+}, G\right)$ as corresponding if $t$ is the trace formed by concatenating the execution in some order the components active in all of the changes ( $u^{i}, u^{i+1}$ ), and where the choices made by elements in $t$ are the same as those made by elements of $u$. To formally define corresponding traces, we define notation for describing active components, i.e. those components which jointly perform a transition in $u \in \operatorname{traces}\left(R^{+}, G\right)$. Note that we use $|u|$ to denote the number of states in a trace $u$, and $|D|$ to denote the number of elements
of a set $D$.

Definition 15 (Active Components) Let ( $V, R, C, E, Q_{0}$ ) be an eligible circuit, $u$ be trace in traces $\left(R^{+}, G\right)$, and $t$ be a trace in $\operatorname{traces}(R, G)$. Let $D^{i}$ denote the set of components active in the transition from $u^{i-1}$ to $u^{i}$

$$
D^{i}=\left\{c \in C \mid \exists v \in O_{c} . v\left(u^{i-1}\right) \neq v\left(u^{i}\right)\right\}
$$

and $m_{i}$ denote the number of state changes required in to allow all of the components in all of the previous sets $D^{i}$ to act (with $D^{0}$ defined to be the empty set)

$$
m_{i}=\sum_{j=0}^{i-1}\left|D^{j}\right|
$$

Definition 16 (Corresponding Traces) Let $\left(V, R, C, E, Q_{0}\right)$ be an eligible circuit. Two traces $t \in \operatorname{traces}(R, G)$ and $u \in \operatorname{traces}\left(R^{+}, G\right)$ correspond if every component which performs an action in the trace $t$ performs the same action as part of a combined transition in the trace $u$, and these actions in $t$ are clustered in subsequences corresponding to each single action in $u$. That is,

$$
\forall i, j \in N . m_{i} \leq j<m_{i+1} \Rightarrow\left(\exists d \in D^{i+1} \cdot\left(t^{j}, t^{j+1}\right) \in R_{d}\right)
$$

where $D^{i}$ and $m_{i}$ are as described in definition 15 above, and the choices made by each environment component are the same

$$
\forall e \in E . \forall i \in N
$$

$$
\left(\forall v \in I_{e} . \operatorname{valid}\left(v_{e}^{i}, t\right) \wedge v_{e}^{i}(t)=v_{e}^{i}(u)\right) \Rightarrow\left(\forall v \in O_{e} \cdot v_{e}^{i}(t)=v_{e}^{i}(u)\right)
$$

The proof of theorem 4 (combined actions) requires the following lemma.

Lemma 3 (Corresponding Traces) Given an eligible circuit ( $V, R, C, E, Q_{0}$ ) and two weakly fair sequences $t \in \operatorname{traces}(R, G)$ and $u \in \operatorname{traces}\left(R^{+}, G\right)$, if $u$ corresponds to $t$ then $t \cong u$.

Proof: Because every local formula $L$ can be represented by terms of the form $v_{c}^{k}$ and the operators $\neg$ and $\wedge$, if every term of the form $v_{c}^{k}$ has the same validity and value in $t$ and $u$ then, by induction on the structure of $L$, the lemma holds.

Let $v_{c}^{k}$ be a term of $L$, where $c$ is a component with stable inputs. We will show by induction on the index $i$ of the $k^{\text {th }}$ action of $c$ in $u$ that $v_{c}^{k}$ has the same validity and value in $u$ as in $t$. The induction hypothesis will be

$$
u^{i}=t^{m_{i}}
$$

$\wedge \forall c \in C \cup E$.

$$
\left(\operatorname{occur}(c, u, i)=\operatorname{occur}\left(c, t, m_{i}\right) \wedge\left(v_{c}^{\text {index }(c, u, i)}(u)=v_{c}^{\text {index }(c, u, i)}(t)\right)\right.
$$

Base: $i=0$
This can be discharged by observing that $m_{0}=0, t^{0}=u^{0}$, for every component $c, \operatorname{occur}(c, u, 0)=\operatorname{occur}\left(c, t, m_{0}\right)=0$, and that every local formula $v_{c}^{0}$ is invalid in both $t$ and $u$.

## Step: from $i$ to $i+1$

From the induction hypothesis we know that $u^{i}=t^{m_{i}}$, and for any component $c$ we have $\operatorname{occur}(c, u, i)=\operatorname{occur}\left(c, t, m_{i}\right)$. Consider a component $d$ in $D^{i+1}$. If $d$ has no output choice, then because it is speed-independent and enabled in $t^{m_{i}}$ it must produce the same values on its output no matter what other components in $D^{i+1}$ have acted before it. If $d$ has output choice, it must be in the environment and thus have stable inputs. Because it is enabled in $t^{m_{i}}$ none of its inputs may be changed by the action of any other component, and because environment choices in $t$ and $u$ are defined to be the same it must produce the same output given the same inputs. The combination of the above facts shows that

$$
\forall d \in D^{i} . \operatorname{stable}(d) \Rightarrow\left(\forall v \in I_{d} \cup O_{d} \cdot v_{d}^{o c c u r\left(c, t, m_{i}\right)+1}(t)=v_{d}^{o c c u r\left(c, t, m_{i}\right)+1}(u)\right)
$$

and $u^{i+1}=t^{m_{i+1}}$. Since every element in $D^{i}$ has acted once in both the trace $u^{i-1}, \ldots, u^{i}$ and the trace $t^{m_{i}}, \ldots, t^{m_{i+1}}$ and no other components have acted, we can see that for any component $c$ we have $\operatorname{occur}(c, u, i+1)=\operatorname{occur}\left(c, t, m_{i+1}\right)$.

This proves the claim.

Theorem 4 (Combined Actions) Given an eligible circuit ( $V, R, C, E, Q_{0}$ ), for all local formulas $L$

$$
\left(R, Q_{0}\right) \models L \quad \Leftrightarrow \quad\left(R^{+}, Q_{0}\right) \models L
$$

Proof: The proof will be broken down into two parts, one for each direction of the implication. First, we will show that for any $L$

$$
\left(R, Q_{0}\right) \models L \quad \Rightarrow \quad\left(R^{+}, Q_{0}\right) \vDash L
$$

Let $u$ be any trace in $\operatorname{traces}\left(R^{+}, Q_{0}\right)$. We will create a weakly fair trace $t$ in $\operatorname{traces}\left(R, Q_{0}\right)$ such that $t$ and $u$ correspond. Because we know that $\left(R, Q_{0}\right) \models L$ we know that $L$ is valid and holds in $t$, which by correspondence gives us that $L$ is valid and holds in $u$. Since this is true for any $u$ in $\operatorname{traces}\left(R^{+}, Q_{0}\right)$, this gives ( $\left.R^{+}, Q_{0}\right) \models L$. Using $m_{i}$ as described in definition 16 , the trace $t$ is created inductively using the induction hypothesis

$$
\left(u^{i}=t^{m_{i}}\right) \wedge \forall c \in C \cup E . \operatorname{occur}(c, u, i)=\operatorname{occur}\left(c, t, m_{i}\right)
$$

We start by setting $t^{0}=u^{0}$, which then satisfies the base case of $i=0$ because for all $c$ we know that $\operatorname{occur}(c, u, 0)=\operatorname{occur}(c, t, 0)=0$. The induction step proceeds by extending $t$ by the actions of the components of $D^{i}$ (again, as described in
definition 16) in any order, and by having any components with output choice make the same choices in $t$ as they did in $u$. By its construction, $t$ corresponds with $u$.

To show that

$$
\left(R^{+}, Q_{0}\right) \models L \Rightarrow\left(R, Q_{0}\right) \vDash L
$$

let $t$ be any trace in $\operatorname{traces}\left(R, Q_{0}\right)$. We will create traces $u$ in $\operatorname{traces}\left(R^{+}, Q_{0}\right)$ and $\alpha$ in $\operatorname{traces}\left(R, Q_{0}\right)$ such that $\alpha$ corresponds to $u$, and $t \cong \alpha$. Because $\left(R, Q_{0}\right) \vDash L$ we know that $L$ is valid and holds in all traces $u$ in $\operatorname{traces}\left(R^{+}, Q_{0}\right)$, and from the fact that $u$ corresponds to $\alpha$ and $\alpha \cong t$, we can conclude that $L$ is valid and holds in $t$. Since $t$ and $L$ were arbitrary, we will have proven our claim.

We construct $u$ and $\alpha$ inductively in parallel, with the induction hypothesis that the subtraces $u^{0}, \ldots, u^{i}$ and $\alpha^{0}, \ldots \alpha^{m_{i}}$ correspond and that

$$
\begin{equation*}
\forall c \in C \cup E . \forall v \in I_{c} \cup O_{c} . \forall k \in N . \operatorname{valid}\left(v_{c}^{k}, \alpha\right) \Rightarrow\left(v_{c}^{k}(t)=v_{c}^{k}(\alpha)\right) \tag{A.3}
\end{equation*}
$$

Start by setting $u^{0}=t^{0}=\alpha^{0}$. This trivially gives us the base case. For the induction step, let $D^{i}$ be the set of components enabled in $t^{m_{i-1}}$. Since $t$ is weakly fair, these actions are all performed at some point in $t$. Extend $\alpha$ by appending after $\alpha^{m_{i-1}}$ the actions of the elements of $D^{i}$, acting in any order, and requiring that if they have output choice they produce the same outputs in $\alpha$ as in $t$. This is possible because they are all performed in $t$ and enabled in $t^{m_{i-1}}$. Now let $u^{i}=\alpha^{m_{i}}$. Because $\alpha$ was formed from $t$ using commuting operations equation A. 3 holds, and by construction $\alpha^{0}, \ldots, \alpha^{m_{1}}$ corresponds to $u^{0}, u^{1}$. In this manner we can create $\alpha$ and $u$ such that $u$ corresponds to $\alpha$ and $\alpha \cong t$, which proves our claim.

## Appendix B

## Source Code

This appendix contains the source code for the files used in the verification of the FIFO, the two adders, and the vector multiplier which is described in the body of the thesis. Also included is the Synchronised Transitions (ST) code used to describe the components for the model checking to show that they are speed-independent. Except for the three files describing the FIFO cell, source, and sink, all files are contained in this appendix. An inventory of the files involved is as follows:

Behavioural VHDL building blocks for the FIFO and the two adders
These are all speed-independent components using dual-rail codes for their data.
source.vbe This describes a data source. The VHDL code is contained in Chapter 4.
sink.vbe This describes a data sink. The VHDL code is contained in Chapter 4.
fifocell.vbe This describes a stage in a FIFO chain. The VHDL code is contained in Chapter 4.
maj3.vbe This describes a component for computing the majority of three values.
vjoin.vbe This describes a C-element.
xor3.vbe This describes a component for computing the exclusive or of two values.

## Structural VHDL describing complete FIFO chain and adders

fifochain.vst This describes the structure of the fifo chain.
add4.vst This describes the structure of the buffered 4 bit adder.
fadd4.vst This describes the structure of the unbuffered 4 bit adder.
FL code describing verification of FIFO and the two adders
aspec.fifochain.fl This code verifies the behaviour of the fifo chain.
aspec.add4.fi This code verifies the behaviour of the buffered 4 bit adder.
aspec.fadd4.fl This code verifies the behaviour of the unbuffered 4 bit adder.
Synchronised Transitions code describing FIFO and adder components
mc.sti This contains ST code describing the components which are implemented in behavioural VHDL.

## Behavioural VHDL building blocks for the vector multiplier

These are all speed-independent components using dual-rail codes for their data. They use a slightly different handshaking protocol, so there is some repetition from the components developed for the FIFO and adders.
src.vbe This describes a data source which provides a single bit of data and then stops.
srcn.vbe This describes a data source which can produce multiple bits of independent data before stopping.
sink.vbe This describes a data sink.
alatch.vbe This describes an asynchronous latch.
aswitch.vbe This describes an asymmetric switch.
fadder.vbe This describes a full adder.
fand.vbe This describes an AND gate.
fork.vbe This describes a fork element.
mux.vbe This describes a multiplexing component.
f1.vbe This describes a component for computing a special function for the control portion of the vector multiplier.
f2.vbe This describes a component for computing a special function for the control portion of the vector multiplier.

Structural VHDL describing the building blocks of the vector multiplier pisobit.vst This describes a bit slice of the PISO (parallel in, serial out) component used in the control block of the vector multiplier. component used in the control block of the vector multiplier. The VHDL code is contained in this appendix.
piso4.vst This describes a 4 bit PISO component used in the control block of the vector multiplier.
control.vst This describes the complete control block for the vector multiplier.
masbit.vst This describes a bit-slice of the MAS block in the vector multiplier.
masblock.vst This describes an 8 bit MAS block for the vector multiplier.
vm.vst This describes the complete vector multiplier (without data sinks and sources).
testvm.vst This describes the addition of sinks and sources to the vector multiplier to allow verification.

FL code used in the verification of the vector multiplier
complib.f This contains FL code for making assertions about the basic multiring circuit components.
blocklib.fl This contains FL code for making assertions about the major blocks of the vector multiplier.
aspec.vm.f This contains FL code to verify the behaviour of the vector multiplier.

Synchronised Transitions code describing vector multiplier components
mc.sti This contains ST code describing the components which are implemented in behavioural VHDL.



Z- VHD structural description of a comp
ENTITY fifo is
PORTY data : in bit
1: difo;
END fifoctine structural_view of fifo is



 parit mas


## 









begin

- input
-- inp


grebl ; source
PORT
gVAP I
outt
I>

 // Calculating next state f'n for the fifo chain

load "Hightowex.f1";
load "util.f1";
// Load fl file for description
load fifochain. $f 1^{*}$;



1/ Functions to initialize sinks and sources



" Thia function takes a 11 st of strings naming the fifo cells and
I/ initiallzes them to variables by the same name

lat fifo_list $=\left(\right.$ enumerate ${ }^{\prime \prime}$ fifo 81 );
 let cons = let nim = END-1 in let $n=$ END
$\left(\begin{array}{l}\left.\text { sankdatal } 0]^{\cdot} \text { is inp from nm to } n\right) \\ \text { init_fcells fifo_liat nm } n \text { ): }\end{array}\right.$
let tr_list = flat (nodes fifochain):
lot verify = STE .- pifochain [1 ant cons (1);



## 

letrac $\begin{gathered}11 \text { atof } \\ \text { 11atof } \\ n\end{gathered}$


1/ Functions to initialize sinks and sources




## 

## 



 // Lond f1 file for description
load-fadds. f1\%: - constant






[^1]
Srd: src.vbe.v 1.5 1995/11/07 21:15:13 weih Exp $\$$
This celi suppliss a bit of data with correct handshaking - can either
repeatedly supply data alternating with empties, or just once

and sre
ARCHITECTURE data_flow or src IS
SIGNL GTout, GFout : REG_BIT REGISTER;
SIGAL done, gData : REG_BIT REGISTER;
mem
Init : BLOCK (init )
BEGIN
gData < $=$ GUARDED data
END BLOCK;

END data_flow;

ef : block ( (at or af) and (bT or bF) and ef and not ackAO and Not ackbo


ARCHITECTURE data_flow of aswitch IS
SIGNAL gAOT, gAOF, gBot, gBoF, gAckA, gAckB, lastc : REG_BIT REGISTER;
begin
-- pass empties (with lastc T),
passt : BLOCK ( NOT (bT OR bF ) AND NOT ( ct OR cF ) AND ackAO AND lastc )
BEGIN
 BEGIN AND ackBO AND ackAO AND NOT Laste



 END
ARCHIT
SII
BEGIN

 End block: OR ( (NOT in1T) and (Not in2T) ) ):




 and







 or function ${ }^{22}$ for the
or multi－ring circuits）
a蔵

－－SId：pisobit．vst，v 1.3 1996／04／07 19：54：49 weih Exp \＄
－－Description of a bit of the PISO structure
－－from paper on MRing circuits
 ARCHITECTURE structural＿view or pisobit is
COMPT
PORENT alatch
－－Transition for calculating data

気动合
天 \％\％





Exp $S$

－－SId：control．vst，v 1.4 1996／04／07 19：55：35 weih Exp $\$$
－－Description of the control block of the vector multiplier
－－from paper on MRing circuits



它言苋


[^2]











- SId: masblock.vst, v 1. 1996/04/07 19:55:20 wein Exp $\$$.
- Module describing an 8 bit MAS block as per the MRing paper


















## // SId: aspec.vm.f1,v 1.17 1996/04/07 21:32:48 weih Exp \$ // Code for verifying a vm unit - using testvm.vst

 // This has been changed to handie multiple inputs. starting with an empty // multiplier, and ending with Last a ${ }^{\text {T }}$. // These have been put in a loader module to simplify reloading/ 1 load "testvm.fl";
/10ad "newarithm.f1"; /looad $\cdot$ Highlowex.f1";
load "blocklib.t1:; $\begin{array}{rll}\text { letrec firstn } n & 1]=[1] \\ \text { Nirsta } & 0 & 1=[1]\end{array}$ let addnm $\mathrm{nm} \mathrm{s}=\mathrm{nm}^{\wedge} \mathrm{s}$; let FOREVER $=2500 ;$
let PENUE $=2499 ;$ // Use Last to control the number of operands consumed by the vil
let Last $=(F, T)$; let nops = length Last


[^3]

${ }_{(1)}^{\text {MPL }}$ Erom verification Import Initially;

$\left.\begin{array}{c}\text { TYPE } \\ \text { BUEC } \\ n\end{array} \operatorname{:~INTEGER}\right)=\operatorname{ARRAY}(i: \operatorname{INDEX}(n)):$ BCOLEAN;
static
(* Concrete cells *)
$\underset{\text { BEGIN }}{\text { alatch }}$ : cAlatchCell $=$


BEGIN astch : cAswitchCell a








$$
\begin{aligned}
& \text { sink1 }: \text { cSinkCell }= \\
& \text { STaTE } \\
& \text { regT, regF, done : BOOLEAN; }
\end{aligned}
$$

[^4]\[

$$
\begin{aligned}
& \text { regT. regF, done : BOOLEAN; } \\
& \text { BEEIN }
\end{aligned}
$$
\]

END;

$$
\begin{aligned}
& \text { STATE } \\
& \text { BEOLVLV : BCoLEAN: }
\end{aligned}
$$

absSrc : aSreCell =

BEGIN
END outV = ack -> outV : $=$ NOT ack >>
absSink : aSinkCell a
BEGIN $\ll$ NOT (inV = ackI ) $\rightarrow$ ackI := inV >>
END;
END.


[^0]:    "Behavioural" VHDL is used to describe the functionality of the components of the circuit and the environment, and "structural" VHDL is used to describe the

[^1]:    
    

[^2]:    ；

[^3]:    $1 /$ The leading zero pattern for the nth input
    let zp
    

[^4]:    

