EQUIVALENCE VERIFICATION FOR NULL CONVENTION LOGIC AND LATENCY-INSENSITIVE CIRCUITS

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Vidura Manu Wijayasekara

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Electrical and Computer Engineering

April 2016

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

EQUIVALENCE VERIFICATION FOR NULL CONVENTION LOGIC AND LATENCY-INSENSITIVE CIRCUITS

By

Vidura Manu Wijayasekara

The supervisory committee certifies that this dissertation complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Sudarshan K. Srinivasan
Chair

Dr. Scott C. Smith

Dr. Debasis Dawn

Dr. Kenneth Magel

Approved:

8 April 2016
Date

Dr. Scott C. Smith
Department Chair

ABSTRACT

NULL convention logic and latency-insensitive circuits are delay-tolerant circuits that can be synthesized from a synchronous specification. These design paradigms can use existing CAD flows to implement circuits that are robust to process variations and wire delays. Verification is an indispensable phase of any commercial design cycle, and needs to be addressed in order to exploit the potential advantages of these design paradigms. Delay-tolerant circuits are of asynchronous nature. Therefore, timing behavior of these delay-tolerant circuits are very disparate from the synchronous specifications, and verifying equivalence of the synthesized circuit to the synchronous specification is one of the main challenges. However, there is no existing work in the literature that address this challenge and still remains an open problem. This study makes an initial effort in developing equivalence verification methods and equivalence checking tools for these design paradigms.

ACKNOWLEDGEMENTS

I would like to thank Dr. Sudarshan K. Srinivasan, Dr. Scott C. Smith, Dr. Debasis Dawn, and Dr. Kenneth Magel for serving in my graduate committee. Dr. Sudarshan K. Srinivasan is my Ph.D. Advisor, and I am very thankful for guidance, and wisdom he gave me through out my research. He has shown me the true satisfaction of research, and I enjoyed many discussions we had on the subjects of formal verification.

I would like to thank Dr. Hyunsook Do for serving in my graduate committee. I appreciate your comments and input to improve this research.

NSF funded my research for many years through grants CCF-1117164 and CCF-1242043, and I am very grateful.

Finally, I like to thank my family for their relentless support in every way, and my wife for giving me strength thorough out this work.

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1. Background

With technology scaling it is becoming increasingly difficult to design digital circuits using a synchronous approach. Previously ignored design challenges such as process variability and wire delays have significant effects on smaller feature size circuits. Process variations lead to stretching of timing margins in static timing analysis resulting in too conservative designs [1]. As transistors become smaller and faster, wire delays become relatively large [2]. However, wire delays cannot be determined until later design stages (e.g. layout stage). If a long wire path is in the root of a timing violation, it is likely to cause more iterations in the design cycle. This is called the *wire-delay challenge* and could prohibitive prolong the design process. Designing the global clock to keep the clock skew at an acceptable level require multiple drivers and more area of the chip. In fact, clock logic itself contributes to increase in power dissipation, which hinders performance of the circuit [3].

Mixed signal environment of System-on-Chip (SoC) designs introduce a set of new problems, which are termed *more-than-Moore*, such as harsh power and timing conditions; and require logic to be robust under such circumstances [4]. Hence, the ability of the synchronous design style to deliver economical and reliable solutions is being threatened.

Asynchronous circuits are clock-less circuits that take a feedback closed-loop control approach for synchronizing the data flow. Therefore, asynchronous circuits are more robust to varying computational delays caused by process, voltage, temperature (PVT) variations, long wire paths, etc. Also, these circuits require less power, and produce less electromagnetic interference. In addition to requiring less power, asynchronous circuits can operate robustly under modern circuit-level power minimization techniques, such as dynamic voltage scaling.

However, using a local synchronization approach, instead of a global approach such as a clock signal, means different stages of the circuit can progress at different speeds making the circuit highly concurrent. Consequently, asynchronous circuits are harder to design and verify. Additional circuit elements needed for hand shaking circuits, data encoding, etc. render area overhead. Hence, although asynchronous design paradigm has been around for 50+ years, asynchronous circuits are primarily utilized for niche markets and in the research domain.

1.2. Motivation

However, as transistor size continues to decrease, asynchronous circuits are being looked to by industry more and more to solve power dissipation and process variability issues associated with these smaller feature size circuits. Presently, companies like Intel and ARM incorporate asynchronous logic to some of their designs using their own proprietary tools [3].

One reason asynchronous design style is avoided for digital circuit design is lack of complete CAD flows, which is essential to design large, realistic designs [1]. Much work has been done in developing new design methodologies that could implement delay-tolerant circuits from a synchronous design [5, 6, 7, 8]. These design methods can exploit existing EDA tools and CAD flows to design delay-tolerant circuits. Such approaches can be divided into three categories:

- Synchronous Elastic Circuits: synchronous elastic circuits are clock-based latency insensitive circuits. Elastic circuits are typically synthesized from synchronous circuits. After synthesis, additional buffers can be arbitrarily inserted in the data path of an elastic circuit without altering its functionality to resolve timing issues.

- Null Convention Logic (NCL) Circuits: NCL circuits are quasi delay-insensitive circuits that can be synthesized from a synchronous design. They make minimal timing assumptions (isochronic forks), and performance is close to average-case performance.

- Desynchronized Circuits: desynchronization is proposed as a design solution, to synthesize bounded-delay asynchronous circuits from synchronous parents in a manner that exploits existing CAD tool support for synchronous designs.

Development of tools to automate the synthesis of the above circuits is largely addressed [9, 10, 11, 12, 13]. So is the verifying correctness of components used in the control networks; and certain properties of the circuit, such as delay insensitivity. Nonetheless, equivalence verification of these circuits is still largely unaddressed. In fact, according to Jordi Cortadella, a top and pioneering researcher in developing latency-insensitive systems, equivalence checking is one of the main challenges and remains an open problem; some of the work done in this study is the initial effort to address it [14].

1.3. Problem Statement

Equivalence verification of the synthesized NCL and latency-insensitive circuits against the synchronous counterpart still remains an open problem [14]. This is one the main challenges as synthesis tools and manual tinkering can introduce bugs to the design. This study makes the initial attempt in developing equivalence verification methods and tools for Null Convention Logic circuits, and delay-insensitive circuits.

1.4. Equivalence Verification

We use the notion of Well-Founded Equivalence Bisimulation (WEB) refinement for the equivalence verification problem. A formal and detailed description of WEB refinement is provided in [15][16]. Here, we provide a brief overview of the key features of WEB refinement relevant to the problem at hand. WEB refinement is a notion of equivalence that can be used to check if an implementation system satisfies its specification system, even if the implementation and specification are defined at very disparate levels of abstraction.

In the context of refinement, digital systems are modeled as transition systems (TSs). A TS is a three tuple and includes the set of states of the system, a transition relation that defines the state transitions of the system, and a labeling function that defines what is observable in each state. The behaviors of a system modeled as a TS are defined using the notion of paths. A *path* in a TS is a sequence of states, say s^0,s^1,s^2, ..., where s^0 is the initial state and s^0 transitions to s^1, s^1 transitions to s^2, and so on. An infinite sequence of such states is a *full path.* The behaviors of a system is the set of all full paths in the TS of the system.

The implementation behaves correctly as given by the specification, if every behavior of the implementation is matched by a behavior of the specification and vice versa. However, the implementation and specification may not have the same timing behavior. For example, the implementation may take many steps to match a single step of the specification. This phenomenon is known as stuttering. To account for such situations, multiple but finite transitions of the implementation system are allowed to match a single transition of the specification system.

Refinement-Based Correctness Formula: Manolios [16] has shown that it is enough to prove the following refinement-based correctness formula to establish the equivalence of an implementation and specification based on WEB refinement. *rank* is used to distinguish stutter from deadlock. *rank* is a witness function from implementation states to natural numbers whose value decreases when the implementation stutters.

Definition 1. *(Refinement-Based Correctness Formula)*

$$\begin{aligned}\Big\langle \forall w \in I :: \;\; & s = \mathit{refinement\text{-}map}(w) \wedge u = \mathit{SStep}(s) \;\; \wedge \\ & v = \mathit{IStep}(w) \wedge u \neq \mathit{refinement\text{-}map}(v) \\ \Rightarrow \;\; & s = \mathit{refinement\text{-}map}(v) \wedge \mathit{rank}(v) < \mathit{rank}(w)\Big\rangle\end{aligned}$$

In the above, *SStep()* and *IStep()* are functions that define the transitions of the implementation and specification. I is the set of all the implementation states reachable from the initial states of the implementation. Note that in order to establish the equivalence of an implementation and specification system, all the reachable states of the implementation should be checked to see if they satisfy the above correctness formula. The correctness formula given above is expressible in a decidable fragment of first-order logic. Therefore, verifying equivalence of an implementation and a specification based on WEB refinement can be accomplished automatically using an SMT solver if a suitable refinement map and rank function are available.

2. EQUIVALENCE CHECKING FOR SYNCHRONOUS ELASTIC CIRCUITS[1]

2.1. Introduction

Latency insensitive (LI) [6], [17] design addresses the wire delay challenge for nanometer technologies within the synchronous framework. The central idea is to use relay stations—which function like latches—to break long wires that cause violations of timing requirements imposed by the clock. A handshaking protocol (known as LI protocol) is used to allow for insertion of buffers/relay stations without altering the functionality of the system. One of the primary impacts is that LI design aids in intellectual-property reuse in systems-on-chip by reducing the expensive iterations required for timing closure [18]. LI design is a very active area of research in both academia and industry and many LI design paradigms, implementations, and optimizations have been proposed [6], [17], [8], [19], [20], [21], [22].

Synchronous Elastic Networks (SEN) [8], [23] is one effective approach to implement LI designs and also synthesize LI systems from synchronous parents. The idea with SEN is to replace all flip flops with elastic buffers (EBs) that are constructed from two elastic half buffers (EHBs), namely a master EHB and a slave EHB. EHBs are gated latches whose clock input is produced by elastic controllers that are used to implement the LI protocol. The clock network is replaced by a network of elastic controllers, where each controller is used to control the elastic buffers in a design stage and synchronize with the controllers of adjacent design stages. In the resulting elasticized design, elastic buffers can be inserted in any place in the data path to break long wires. Figure 2.1 shows the example of an elasticized processor data path with two additional elastic buffers.

[1] The material in this chapter was co-authored by Vidura M. Wijayasekara and Sudarshan K. Srinivasan. Vidura Wijayasekara was the primary developer of the conclusions that are advanced here.Vidura Wijayasekara also drafted and revised all versions of this chapter. Sudarshan Srinivasan served as proofreader and checked correctness of the theories and algorithms developed by Vidura Wijayasekara. ©2013 IEEE. Reprinted, with permission, from V. Wijayasekara and S. K. Srinivasan, "Equivalence Checking for Synchronous Elastic Circuits," *Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on*, 10/2013.

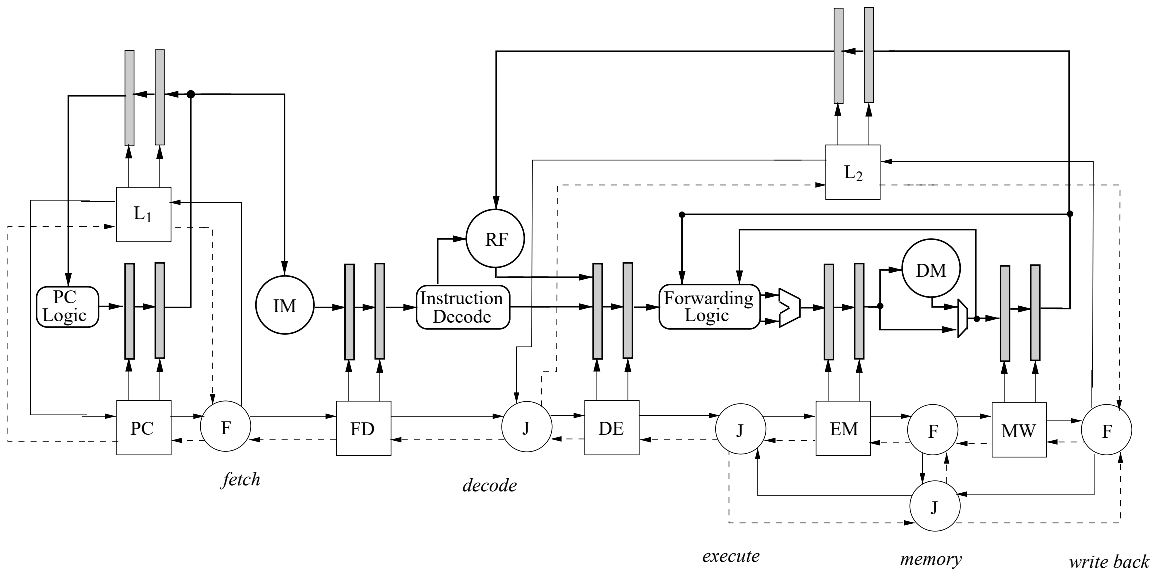


Figure 2.1. High-Level Organization of a 5-Stage Elastic Processor with Two Additional Elastic Buffers L1 and L2. The J and F Blocks Denote the Join and Fork Structures.

In this work, we present an automated equivalence checker that verifies the functionality of the elastic circuit (even after the inclusion of any number of arbitrarily placed additional elastic buffers) against its synchronous parent circuit. Why would such an equivalence checker be useful in practice? Consider the IP reuse based SoC design paradigm. The SoC design is constructed from digital IP components. Some of these IP components are obtained from third party vendors, some are proprietary IPs, while others may be designed from scratch. All these IP components have to be integrated together in a technology (that some of these IPs may not have been designed for). At this stage in the design cycle, some of the IPs will require infeasible and expensive redesign to resolve timing issues that arise in the new technology. The designer can instead opt to generate elastic versions of these IPs and solve the timing issues in the elastic domain without requiring expensive redesign. However, after generating the elastic design and fixing all timing issues using additional elastic buffers, the resulting elastic IP will have undergone a considerable transformation w.r.t. the original synchronous IP. The designer cannot assume that the resulting elastic IP will not have bugs. The designer will instead have to expend time and resources to verify the elastic IP.

Our fully automated equivalence checker comes to the aid here and addresses the verification problem of the elastic IP. One may argue that the transformations used in generating elastic circuits are already proved to be correct and so verification is not required. However, the IP circuits will undergo considerably significant mutations and there could be many sources of error in the elasticization process (such as buggy synthesis tools, and manual tinkering of the circuit). Also, commercial design processes will not assume the functional correctness of the resulting design and will require verification.

The equivalence checker takes as input the elastic and synchronous circuits in RTL VHDL. The checker can currently handle closed circuits, where the behavior of the elastic controller network is deterministic. For future work, we will extend the techniques implemented in the checker to handle elastic controllers with non-deterministic behaviors. The equivalence verification proof obligations are automatically generated. An SMT solver is used in the back end to check the proof obligations. The notion of equivalence used is described in Section 2.2. The equivalence verification algorithms incorporated in the tool are described in Section 2.3. The overall tool flow is described in Section 2.4. Experimental results, related work, and conclusions are given in Sections 2.5, 2.6, and 2.7, respectively. The capacity of the equivalence checker is demonstrated with results from 24 elastic circuit benchmarks. *The benchmarks and tools required to reproduce our results is available in [24].*

2.2. Background: Equivalence Notion

The notion of equivalence we use is Well-Founded Equivalence Bisimulation (WEB) refinement [15][16]. A formal and detailed description of WEB refinement is provided in [15][16]. Here, we provide a brief overview of the key features of WEB refinement relevant to the problem at hand. WEB refinement is a notion of equivalence that can be used to check if an implementation system satisfies its specification system, even if the implementation and specification are defined at very disparate levels of abstraction.

In the context of refinement, digital systems are modeled as transition systems (TSs). A TS is a three tuple and includes the set of states of the system and a transition relation that defines the state transitions of the system. The behaviors of a system modeled as a TS are defined using the notion of paths. A *path* in a TS is a sequence of states, say s^0,s^1,s^2, ..., where s^0 is the initial state and s^0 transitions to s^1, s^1 transitions to s^2, and so on. An infinite sequence of such states is a *full path.* The behaviors of a system is the set of all full paths in the TS of the system. The implementation behaves correctly as given by the specification, if every behavior of the implementation is matched by a behavior of the specification and vice versa. However, the implementation and specification may not have the same timing behavior. For example, if the implementation is an elastic circuit and the specification is a synchronous circuit, the elastic circuit may take many steps to match a single step of the synchronous circuit. This phenomenon is known as stuttering. To account for such situations, multiple but finite transitions of the implementation system are allowed to match a single transition of the specification system.

Another issue is that to check equivalence, synchronous states and elastic circuit states need to be compared. However, these states look very different. While the synchronous states have flip-flops, elastic circuit states have elastic buffers and also possibly additional buffers. WEB refinement employs refinement maps, functions that map implementation states to specification states to bridge this abstraction gap.

Refinement-Based Correctness Formula: Manolios [16] has shown that it is enough to prove the following refinement-based correctness formula to establish the equivalence of an implementation and specification based on WEB refinement. *rank* is used to distinguish stutter from deadlock. *rank* is a witness function from implementation states to natural numbers whose value decreases when the implementation stutters.

Definition 2. *(Refinement-Based Correctness Formula) For every implementation state w, let s be a specification state such that s = refinement-map(w). If u is the specification successor of s (u = Sstep(s)) and v is the implementation successor of w (v = Istep(w)), then one of the following has to hold:*

1. $u = r(v)$ {*non-stuttering step*}

2. $s = r(v) \wedge rank(v) < rank(w)$ {*stuttering step*}

In the above, *Sstep()* and *Istep()* are functions that define the transitions of the implementation and specification. Note that in order to establish the equivalence of an implementation and specification system, all the reachable states of the implementation should be checked to see if they satisfy the above correctness formula. The correctness formula given above is expressible in a decidable fragment of first-order logic. Therefore, verifying equivalence of an implementation and a specification based on WEB refinement can be accomplished automatically using an SMT solver if a suitable refinement map and rank function are available.

2.3. Automating Computation of Refinement Maps

If an implementation is a refinement of a specification, then a refinement map does exist [25]. However, finding/constructing a refinement map can be very challenging in practice and can require deep understanding and analysis of the systems being compared. Also, often times, the refinement map can be computationally expensive, resulting in the approach being infeasible. The primary contribution of this work is a fully automated procedure to compute refinement maps for the elastic verification problem described in the introduction section. The refinement maps synthesized by our procedure leads to an efficient and scalable verification approach. The key idea is to use reachability analysis of the elastic controller network (using the notion of token-flow diagrams) to systematically synthesize refinement maps.

We use the example of a 5-stage elastic processor circuit (shown in Figure 2.1) to illustrate the ideas presented in the rest of the paper. We call this elastic circuit example E-2. The elastic processor pipeline has 5 elastic buffers corresponding to pipeline latches PC, FD, DE, EM, and MW. We also inserted two additional elastic buffers at arbitrary points in the data path that are labeled *L1* and *L2*. The elastic controller network is also shown in the figure. The controllers use *valid* (indicated by solid lines) and *stop* signals (indicated by dashed lines) to implement the LI protocol. The synchronous parent of E-2 would have regular registers instead of EBs, and will not have the additional EBs. A clock signal would replace the entire elastic controller network.

2.3.1. Token-Flow Diagrams for Elastic Circuits

In this section, we describe a procedure for generating token-flow diagrams for elastic circuits. Token-flow diagrams are used for reachability analysis of the elastic control layer and also for synthesizing refinement maps.

S_y is the set of stages in the synchronous circuit. It is an ordered set. Each stage in S_y is hence identified by a unique number i, $0 \leq i < |S_y|$. E is the set of stages in the elastic circuit. E is an ordered set. Each stage in E is hence identified by a unique number i, where $0 \leq i < |E|$. Each stage of the elastic circuit corresponds to an elastic buffer and vice versa. The elastic buffers are classified as non-additional elastic buffers and additional elastic buffers. Note that each non-additional EB would correspond to a stage in the synchronous parent circuit. An EB is described to be in an empty, half, or full state, if the EB holds 0, 1, or 2 valid data units, respectively.

Definition 3. *The token state of an elastic controller network circuit, T_E is defined as the set* $\{\langle m_0, s_0 \rangle, \langle m_1, s_1 \rangle, ..., \langle m_{|E|-1}, s_{|E|-1} \rangle\}$ *such that* $m_0, m_1, ..., m_{|E|-1}, s_0, s_1, ..., s_{|E|-1} \in \mathbb{N}$.

The token state is used to capture the distribution of valid data in the elastic circuit. m_i and s_i indicate the token values corresponding to the master and slave EHBs, respectively. EHBs without valid data are assigned the token value 0. EHBs with positive non-zero token values indicates valid data. Unique data tokens are identified by unqiue token values.

We next define elastic token-flow diagrams (*etfd*), which are used to compute refinement maps.

Definition 4. *An elastic token-flow diagram (etfd) is a finite sequence of elastic token states such that for any adjacent pair of token states in the sequence, say* T_{Ei} *and* T_{Ej}*, such that* T_{Ei} *and* T_{Ej} *are elastic token states corresponding to the elastic controller network states* e_i *and* e_j*, then* $(e_i , e_j) \in \rightarrow_{ecn}$*, where* $\rightarrow_{ecn}$ *is the transition relation of the elastic controller network circuit.*

The token-flow diagram for a sequence of states of the E-2 elastic circuit is shown in Table 2.2. First, we define a procedure that given a token state of the elastic circuit, computes the next token state of that circuit. The procedure also takes as input the connectivity matrix of an elastic circuit M_C, which is defined as follows.

Definition 5. *The connectivity matrix of an elastic circuit* $M_C = [a_{ij}]_{|E|\times|E|}$ *such that* $a_{ij} = 1$ *if there is a data channel from EB* i *to EB* j. *Otherwise* $a_{ij} = 0$.

Table 2.1. Datapath Connectivity Matrix M_c for Benchmark E-2

	PC	FD	DE	EM	MW	L1	L2
PC	0	1	0	0	0	1	0
FD	0	0	1	0	0	0	0
DE	0	0	0	1	0	0	0
EM	0	0	0	1	1	0	0
MW	0	0	0	1	0	0	1
L1	1	0	0	0	0	0	0
L2	0	0	1	0	0	0	0

Another data structure used in the procedure is the token transition matrix M_{TT}. For EBs with multiple destinations, it is possible that in a cycle, data is transferred in only some of the output channels, but not all. To keep track of this, we use the M_{TT} matrix. Initially, all entries in the matrix are assigned a value 0. If data transfers from a source EB on only some but not all output channels, then the channels on which the transition takes place is given a value 1 in M_{TT}. When in the future, the data from the source is transferred on all output channels, then all the output channels from that source are reassigned a value 0 in M_{TT}.

Note that unlike synchronous circuits, data need not always transfer from source EB to destination in a given cycle. In fact, data transfers only when valid data is available at the source EB and the destination EB is ready to accept data (which is indicated by deasserting the stop signal). Therefore, we define the function *ValidDataInputs(i)* that determines if all the sources of a destination EB have valid data.

Definition 6. *ValidDataInputs(i)* =

$$\bigwedge_{0 \le j < |E|} \Big\{ \big(M_c[j][i] = 1\big) \rightarrow \big(M_{TT}[j][i] = 0 \wedge s_j \neq 0\big) \Big\}$$

In the above definition, a destination EB i has valid data at all its sources j only if slave EHB (s_j) is not empty, and that data in s_j has not previously transferred on that channel ($M_{TT}[j][i] = 0$). We only check those EBs that are sources to EB i ($M_c[j][i] = 1$).

Procedure 1 Next token calculation for master EHBs

```
1: for i ← 0 to $|E| - 1$ do
2:     if $m_i \neq 0$ then
3:         $m'_i \leftarrow m_i$
4:     else if ValidDataInputs(i) then
5:         $TokenGenerator \leftarrow TokenGenerator + 1$
6:         $m'_i \leftarrow TokenGenerator$
7:         for j ← 0 to $|E|$ -1 do
8:             $M_{TT}[j][i] \leftarrow 1$;
9:     else
10:        $m'_i \leftarrow 0$;
```

The algorithm that computes the next token state is given in two steps. In the first step (shown in Procedure 1), the next token value of all master EHBs (m'_i) is computed. The procedure enumerates through all the master EHBs and uses the following property of elastic circuits.

Property 1. *[8] For any elastic buffer in a half state (one of the EHB has valid data and the other EHB is empty), the master EHB will be empty and the slave EHB will have valid data.*

As can be inferred from the above property, if the master EHB is not empty, then the corresponding EB is full, meaning that both master and slave EHBs hold valid data. In this state, the master EHB retains its previous value and the EB does not accept any new data. Otherwise, if the master EHB is empty, then the procedure checks to see if all the sources to EB i have valid data. If this is the case, a new unique token number is generated and assigned to m'_i, the next value of m_i. The *TokenGenerator* is a counter that is initialized to a natural number value greater than the greatest token value in the input elastic token state. The M_{TT} matrix is updated. If otherwise, one or more of the sources do not have valid data, then the Master EHB will become empty. Therefore m'_i is assigned zero in this case.

Procedure 2 Next token calculation for slave EHBs

```
1:  for i ← 0 to $|E| - 1$ do
2:      if $s_i = 0$ then
3:          $s'_i \leftarrow 0$
4:      else
5:          if DataTransferredAll(i) then
6:              $s'_i \leftarrow 0$
7:              for j ← 0 to N-1 do
8:                  $M_{TT}[i][j] \leftarrow 0$
9:          else
10:             $s'_i \leftarrow s_i$
11:     if $(s'_i = 0) \wedge (m'_i \neq 0)$ then
12:         $s'_i \leftarrow m'_i$
13:         $m'_i \leftarrow 0$
```

Definition 7. *DataTransferredAll(i)* =

$$\bigwedge_{0 \leq j < |E|} \Big\{ \big(M_c[i][j] = 1\big) \rightarrow \big(M_{TT}[i][j] = 1\big) \Big\}$$

The second step of the algorithm computes the next value of slave EHBs and is shown in Procedure 2. If the slave EHB is currently empty, then it will remain empty. Otherwise, if transfers have taken place on all the output channels (determined by the *DataTransferredAll* function), then the slave EHB can let go of its current token value and is updated to empty. The *DataTransferredAll* function is defined in Definition 7.

The *DataTransferredAll(i)* function examines only those entries in M_{TT} corresponding to EB i that are destinations of EB i ($M_c[i][j] = 1$). The functions examines the output channels of EB i to determine if transfers have been completed on these channels ($M_{TT} = 1$).

Also, the entries in the M_{TT} matrix corresponding to the output channels of EB i are assigned a value 0 to indicate that transfers have taken place on all the output channels from EB i. If there are still output channels in which transfers are yet to be completed, then the slave EHB retains its token.

After the slave EHBs have been updated (lines 1-13 of Procedure 2), the procedure checks the new token values of the master and slave EHBs. If the EB is in a half state where the slave is empty and the master has a token, then this token is transferred from master to slave (lines 14-16 of Procedure 2), due to Property 1. This completes the computation of the next token state of the elastic circuit.

2.3.2. Reachability for Elastic Controller Networks

The reachable states of an elastic controller network can be computed using token-flow diagrams. At reset, elastic buffers are initialized to the half state and additional buffers are initialized to the empty state. Such a reset state is a requirement of the SEN paradigm.

Definition 8. *The binary token state corresponding to the token-state T_E of an elastic circuit is defined as the set $\{\langle\ bm_0,\ bs_0\ \rangle,\ \langle\ bm_1,\ bs_1\ \rangle,\ ...,\ \langle\ bm_{|E|-1},\ bs_{|E|-1}\ \rangle\}$ such that*

$$bm_i/bs_i = \begin{cases} 1 & m_i/s_i > 0 \\ 0 & m_i/s_i = 0 \end{cases}$$

To compute token-flow diagrams for reachability, the initial token-state is constructed by assigning a token value 0 to the empty EHBs and a unique non-zero natural number token value to each of the non-empty EHBs.

Table 2.2. Token Flow Diagram for E-2

State	0 (pc)		1 (fd)		2 (de)		3 (em)		4 (mw)		5 (l1)		6 (l2)	
	m	s	m	s	m	s	m	s	m	s	m	s	m	s
0	0	1	0	2	0	3	0	4	0	5	0	0	0	0
1	0	0	1	2	0	0	0	6	0	4	0	1	0	5
2	0	7	0	1	0	8	0	6	6	4	0	0	0	4
3	0	0	0	7	0	9	0	10	0	6	0	7	0	0
4	0	11	0	7	0	0	0	12	0	10	0	0	0	6
5	0	0	0	11	0	13	0	12	12	10	0	11	0	10
6	0	14	0	0	0	15	0	16	0	12	0	0	0	0
7	0	0	0	14	0	0	0	17	0	16	0	14	0	12
8	0	18	0	0	0	19	0	17	17	16	0	0	0	16
9	0	0	0	18	0	0	0	20	0	17	0	18	0	16
10	0	21	0	0	0	22	0	20	20	17	0	0	0	17
11	0	0	0	21	0	0	0	23	0	20	0	21	0	17
12	0	24	0	0	0	25	0	23	23	20	0	0	0	20
13	0	0	0	24	0	0	0	26	0	23	0	24	0	20

We use the notion of a binary token state for reachability, which is given in Definition 8. The binary token state captures the distribution of data in the elastic circuit without distinguishing the data units. Thus, two elastic controller network states with the same binary token states are essentially equivalent.

The reachable states of the elastic controller network are computed by simulating the token-flow diagram until a binary token state is reached that has already been visited before. Note that since we consider only deterministic elastic controller networks, the reachable states will be a finite sequence of states that will converge and hence can be computed as an *etfd*. The resulting token states correspond to the reachable states of the controller network of the elastic circuit. Table 2.2 shows the token-flow diagram for computing the reachable states of the E-2 benchmark. As can be seen from the table, states 7, 9, 11, and 13 map to the same binary token states. Also, states 8, 10, and 12 map to the same binary token states. Therefore, the reachable states of the elastic controller network of E-2 are states 0 through 8, after which the states start to repeat.

2.3.3. Token-Flow Diagrams for Synchronous Circuits

Token flow diagrams for the synchronous parent circuit (*stfd*), which is the specification, are also computed. Given an *etfd*, an *stfd* captures how the tokens in the *etfd* would progress in the synchronous parent circuit. The token-flow diagram for the synchronous circuit corresponding to the *etfd* of Table 2.2 is shown in Table 2.3. The refinement map is constructed by examining the token states of the elastic and the synchronous circuits.

Table 2.3. Token Flow Diagram for Synchronous Circuit of E-2

State	0 (pc)	1 (fd)	2 (de)	3 (em)	4 (mw)
0	1	2	3	4	5
1	7	1	8	6	4
2	11	7	9	10	6
3	14	11	13	12	10
4	18	14	15	16	12
5	21	18	19	17	16
6	24	21	22	20	17

As such, *etfd* and *stfd* are matrices, where rows correspond to token states and columns correspond to design stages. In the *etfd* matrix, each element corresponds to two token values $etfd[i,j]_m$ and $etfd[i,j]_s$, indicating master and slave tokens, respectively.

The synchronous circuit is comprised of registers as opposed to EBs. Therefore, a stage in the design has only one token value in contrast to two token values (master and slave) for EBs. Second, the registers always hold valid data and therefore always have a non-zero token value. Note that the notion of valid data is w.r.t. elastic control and is used to contrast the progress of data in the elastic and synchronous circuits. For example, bubbles in the design due to synchronous control (for example, pipeline control) are still considered to be valid data in both circuits. Invalid data are only the result of bubbles introduced by elastic control.

Property 2. *The sequence of unique non-zero token values in any column of an stfd and the corresponding slave column of etfd are identical.*

The synchronous specification and the elastic implementation are flow equivalent [8]. Therefore, the flow of tokens in a stage of the synchronous specification circuit can be obtained by removing bubbles (tokens with value 0) and stale tokens (token values duplicated in the sequence) from the flow of tokens of the slave EHB of the corresponding stage in the elastic circuit. The slave EHB is chosen because all the valid data that flow through an EB is retained in the slave EHB for at least one cycle. An *stfd* satisfies the above Property 2. Therefore, we construct the *stfd* column by column. Each column of the *stfd* is obtained by removing the tokens with value zero and duplicate tokens from the column in *etfd* corresponding to the slave EHB.

2.3.4. Refinement Map Computation

The refinement map for the equivalence verification problem at hand takes as input an elastic circuit state and returns the corresponding synchronous circuit state. Due to the presence of additional buffers, design stages in the elastic circuit progress at different speeds when compared to corresponding stages in the synchronous circuit. For example, in the E-2 benchmark and its corresponding synchronous specification, for states with the same program counter value, the value in the decode stage may not be the same. Thus to map elastic states to synchronous states, we choose a design stage and use it as a reference point to construct the mapping. An elastic state w will be mapped to a synchronous state s such that the values of the latches/registers in the stage corresponding to the reference point in both w and s match.

Next, we want to define a projection function that given an elastic state as input, constructs the corresponding synchronous state. A systematic approach to computing the projection function is possible by comparing the distribution of data tokens in an elastic state and its synchronous counterpart. The token-flow diagrams can be used to determine the distribution of data tokens in the reachable states of both elastic and synchronous circuits.

Procedure 3 Refinement Map

```
1: for $elas\text{-}state \leftarrow 0$ to $|etfd|$-1 do
2:     $h \leftarrow elas\text{-}state$
3:     repeat
4:         $t_p \leftarrow etfd[h, reference\text{-}point]_s$
5:         $h \leftarrow h - 1$
6:     until $t_p \neq 0$
7:     $sync\text{-}state \leftarrow 0$
8:     while $stfd[sync\text{-}state, reference\text{-}point] \neq t_p$ do
9:         $sync\text{-}state \leftarrow sync\text{-}state + 1$
10:    for j $\leftarrow 0$ to $|S_y|$-1 do
11:        $h \leftarrow 0$
12:        while $etfd[elas\text{-}state - h, j]_s \neq stfd[sync\text{-}state, j]$ do
13:            $h \leftarrow h - 1$
14:        $refinement\text{-}map[elas\text{-}state, j] \leftarrow h$
```

Since the distribution of valid data is different for different reachable states of the elastic controller network, one projection function is defined for each controller reachable state by examining the token flow diagrams of the elastic and synchronous circuits, *etfd* and *stfd*, respectively. Note that a reachable state of the controller network corresponds to many states of the elastic circuit. The objective of the projection function is to construct the synchronous state from the elastic state. To achieve this, history information may be used. For example, to get the value of the *fd* latch, for the synchronous state from the elastic state, the master and slave EHBs of *fd* can be examined. However, it is also possible that *fd* is empty or the *fd* value in the elastic state is not the required value to construct the synchronous state. Note that this is because while stages in the synchronous circuit progress together, stages in the elastic circuit can progress at different rates. In such a situation, the projection function can examine history values of *fd* to obtain the required value.

As such, the refinement map is a matrix $\textit{refinement-map}_{|etfd| \times |S_y|}$, where each row corresponds to a reachable state of the elastic controller network. Each row has one entry for each stage in the synchronous circuit. Each $[i, j]$ entry in *refinement-map* indicates which history should be projected for design stage j in controller state i. For example, an entry of 0 indicates the current value should be projected and an entry of -2 indicates that the value of the stage 2 cycles before should be projected.

An algorithm for computing the *refinement-map* matrix is given in Procedure 3. The algorithm enumerates over the rows of *etfd*. We describe the algorithm using row 11 of *etfd* shown in Table 2.2. For this circuit, *reference-point*=0 (the program counter is chosen as the reference point). t_p is the token value corresponding to the *reference-point* in *etfd*. For *elas-state*=11, t_p=21, which is the first valid token (when searching upward) in the *pc* column of *etfd*. Next, we find the synchronous token state (*sync-state*) corresponding to elastic token state 11 by searching the *reference-point* in *stfd* for t_p. A match is found for *sync-state*=5. Then we compute *refinement-map*[11] by searching backward from row 11 of *etfd* for a match with tokens in each stage of *sync-state*=5. *refinement-map*[11] = [-1, -2, -3, -3, -3]. Using the *refinement-map* matrix, proof obligations are generated for each elastic controller state as described next.

We choose a stage of the elastic design as a reference point (*reference-point*) such that the procedure for finding a refinement map will always complete successfully. Note that *reference-point* should be a stage that has a counterpart in the synchronous circuit. Therefore, *reference-point* has to be a non-additional EB. For example, *FD* can be an *reference-point*, whereas *L1* cannot. To find a suitable *reference-point*, we define the following graph.

Definition 9. *$G_{elastic}(V, E)$ of an elastic circuit is a directed graph, such that vertices (V) are the EBs of the circuit and the edges (E) correspond to data channels directed opposite to the direction of data flow.*

For example, if there is a data channel from EB_i to EB_j then there is an edge from V_j to V_i in the graph.

Property 3. *Every cycle in graph $G_{elastic}(V, E)$ includes at least one EB node, which is a non-additional buffer.*

The above property is a result of the elastic design process that incorporates additional EBs only to break paths between two non-additional EBs. The elastic design/synthesis process does not create cycles of additional EBs [8].

Definition 10. *$T_{elastic}$ is the set of directed graphs obtained by removing edges from $G_{elastic}(V, E)$ that are incoming to all non-additional elastic buffers.*

Lemma 1. *Every directed graph in set $T_{elastic}$ is a tree with a non-additional EB as the root.*

Lemma 2. $|T_{elastic}| = |S_y|$.

The above lemmas derive from Property 3. The lemmas indicate that $T_{elastic}$ contains one tree each for every non-additional EB.

Definition 11. *Latency of EB i of an elastic circuit is the depth of the tree $\in T_{elastic}$ with root i.*

Definition 12. *reference-point of an elastic circuit is the non-additional EB with the greatest latency in the circuit.*

Note that the *reference-point* need not be unique. Any EB with the greatest latency can be chosen as the *reference-point*.

Theorem 1. *If S is a closed synchronous circuit and E is the elastic implementation of S obtained using the SEN approach [8][23], then Procedure 3 will complete for any such S and E.*

Proof. Procedure 3 can be analyzed in 3 parts. In the first part (lines 2-6), the procedure searches backward starting from row *elas-state* of *etfd* for a non-zero token value in the column corresponding to the slave of the *reference-point*. For every row of *etfd*, this search will complete successfully because in the reset state (row 0 of *etfd*) of the elastic circuit, slave EHBs of every non-additional EB is initialized with a non-zero token value. t_p is the token-value found as the result of this search.

In the second part (lines 8-10), the procedure searches for t_p in column *reference-point* of *stfd* starting from row 0. The search will complete due to Property 2, which states that the sequence of tokens in the slave column (stage) of *etfd* and the corresponding column of *stfd* are identical modulo bubbles and duplicated values.

In the third part (lines 11-17), the procedure searches backward in *etfd* starting from row *elas-state* for every token in row *sync-state* of *stfd*. Note that the searches are done in corresponding columns of *etfd* and *stfd*. The search in each column will complete because the reference point was chosen as the point of synchronization. From Definition 12, the reference point has the greatest latency, so its progress is the slowest. Therefore, other columns (corresponding to other stages in the design) would have progressed faster and hence would have generated the tokens in row *sync-state*. Therefore, searching backward, each of the searches will complete. □

2.4. Tool Flow

The overall tool flow is shown in Figure 2.2. The equivalence checker takes as input the elastic and synchronous circuits in RTL VHDL and generates the verification proof obligations in SMT-LIB [26] format. The SMT logic used is QF_ABV, which is the logic of closed quantifier-free formulas over the theory of bitvectors and bitvector arrays. The proof obligations are then discharged using an SMT solver. The front end of the tool uses Verific Design Automation's parser platform [27] to parse the RTL VHDL input circuits to an internal representation. Also as input as meta data, the list of EBs/registers and their interconnection information is required for elastic/synchronous input circuits. This information is used to generate the connectivity matrix M_C for both circuits, which are then used to construct the token flow diagrams for both circuits as described in Sections 2.3.1 and 2.3.3. Also, the tool translates the internal representation from the RTL VHDL input circuits into SMT-LIB functions that implement the transition relation corresponding to the circuits.

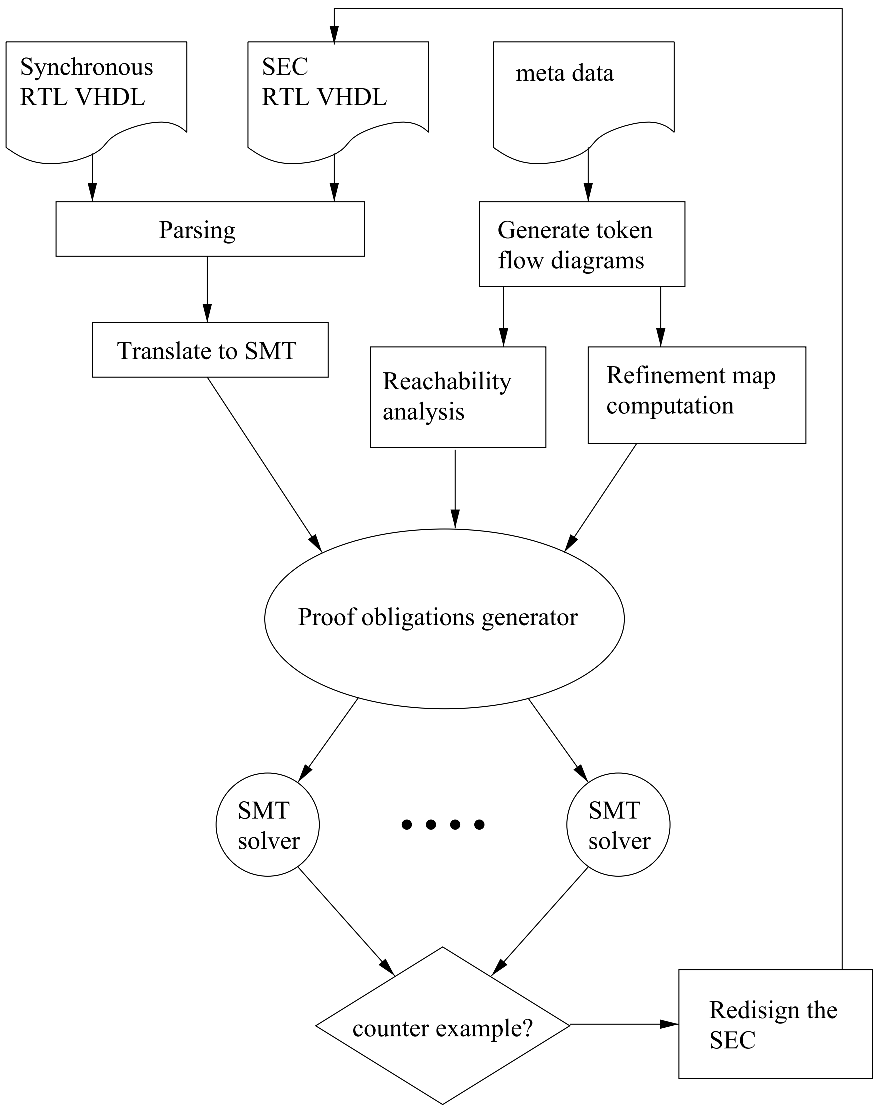


Figure 2.2. Tool Flow

From the token flow diagram of the elastic circuit, the reachable states of its elastic controller network are computed (as described in Section 2.3.2). The reachability analysis provides the reachable states and the transitions of the elastic controller network. Based on the reachability analysis, the tool generates invariant proof obligations to check that the elastic circuit satisfies these transitions. For example, if the E-2 circuit is in state 10, then it should transition to state 11 and if in state 11, it should transition to state 10. Note that states 10 and 12 are the same controller states, as they have the same distribution of tokens. The invariant proof obligation for the transition from state 10 to 11 is shown below.

$$\Big\{ \mathit{half}(PC_{10}) \wedge \mathit{empty}(FD_{10}) \wedge \mathit{half}(DE_{10}) \wedge \mathit{half}(EM_{10}) \wedge$$
$$\mathit{full}(MW_{10}) \wedge \mathit{empty}(L1_{10}) \wedge \mathit{half}(L2_{10}) \Big\} \longrightarrow$$
$$\Big\{ \mathit{empty}(PC_{11}) \wedge \mathit{half}(FD_{11}) \wedge \mathit{empty}(DE_{11}) \wedge \mathit{half}(EM_{11}) \wedge$$
$$\mathit{half}(MW_{11}) \wedge \mathit{half}(L1_{11}) \wedge \mathit{half}(L2_{11}) \Big\}$$

In the above, *half(x)*, *empty(x)*, and *full(x)*, indicate that the EB x is in a half state, empty state, and full state. FD_{10} is the value of EB FD in state 10, and similarly for others.

Next, the tool computes the refinement map by examining the token flow diagrams as described in Section 2.3.4. One refinement map function for each of the elastic controller network states is then generated in SMT. Using the refinement map and the transition relation functions in SMT, the proof obligations required for equivalence verification based on WEB refinement are generated (see Section 2.2). One proof obligation is generated for each transition based on the reachability analysis. Each of the proof obligations are generated in separate SMT files. Hence these obligations can be checked in parallel. If one or more of the proof obligations do not succeed, then the SMT solver will generate a counter example indicating one or more bugs.

2.4.1. Liveness

WEB refinement takes into consideration liveness. Establishing that an implementation refines its specification guarantees that the implementation will always progress and never deadlock w.r.t. the specification. Liveness is ensured using the mechanism of rank functions, functions that map the implementation (SEN) states to natural numbers. The goal is to devise a witness rank function such that for the stuttering steps of the implementation, the rank always decreases. There are no requirements of the rank of the implementation in non-stuttering steps, as the liveness of the implementation is witnessed in the fact that the implementation matches specification's progress in a non-stuttering step.

For the equivalence problem at hand, it is enough check that the implementation satisfies the invariants generated by the reachability analysis to guarantee liveness. To see why consider the following. The SEN framework is correct by construction and is guaranteed to be live. Therefore, every cycle in the reachability graph generated form the token flow diagrams should include at

Table 2.4. Results

Benchmark	No. of gates	No. of latches	Equivalence Checker Runtime (sec)	SMT statistics	
				Time (sec)	Memory (MB)
E-32-0	36,875	620	0.804	9.39	16.54
E-32-1	37,225	688	0.952	16.51	13.75
E-32-2	37,635	768	0.932	20.43	15.22
E-32-3	38,045	848	1.020	13.99	18.86
E-32-4	38,455	928	0.988	18.33	20.51
E-32-5	38,865	1,008	0.964	17.63	20.99
EB1-32-2	37,624	768	0.972	4.97	14.59
EB2-32-2	37,635	768	0.984	1.64	13.99
E-64-0	130,939	1,132	1.548	84.03	53.32
E-64-1	131,609	1,264	2.004	134.14	47.5
E-64-2	132,339	1,408	2.144	113.58	46.48
E-64-3	133,069	1,552	2.152	84.65	65.25
E-64-4	133,799	1,696	2.160	139.35	63.81
E-64-5	134,529	1,840	2.184	89.2	64.28
EB1-64-2	132,328	1,408	2.108	31.2	45.25
EB2-64-2	132,339	1,408	2.128	6.04	44.44
E-128-0	494,171	2,156	3.972	560.88	588.13
E-128-1	495,481	2,416	5.388	764.72	157.77
E-128-2	496,851	2,688	5.568	947.30	313.51
E-128-3	498,221	2,960	5.488	445.12	341.87
E-128-4	499,591	3,232	5.680	792.69	240.52
E-128-5	500,961	3,504	5.788	606.68	237.73
EB1-128-2	496,840	2,688	5.420	273.84	158.45
EB2-128-2	496,851	2,688	5.516	14.46	156.73

least one non-stuttering transition. Note that if all transitions in a cycle are stuttering, then this indicates deadlock. As long as the implementation satisfies the reachability invariants, it is guaranteed to never be in a stuttering cycle, guaranteeing liveness w.r.t. its specification. Also, if there are no stuttering cycles, its not hard to see that a rank function can be devised such that the rank decreases for every stuttering step.

2.5. Results

The capacity of the equivalence checker is demonstrated using 24 elastic circuits. The circuits are based on the elastic 5-stage processor shown in Figure 2.1. The circuits were obtained by varying the size of the datapath and the number and location of additional buffers/relay stations in the data path. A maximum of 5 additional buffers were used. Verification statistics are shown in Table 2.4. The benchmark names are of the form "E-m-n" or "EB-m-n". "E" indicates that the circuit was proved correct and "EB" indicates a buggy circuit. "m" indicates the size of the data path and "n" is the number of additional buffers in the elastic circuit. We are not aware of any other equivalence checker that can verify the equivalence of elastic circuits and their synchronous parents. Therefore, we do not have another tool to quantitatively compare the efficiency of our results.

To demonstrate the results when checking buggy versions of elastic circuits, we developed two buggy versions of the elastic processor. B1 has a data path bug in its forwarding logic. The address of source register 1 in the execute stage is compared with the destination register address of the memory stage, when instead the address of source register 2 should be compared. B2 has a bug in the elastic controller, where the controller for the *de* elastic buffer receives its valid input from the fetch stage instead of the decode stage.

Verification was performed on a 2.1GHz AMD (R) Athlon (TM) 2700+ CPU with a 256 KB L1 cache. The SMT solver used was Z3 [28]. The equivalence checker run time is the time taken to parse the input circuits and generate the proof obligations in SMT. The SMT time is the total time required to check all the proof obligations corresponding to the verification of the benchmark. The SMT memory is the maximum memory required for checking all the proof obligations of that benchmark.

2.6. Related Work

Carloni et al. [17] developed a theory for Latency-Insensitive design. In this work, they introduced latency equivalence, a notion of correctness that can be used to design latency insensitive systems in a correct-by-construction compositional manner. Li et al. [29] have used property-based verification to check latency equivalence, liveness, and storage capacity for three latency-insensitive designs. Suhaib et al. [30] have developed a general property-based and simulation-based validation framework that can be used with a large number of LI design methods. Their approach is also based on latency equivalence. Another approach to property-based verification of elastic systems is based on static data flow structures (SDFS) [31]. SDFS can be used to model synchronous and asynchronous elastic systems. The SDFS models are translated to petri-net models that are amenable to analysis and by model checking tools. In contrast, our contributions are in equivalence checking for elastic circuits. In general, simulation-based validation, property-based verification, and equivalence verification compliment each other very well as can be witnessed in commercial design cycles. Also, our equivalence framework does not require additional properties as the synchronous circuit is the specification.

Cortadella et al. [8] have verified the elastic controller implementations against a high-level specification that describes how these controllers should behave. Kristic et al. [23] have shown that additional buffers (empty buffers) can be inserted in the datapath without altering the functionality of the design. Synthesis approaches based on correct-by-construction transformations ensure that the synthesis methods are reliable. However as noted earlier, the synthesis process and any further modifications to the circuit can introduces bugs. The target of our equivalence verification framework is to catch these bugs.

Srinivasan et al. [32] have developed a refinement-based verification method for elastic circuits. They provide methods and rules to construct equivalence proofs between pipelined elastic circuits and their synchronous parents. Their proofs are constructed manually. Our equivalence checker builds on this work. Following are the novelty of our work, over and above what was presented in [32]. (1) We have formally defined token-flow diagrams and developed procedures to automatically derive token-flow diagrams for elastic circuits. (2) We have generalized the algorithms to be applicable to any circuit structure. In [32], the approach was applicable only to linear pipelines. (3) Generalization required formalizing the concept of reference-points and incorporation of the use of reference-points in the algorithm to compute refinement maps. (4) The computation of token-flow diagrams and refinement maps is fully automated and implemented in a tool, whereas previously all of this was done manually. (5) Completeness result for the algorithm that computes refinement maps is derived. (6) The tool has been successfully applied to elastic circuits with as many as 0.5M gates.

2.7. Conclusions

We have presented an automated equivalence checker that can verify elastic circuits against their synchronous parent circuits. The efficiency of the tool was demonstrated using 24 elastic VHDL benchmarks. The most complex elastic circuit verified has over 0.5**M** gates and over 3,500 latches. We believe that this equivalence checking technology can have a positive impact on the use of latency insensitive design in commercial design cycles, especially in the context of IP reuse to deal with timing issues.

There are several areas for future progress. Current algorithms are limited to dealing with deterministic elastic controller networks. We plan to extend the methods to deal with non-deterministic behavior of the elastic controllers as seen in open circuits, and circuits with variable latency units. Currently our applications are limited to closed circuits. Another area of future work is to explore the use of automated abstraction techniques to improve the scalability and capacity of the tool.

3. EQUIVALENCE VERIFICATION FOR NCL CIRCUITS[2]

3.1. Introduction

The synchronous design paradigm is facing many challenges as fabrication technologies scale down to keep up with the demand for high-performance low-power circuits. Such challenges include dominant wire delays that cannot be accurately determined until the later stages of the design cycle, and managing clock skew. In addition to these design challenges, high power consumption and noise are also inherent challenges in the synchronous design domain that are becoming increasingly difficult to manage. Asynchronous circuits in general have properties that could potentially solve many of the issues that are becoming dominant in the synchronous domain [3]. Therefore, in recent years, a lot of research work has been done to develop design methodologies and processes to integrate asynchronous circuits in commercial systems [9][7].

NULL Convention Logic (NCL) circuits are *delay-insensitive* (DI) asynchronous circuits. In comparison to synchronous circuits, DI circuits have lower power consumption, less noise, and lower electro-magnetic interference. The delay-insensitive nature of DI circuits can also be exploited to ease component reuse in complex SoC designs, especially for SoCs that employ multiple clocks [3]. DI asynchronous circuits are correct-by-construction designs that do not require extensive timing analysis as needed by *bounded-delay* asynchronous circuits (micropipelines). Also, performance of delay-insensitive asynchronous circuits is close to average case performance; whereas, performance of bounded-delay circuits is close to worst case [3].

Functional testing and verification for asynchronous circuits is a challenging problem, because the control components of asynchronous designs are highly non-deterministic and exhibit a

[2] The material in this chapter was co-authored by Vidura M. Wijayasekara, Sudarshan K. Srinivasan, and Scott C. Smith. Vidura Wijayasekara was the primary developer of the conclusions that are advanced here.Vidura Wijayasekara also drafted and revised all versions of this chapter. Sudarshan Srinivasan and Scott Smith served as proofreaders and checked correctness of the theories and algorithms developed by Vidura Wijayasekara. ©2014 IEEE. Reprinted, with permission, from V. M. Wijayasekara, S. K. Srinivasan, and S. C. Smith, "Equivalence Verification for NULL Convention Logic (NCL) Circuits," *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 10/2014.

prohibitively large state space. Therefore, exhaustive testing is very hard. Formal verification techniques have been successfully integrated in synchronous commercial design cycles and have shown to significantly improve functional coverage and find corner case bugs. In this paper, we propose a formal verification methodology for NCL circuits.

The proposed verification methodology is developed in the context of NCL synthesis. A recent trend to overcome the design challenges for asynchronous circuits (such as lack of CAD tools and design complexity) is to develop tools to synthesize asynchronous circuits from synchronous circuits. Such tools have been developed to automate the synthesis of NCL circuits from synchronous circuits [9]. The goal of our verification methodology is to check the equivalence of the synthesized NCL circuit against the synchronous circuit that was input to the synthesis tool, which we call the parent synchronous circuit. Why would such an equivalence verification methodology be useful in practice? Consider the integration of IP blocks in a multi-rate SoC environment. Such an integration in the synchronous domain is a challenging task. Instead, the designer may opt to use a technology in the DI design paradigm such as NCL circuits to ease integration of the multi-rate blocks.

After generating the NCL design, the resulting NCL circuit will have undergone a considerable transformation w.r.t. the original synchronous circuit. The designer cannot assume that the resulting NCL circuit will not have bugs. The designer will instead have to expend time and resources to verify the NCL circuit. Even though the NCL circuits are correct-by-construction, the synthesis tools that generate NCL circuits from the synchronous circuit may have bugs. Also, there maybe other sources of bugs such as manual tinkering of the NCL circuit after synthesis. The commercial design process will definitely require functional verification of the NCL circuit used in the SOC. *The full verification of the NCL circuit cannot be bypassed because it is correct-by-construction or by performing some checks on the connections of the NCL circuit.* Our equivalence verification methodology comes to the aid here as it is targeted at checking the correctness of the NCL circuit against the parent synchronous circuit. Equivalence checking technology has been very effective and useful in the synchronous domain.

The rest of the paper is organized as followed. First, we describe prior work related to verification of asynchronous circuits in Section 3.2. An introduction to NCL circuits is given in Section 3.3. Equivalence verification for combinational and sequential NCL circuits are described in Sections 3.4 and 3.5, respectively. Experimental results are presented in Section 3.6. Finally, we conclude in Section 3.7.

3.2. Related Work

Related work on verification technology for asynchronous circuits can be classified as property checking approaches [33][34] or methods based on trace theory [35]. The trace theory approaches target the verification of gate-level asynchronous circuits. In trace theory based approaches, the circuit is modeled as a petri-net. The correctness property is also modeled as a petri-net. In contrast, our equivalence verification approach is based on the theory of Well-Founded Equivalence Bisimulation (WEB) refinement. In WEB refinement, both the specification and implementation are modeled as transition systems. We are not aware of any trace theory based methods that have been developed for equivalence verification of synchronous and NCL circuits. Approaches based on property checking can be used for NCL circuits, but, are cumbersome because a large number of properties are required and also the properties themselves can be hard to write leading to erroneous specifications [35].

Loewenstein [36] verified some properties of a counter-flow pipeline using the HOL theorem prover. Counter-flow pipelines are asynchronous in nature with results flowing in the pipeline in a direction opposite to that of instruction flow. The NCL circuits we verify do not use the counter-flow mechanism. Also, our correctness proofs are based on the use of decision procedures and are highly automated.

Verbeek [37] verified deadlock freedom of DI circuits compiled from Click library (a DI primitives library) using SAT/SMT instances of the circuit. We verify NCL circuits, which is a different DI design paradigm from the technology implemented in the Click library. Also, we verify safety, i.e., we verify that the implementation (NCL circuit) behaves correctly as given by the specification (synchronous circuit).

A method to check the delay insensitive property of combinational NCL circuits is presented in [38]. Our method verifies the functional equivalence of both combinational and sequential NCL circuits against the parent synchronous circuit. A method to verify desynchronized pipelined circuits against ISA-type specifications using refinement is presented in [39]. Desynchronized circuits are bounded-delay circuits. In contrast, NCL circuits are delay-insensitive. Hence, their intrinsic properties are different and require domain-specific verification methods. Also, our specifications are the parent synchronous circuit used for synthesis of NCL circuits. We do not use an ISA-type specification. To our knowledge, there is no prior work that uses formal methods to verify the functional equivalence of NCL circuits against their synchronous parent circuits.

Cortadella et al. [40] have used flow equivalence (FE) to prove the correctness of their desynchronization method; and FE is well suited for this purpose. However, they have not demonstrated verification based on FE. Why do we use refinement instead of FE? Refinement is a more general notion. For example, one requirement of FE is that the specification and implementation should have the same set of registers. This requirement is not satisfied when comparing NCL circuits and their synchronous parents.

3.3. Background: NCL Circuits

NCL circuits are four phase (i.e., dual-rail) DI logic systems. Two rails or wires, D^0 and D^1, are used to represent a Boolean value. As shown in Table 3.1, D^0 and D^1 can take values DATA0, DATA1, or NULL. The two rails cannot be asserted at the same time, which is an illegal state. Values DATA0 and DATA1 correspond to Boolean zero and Boolean one, respectively. NULL means valid data is not available.

NCL has 27 threshold gates that implement all functions of four or fewer Boolean variables. A THmn gate, where m $\leq$ n, has n inputs and a threshold of m. When m or more inputs are asserted, the output of the gate is asserted. The output is deasserted only when all inputs are deasserted. A THnn gate is equivalent to an n-input C-element. A TH1n gate acts as an n-input OR gate. Data propagates through an NCL circuit as wavefronts alternating between DATA and NULL. The NULL wavefront (spacer) brings the output of all the threshold gates to logic zero in preparation for the next set of DATA inputs.

Table 3.1. Dual-Rail Signaling

	DATA0	DATA1	NULL	Illegal
D^0	1	0	0	1
D^1	0	1	0	1

Sequential circuits are implemented using NCL registers. A single-bit dual-rail NCL register is shown in Figure 3.1. The register receives K_i, a request signal, from a destination register as

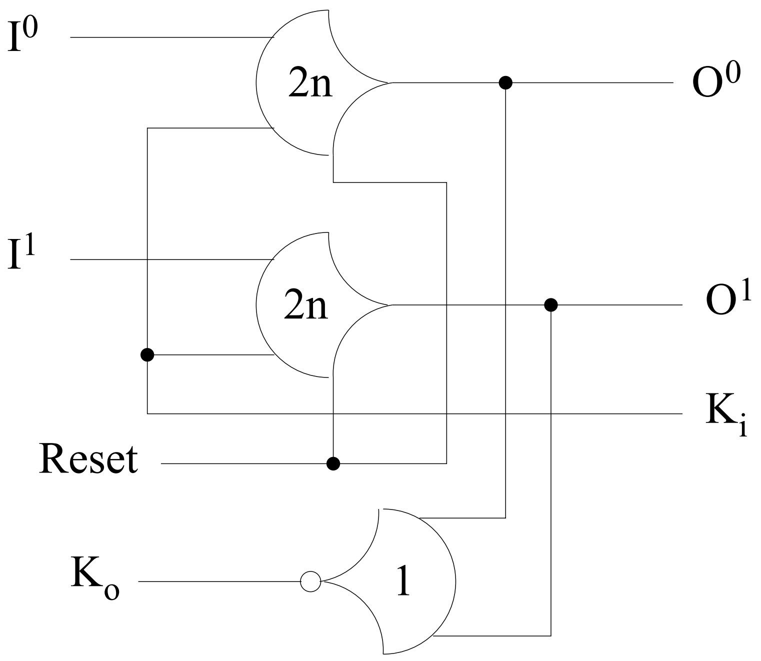


Figure 3.1. Single-Bit Dual-Rail Register

input. When K_i is asserted, DATA is requested, and when K_i is deasserted, NULL is requested. Each single-bit NCL register outputs K_o, an acknowledge signal. The K_o signals are combined in a completion component to form a single K_o signal, which is the K_i input to the source registers. When K_o is deasserted, the register acknowledges that DATA is received. When K_o is asserted, the register acknowledges that NULL is received. Completion components are implemented using THnn gates.

3.4. Equivalence Verification for Combinational NCL Circuits

We first describe the equivalence verification of combinational NCL circuits. Without loss of generality, consider an NCL circuit X with i dual-rail inputs, g NCL gates, and o dual-rail outputs. A subset of the gates form the dual-rail outputs of the circuit. Therefore, we have $2 * o \leq g$. x_1^{I0} ... ,x_i^{I0} denote the i D^0 inputs of X. x_1^{I1} ... ,x_i^{I1} denote the i D^1 inputs of X. x_1^{O0} ..., x_o^{O0} denote the o D^0 outputs of X. x_1^{O1} ..., x_o^{O1} denote the o D^1 outputs of X. x_1^G, ..., x_g^G denote the g gate values of X. An NCL circuit is in the NULL state if all gate outputs are deasserted, as given by the following equation.

$$\bigvee_{j=1}^{g} x_j^G = 0$$

If the inputs or outputs of the circuit have valid DATA, then for each dual-rail wire, the D^0 wire should be the negation of the D^1 wire. D^1 and D^0 both cannot be either 0 or 1. The condition that the inputs have valid DATA is given below:

$$\bigwedge_{j=1}^{i} x_j^{I0} \oplus x_j^{I1} \;= 1$$

NCL circuits are operated as follows. First, a NULL wave is propagated through the circuit, which deasserts all the inputs. That the outputs of all gates are deasserted when a NULL wavefront is propagated, is a pre-condition for the DATA wave to work correctly. The proof obligation to be checked to verify that an NCL circuit satisfies the above requirement is given below.

Proof Obligation 1.

$$\begin{aligned}\Big\langle &\forall x_1^{I0},...,x_i^{I0},x_1^{I1},...,x_i^{I1},x_1^{G},...,x_g^{G} \in \{0,1\} :: \\ &\quad \langle y_1^G,...,y_g^G\rangle = \mathit{nclstep}(x_1^{I0},...,x_i^{I0},x_1^{I1},...,x_i^{I1},x_1^{G},...,x_g^{G}) \\ &\quad \wedge \quad \neg(\bigvee_{j=1}^{i} x_j^{I0}) \quad \wedge \quad \neg(\bigvee_{j=1}^{i} x_j^{I1}) \\ &\Longrightarrow \; \neg(\bigvee_{j=1}^{g} y_j^{G})\Big\rangle\end{aligned}$$

In the above, *nclstep*() corresponds to a single step of the circuit and is modeled as a function that takes as input, the circuit inputs and the current state of the gates in the circuit. The *nclstep*() function outputs the values of the next state of the gates in the circuit.

A combinational NCL circuit is correct w.r.t. its synchronous counterpart, if for all combinations of valid data inputs, the outputs of both circuits are the same; however, inputs to latter are Boolean, and inputs to former are dual-rail. Same issue exists when checking the equality of outputs. To map valid dual-rail DATA to Boolean values, we exploit the fact that D^1 values represent the corresponding Boolean values, and D^0 values are always the negation of the corresponding D^1 values. Also, the NCL circuit operates under the assumption that the DATA wave is preceded by a NULL wave. Therefore, all the gate outputs in the circuit are deasserted. If *syncstep* corresponds to a single step of the synchronous circuit, the proof obligation below gives the equivalence correctness condition of a combinational NCL circuit against its combination synchronous counterpart.

Proof Obligation 2.

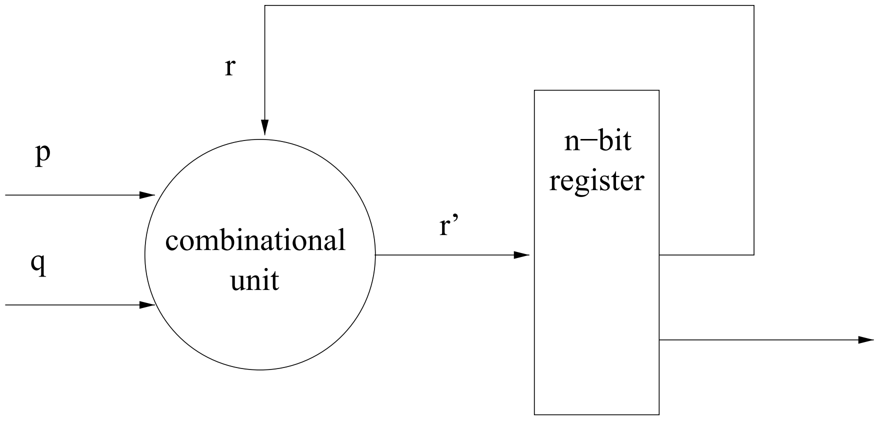
$$\Big\langle \forall x_1^{I0}, ..., x_i^{I0}, x_1^{I1}, ..., x_i^{I1}, x_1^{G}, ..., x_g^{G} \in \{0,1\} ::$$
$$\langle y_1^G, ..., y_g^G \rangle = \mathit{nclstep}(x_1^{I0}, ..., x_i^{I0}, x_1^{I1}, ..., x_i^{I1}, x_1^G, ..., x_g^G)$$
$$\wedge \quad \neg(\bigvee_{j=1}^{g} x_j^G) \quad \wedge \quad \bigwedge_{j=1}^{i} x_j^{I0} \oplus x_j^{I1}$$
$$\wedge \quad \langle z_1, ..., z_o \rangle = \mathit{syncstep}(x_1^{I1}, ..., x_i^{I1})$$
$$\Longrightarrow \quad \bigwedge_{j=1}^{o} z_j = y_j^{O1} \quad \wedge \quad \bigwedge_{j=1}^{o} y_j^{O0} \oplus y_j^{O1} \Big\rangle$$

Above, $z_1, ..., z_o$ represent the outputs of the synchronous circuit. Verification of proof obligations 1 and 2 guarantee the correctness of an NCL combinational circuit w.r.t. its synchronous counterpart. Verification is performed by modeling the circuits in the SMT2-LIB modeling language. An SMT solver can be used to check the proof obligations. We use the Z3 SMT solver for verification.

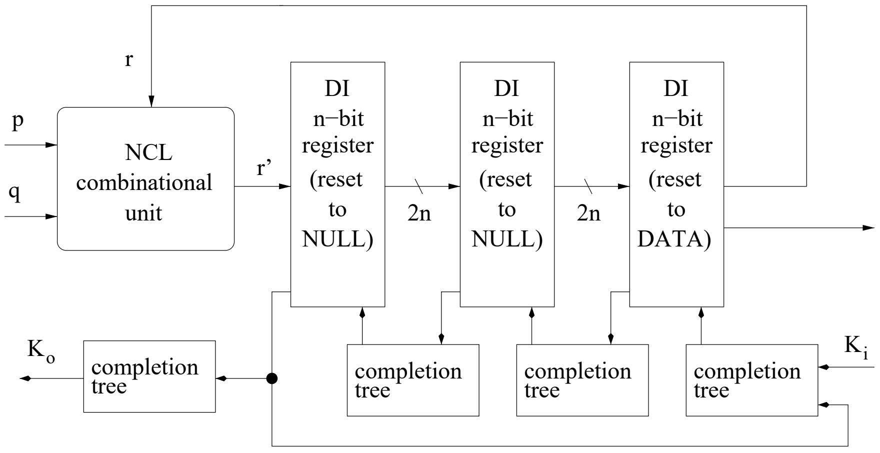
3.5. Equivalence Verification for Sequential NCL Circuits

In this section, we present a methodology for verification of sequential NCL circuits. The methodology is illustrated using the example shown in Figure 3.2. Fig 3.2.(a) is the circuit for a synchronous multiplier and accumulate (MAC) unit. The circuit takes two 4-bit inputs p and q, and computes $r' \leftarrow r + pq$. Fig 3.2.(b) shows an NCL version of the MAC unit, obtained by synthesizing the synchronous circuit in Fig 3.2.(a). Any feedback loop in an NCL circuit requires at least 3 NCL registers to ensure that the loop does not get deadlocked [3]. Other than the inclusion of two additional registers, the circuits look similar from the figure. However, there are many other differences. The data path (as mentioned earlier) is dual-rail in the NCL circuit. The MAC logic unit is implemented using threshold gates. Also, the registers in the synchronous circuit are clocked, whereas progress in the NCL circuit is achieved using the movement of alternate DATA and NULL waves that is achieved using completion logic communication between adjacent NCL registers.

There are many states the NCL circuit can be in and there is not a straightforward relationship between the synchronous circuit and the NCL circuit synthesized from it. Hence, we need a



(a) Synchronous


r
p
q
NCL combinational unit
r'
DI n−bit register (reset to NULL)
2n
DI n−bit register (reset to NULL)
2n
DI n−bit register (reset to DATA)
K_o
completion tree
completion tree
completion tree
completion tree
K_i

(b) NCL

Figure 3.2. Multiplier Accumulator Circuit (MAC)

general theory of equivalence to relate and verify sequential NCL circuits against their synchronous counterparts.

3.5.1. WEB Refinement

We use the notion of Well-Founded Equivalence Bisimulation (WEB) refinement for the equivalence verification problem. A formal and detailed description of WEB refinement is provided in [15][16]. Here, we provide a brief overview of the key features of WEB refinement relevant to the problem at hand. WEB refinement is a notion of equivalence that can be used to check if an implementation system satisfies its specification system, even if the implementation and specification are defined at very disparate levels of abstraction. We consider the NCL circuit to be verified as the implementation, and the synchronous parent circuit as the specification.

In the context of refinement, digital systems are modeled as transition systems (TSs). A TS is a three tuple and includes the set of states of the system, a transition relation that defines the state transitions of the system, and a labeling function that defines what is observable in each state. The behaviors of a system modeled as a TS are defined using the notion of paths. A *path* in a TS is a sequence of states, say s^0,s^1,s^2, ..., where s^0 is the initial state and s^0 transitions to s^1, s^1 transitions to s^2, and so on. An infinite sequence of such states is a *full path*. The behaviors of a system is the set of all full paths in the TS of the system.

The implementation behaves correctly as given by the specification, if every behavior of the implementation is matched by a behavior of the specification and vice versa. However, the implementation and specification may not have the same timing behavior. For example, if the implementation is an NCL circuit and the specification is a synchronous circuit, the NCL circuit may take many steps to match a single step of the synchronous circuit. This phenomenon is known as stuttering. To account for such situations, multiple but finite transitions of the implementation are allowed to match a single transition of the specification.

Another issue is that to check equivalence, synchronous states and NCL circuit states need to be compared. However, these states can look very different. While synchronous states use D-FF based registers, NCL circuit states use NCL registers that encode data in dual-rail format. Also, there need not be a direct mapping between the synchronous registers and the NCL registers. For the MAC example, the synchronous circuit has one register, while the NCL circuit has three registers. WEB refinement employs refinement maps, functions that map implementation states to specification states to bridge this abstraction gap.

Manolios [16] has shown that it is enough to prove the following refinement-based correctness formula to establish the equivalence of an implementation and specification based on WEB refinement. *rank* is used to distinguish stutter from deadlock. *rank* is a witness function from implementation states to natural numbers whose value decreases when the implementation stutters.

Definition 13. *(Refinement-Based Correctness Formula)*

$$\begin{aligned} \langle \forall w \in I :: \quad & s = \textit{refinement-map}(w) \wedge u = \textit{SStep}(s) \quad \wedge \\ & v = \textit{IStep}(w) \wedge u \neq \textit{refinement-map}(v) \\ \Rightarrow \quad & s = \textit{refinement-map}(v) \wedge \textit{rank}(v) < \textit{rank}(w) \rangle \end{aligned}$$

In the above, *SStep()* and *IStep()* are functions that define the transitions of the implementation and specification. I is the set of all the implementation states reachable from the initial states of the implementation. Note that in order to establish the equivalence of an implementation and specification system, all the reachable states of the implementation should be checked to see if they satisfy the above correctness formula. The correctness formula given above is expressible in a decidable fragment of first-order logic. Therefore, verifying equivalence of an implementation and a specification based on WEB refinement can be accomplished automatically using an SMT solver if a suitable refinement map and rank function are available.

3.5.2. Reachability for Sequential NCL Circuits

The first step in our verification methodology is to find the reachable states of the implementation NCL circuit. Registers in the NCL circuit are initialized to either hold DATA (D) or NULL (N) in a manner that ensures liveness of the circuit. For the MAC example, the registers $r_1r_2r_3$ are initialized to NND. Not every syntactically possible state of the three registers can be reached from the NND initial state. For example, states DDD and NNN will be never reached. We have developed a procedure to compute the reachable states of sequential NCL circuits. We require one assumption, that the NCL circuit has only one input source and one destination output. A large class of circuits fall under this category. We plan to tackle circuits with multiple input sources and destination outputs for future work.

Definition 14. *For an NCL circuit with n registers, an N/D state of the circuit is an n-tuple $\langle r_1, ..., r_n \rangle$, such that for 1$\leq l \leq n$, $r_l = N$ if register l is holding NULL and $r_l = D$ if register l is holding valid DATA.*

To compute reachable states of an NCL circuit, we define the notion of an N/D state of the circuit given in above Definition 14. Next, we develop a procedure that given an N/D state of a circuit, computes its next N/D state. This procedure is based on the notion of the connectivity matrix of an NCL circuit (Definition 15). In the definition below, R is the set of NCL registers in the circuit.

Definition 15. *The connectivity matrix of an NCL circuit* $M_C = [a_{lm}]_{|R|\times|R|}$ *such that* a_{lm} *= 1 if there is a data channel from register* l *to register* m*. Otherwise* a_{lm} *= 0.*

The following functions are used in Definition 16 that gives a function to compute the next N/D state of register l. For the functions, $r_{N/D}$ ∈{D,N}. If $r_{N/D}$ =D, then *src-status*(l,$r_{N/D}$) returns true only if all source registers to register l are in DATA state. If $r_{N/D}$ =N, then *src-status*(l,$r_{N/D}$) returns true only if all source registers to register l are in NULL state.

$$\textit{src-status}(l, r_{N/D}) = \bigwedge_{0<m\leq n} \Big\{ \big(M_c[m][l] = 1\big) \rightarrow \big(r_m = r_{N/D}\big) \Big\}$$

Similarly, *dst-status*(l,$r_{N/D}$) evaluates the status of the destination registers of register l. If $r_{N/D}$ =D, then *dst-status*(l,$r_{N/D}$) returns true only if all destinations of register l are in DATA state. If $r_{N/D}$ =N, then *src-status*(l,$r_{N/D}$) returns true only if all destinations of register l are in NULL state.

$$\textit{dst-status}(l, r_{N/D}) = \bigwedge_{0<m\leq n} \Big\{ \big(M_c[l][m] = 1\big) \rightarrow \big(r_m = r_{N/D}\big) \Big\}$$

Definition 16.

$$\textit{get-next-state}(l) = \begin{cases} D, & \Big\{ \big(r_l = N \wedge \textit{src-status}(l, D) \wedge \textit{dst-status}(l, N)\big) \bigvee \\ & \quad \big(r_l = D \wedge \neg\big[\textit{src-status}(l, N) \wedge \textit{dst-status}(l, D)\big]\big) \Big\} \\ N, & default \end{cases}$$

get-next-state(l) computes the next N/D state of register l. If the current N/D state of register l is NULL, all source registers to register l are in DATA state, and all destination registers of register l are in NULL state, then the next N/D state of register l is DATA. Otherwise, register l remains in NULL state in the next state. When the current N/D state of register l is DATA, the next N/D state of register l is NULL if source registers to register l are in NULL state and all the destination registers to register l are in DATA state. Else, register l remains DATA in the next state.

Procedure 4 Procedure to compute the next N/D settling state

```
1: procedure GetNextSettingState($\langle r_1, ..., r_n \rangle$)
2:     $settled \leftarrow false$
3:     $1\text{-}flipped \leftarrow false$
4:     $n\text{-}flipped \leftarrow false$
5:     while $settled = false$ do
6:         if $1\text{-}flipped = true$ then
7:             $r'_1 \leftarrow r_1$
8:         else
9:             $r'_1 \leftarrow GetNextState(1)$
10:        if $r'_1 \neq r_1$ then
11:            $1\text{-}flipped \leftarrow true$
12:        if $n\text{-}flipped = true$ then
13:            $r'_n \leftarrow r_n$
14:        else
15:            $r'_n \leftarrow GetNextState(n)$
16:        if $r'_n \neq r_n$ then
17:            $n\text{-}flipped \leftarrow true$
18:        for l $\leftarrow$ 2 to n-1 do
19:            $r'_l \leftarrow GetNextState(l)$
20:        if $\bigwedge_{l=1}^{n}(r_l = r'_l)$ then
21:            $settled \leftarrow true$
22:        else
23:            $\langle r_1, ..., r_n \rangle \leftarrow \langle r'_1, ..., r'_n \rangle$
```

An NCL circuit has transient states and settling states. Settling states are stable. An NCL circuit transitions from a settling state only when there is a change in the inputs to the circuit. Note that on the input side, an NCL circuit has a data input. On the output side, an NCL circuit has a K_i input. Before the circuit transitions from one settling state to another, it transitions through a series of states called transient states. These states occur temporarily and are unstable. Given an input N/D settling state, Procedure 4 computes the next N/D settling state. Without loss of generality, for a circuit with n registers, we assume that r_1 is connected to the data input to the circuit, and r_n is connected to K_i on the output side.

The construction of refinement maps depends on the number and distribution of unique DATA wavefronts in the reachable settling states of an NCL circuit. In the reset state, the number of DATA wavefronts is equal to stages in the corresponding synchronous circuit. However, this may not be true for all settling states. If the number of DATA wavefronts is greater or less than the number of stages in the synchronous circuit, it is harder to construct refinement mapping functions.

To address this issue, we introduce a design-for-verification technique that ensures that the number of DATA wavefronts in all settling states is an invariant and is equal to the number of DATA wavefronts in the reset state, which is again equal to the number of stages in the synchronous circuit. *The design-for-verification technique makes two additional connections in the circuit. The first is to connect the K_o of register r_1 to the input of the completion tree of register r_n. The second is to introduce a data channel from r_n to r_1.* The data channel can simply be introduced by connecting a bit in r_n that is always initialized to the value 0, to a bit in r_1. These additional connections ensures that a DATA wavefront can enter the circuit iff a DATA wavefront exits the circuit. Thus, the number of DATA wavefronts in the circuit always remains a constant. This design-for-verification technique essentially causes the circuit to behave as if there is a feedback loop from the last NCL register to the first NCL register. Procedure 4 exploits the property that the number of DATA wavefronts in the reachable states are invariant.

Procedure 4 uses three flags. The *settled* flag indicates if a settling state has been reached. Since r_1 is connected to the input data, its current value will be flipped (D to N or N to D) exactly once during the computation of the transient states and next settling state. Similarly, since r_n is connected to K_i, r_n value will also be flipped exactly once. *That r_1 and r_n are flipped exactly once when transitioning from one settling state to another is also a result of the design-for-verification technique. 1-flipped* and *n-flipped* keep track of whether r_1 and r_n have been flipped yet or not, respectively. All the flags are initialized to *false*. In the procedure, r_l indicates the current state and r'_l indicates the next state. The while loop (lines 5 to 23) is repeated until the next settling state is reached (*settled* is true). In the while loop, next state of r_1, the register that the input is connected to, remains unchanged if it has already been flipped once (*1-flipped* is true). Otherwise, the next N/D state of r_1 is computed using *get-next-state*() function (lines 6 to 9). If the next N/D state of r_1 is different from the current state of the register, *1-flipped* flag is set (lines 10 to 11). The next state of r_n is computed similarly using the *n-flipped* flag. The next N/D state of all other registers are computed using the *get-next-state*() function (lines 18 to 19). Once the next state of all the registers are computed, the next state of the NCL circuit is compared with the current state of the circuit. If the two states are equal, the *settled* flag is set causing the loop to terminate (lines 20 to 21). The next state at the loop termination is the next settling state. Else, the computed next state is set as the current state for the next loop run (lines 22 to 23). Using the procedure we get two settling states for the MAC example, NND and DDN. The transition from NND state to DDN state, transitions through the transient states DND and DNN. Similarly, transition from DDN to NND state transitions through NDN and NDD.

The reachable settling states and transitions between settling states form a transition system. For the MAC example, the transitions are NND to DDN and DDN to NND. We then check that the NCL circuit satisfies these transitions.

3.5.3. Refinement Maps

In this section, we derive a formula that can be used to construct refinement maps from NCL circuit states to parent synchronous circuit states. An NCL circuit is synthesized from an m-stage synchronous circuit by replacing the registers of the synchronous circuit with two or more NCL registers. Registers with self feedback loops should be replaced with a minimum of three NCL registers to avoid deadlocks. Hence, the n registers of the NCL circuit can be divided into m groups of two or more registers that correspond to the m stages of the synchronous pipeline. The last register in every group is initialized to DATA state at reset. Other NCL registers are initialized to NULL state. Therefore, there are m unique DATA wavefronts in the NCL circuit in the reset states. Also, our design-for-verification technique ensures then that every state of the NCL circuit reachable from reset state also has only m unique DATA wavefronts. Unique DATA wavefronts are separated from each other by NULL wavefronts (spacers).

We number registers such that r_2 is the register with input from r_1, r_3 is the register that has input from r_2 and so on. As such, we are assuming that the circuit has a linear pipeline structure with arbitrary feedback loops, but a large class of circuits can be cast into this structure. Register r_v holds a unique DATA wavefront if $r_v^D r_{v-1}^N$ is true, where for $1 < v \leq n$, r_v^D is a predicate that is true iff r_v is holding a DATA wavefront, and r_v^N is a predicate that is true iff r_v is holding a NULL wavefront. The DATA wavefront in r_1 is always unique.

Since the number of unique DATA fronts in the settling states of the NCL circuit is always equivalent to the number of registers in the parent synchronous circuit (m), the problem of finding a refinement map from the NCL circuit to the synchronous circuit reduces to finding the unique DATA wavefronts of the settling states of the NCL circuit and projecting them onto the stages of the synchronous circuit. Note that how the DATA wavefronts in NCL circuit states are projected depends on how the DATA wavefronts are distributed in that state, which is captured by the N/D settling states. As such each N/D settling state corresponds to a set of the NCL circuit states. The refinement map is constructed by having one projection function per N/D settling state. To construct such refinement maps, we introduce the notion of projection-predicates.

Definition 17. *A projection-predicate $p_{u \leftarrow v}$ is a predicate that is true only when register r_v of an NCL circuit state maps to stage u of the parent synchronous circuit.*

$$p_{u \leftarrow v} = \begin{cases} r_v^D r_{v-1}^N \left(\bigwedge_{k=1}^{u-1} \neg p_{k \leftarrow v} \cdot \bigwedge_{k=1}^{v-2} \neg p_{u \leftarrow k} \right) & , (1 \leq u < v \leq n) \\ r_1^D & , (v = 1 \wedge u = 1) \\ 0 & , default \end{cases}$$

In the above definition, the first condition for $p_{u \leftarrow v}$ to be true is that r_v should hold a unique DATA wavefront ($r_v^D r_{v-1}^N$). Then, we should determine if the unique DATA wavefront in r_v is the u^{th} unique DATA wavefront. Since the distribution of the DATA wavefronts in an NCL state is highly variable, it is hard to formulate an expression that would capture all possible combinations of DATA wavefront distributions with r_v holding the u^{th} unique DATA wavefront. We tackle this problem using an inductive approach to compute the projection-predicates using $p_{1 \leftarrow 1}$ as the base case. The rest of the projection-predicates are constructed inductively. $p_{1 \leftarrow 1}$ is true iff r_1^D. We provide two conditions in terms of projection predicates that if satisfied, the unique DATA wavefront in r_v is the u^{th} unique DATA wavefront. The first condition is that the DATA wavefront in r_v should not be mapping to any of the previous stages. The second condition is u^{th} stage of the synchronous circuit should not have a mapping from register in the NCL circuit before r_v.

We now provide the following formula for the refinement map, which gives the value of each register s_u of synchronous pipeline for a given NCL state, using the projection-predicates and projection functions.

$for\ 1 \leq u \leq m,$

$$s_u = \begin{cases} pf_{u \leftarrow 1}(r_1), & p_{i \leftarrow 1} \\ pf_{u \leftarrow 2}(r_2), & p_{i \leftarrow 2} \\ \vdots & \\ pf_{u \leftarrow n}(r_n), & p_{i \leftarrow n} \end{cases}$$

Projection function $pf_{u \leftarrow v}(r_v)$ projects register r_v of the NCL circuit to register u of the synchronous circuit. The value of the synchronous register is generated by extracting the value of the D^1 wires of the register r_v. Using the above refinement map we can instantiate the general web refinement theorem (Subsection 3.5.1) for equivalence verification problem of NCL circuits as follows.

$$\Big\langle \forall \langle r_1, ..., r_n \rangle \in \textit{reachable-states} :: \\ \bigwedge_{u=1}^{m} \big[s_u = pf(\langle r_1, ..., r_n \rangle, p_{u \leftarrow 1}, ..., p_{u \leftarrow n}) \big] \wedge \\ \langle s'_1, ..., s'_m \rangle = \textit{syncstep}(\langle s_1, ..., s_m \rangle) \wedge \\ \langle r'_1, ..., r'_n \rangle = \textit{nclstep}(\langle r_1, ..., r_n \rangle) \wedge \\ \neg \bigwedge_{u=1}^{m} \big[s'_u = pf(\langle r'_1, ..., r'_n \rangle, p'_{u \leftarrow 1}, ..., p'_{u \leftarrow n}) \big] \\ \Rightarrow \bigwedge_{u=1}^{m} \big[s_u = pf(\langle r'_1, ..., r'_n \rangle, p'_{u \leftarrow 1}, ..., p'_{u \leftarrow n}) \big] \Big\rangle$$

In the above proof obligation *reachable-states* is the set of reachable states of the NCL circuit found from the reachability procedure described in Subsection 3.5.2. *pf*() is the refinement map constructed from projection predicates and projection functions. *nclstep*() steps the NCL circuit to the next settling state. By proving the above proof obligation together with the reachability invariants mentioned in the Subsection 3.5.2, equivalence of the NCL circuit can be verified against its synchronous parent circuit using theory of WEB refinement. In this work, we consider only the safety part of WEB refinement, i.e., we only check that if the NCL circuit makes progress, that progress is correct w.r.t. to the specification (synchronous circuit). We do not verify liveness, which we plan to address in future work.

Table 3.2. Results

Model	No. of NCL gates	Time (s)	Memory (MB)
mul4X4	91	0.04	1.10
mul4X4-2s	279	0.32	2.94
mac4X4	202	2.77	6.68
mac5X5	281	10.63	8.12
mac6X6	370	96.17	16.59
mac7X7	472	788.93	24.83
mac8X8	587	13,527.12	47.97
mac8X8-B1	587	1.27	10.60
mac8X8-B2	587	4.00	13.63

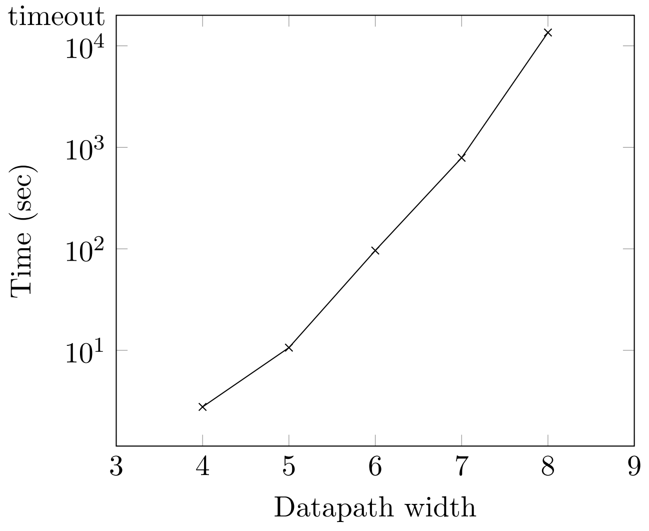


Figure 3.3. Verification Times for MAC Circuits

3.6. Results

The presented verification method was used to verify 9 NCL circuits including two buggy circuits. Verification was performed on a 1.86GHz Intel$^{\circledR}$ Celeron (R) CPU 540 with a 1 MB L2 cache. The SMT solver used was Z3 [28]. Results are summarized in Table 3.2. mul4X4 is a 4-bit combinational multiplier circuit. mul4X4-2s is a pipelined 4-bit multiplier circuit that has two stages. mac$n$Xn circuits are n-bit wide MAC units. Finally, mac8X8-B1 and mac8X8-B2 are buggy 8-bit MAC units. In mac8X8-B1, a bug was injected to the data path by connecting a wrong wire to the input of an adder. In mac8X8-B2, a bug was injected in the completion tree path of the circuit by connecting K_i^0, which is generated by the completion tree for r_0, to r_1 instead. The verification of NCL circuits is a complex problem. In fact, as the datapath width of the MAC circuit was increased from 4 to 8, the verification times increased exponentially as can be seen from the plot in Figure 3.3. The plot shows the time taken in seconds to complete the proof on a log scale against the datapath width.

3.7. Conclusions

Our contributions include a design-for-verification technique that reduces the state space of NCL circuits and ensures that the number of data tokens in reachable states of the circuit is an invariant. We exploit this property to develop a procedure to compute the reachable states of NCL circuits. We then use the reachable states to compute mapping functions or refinement maps for equivalence verification. The techniques were demonstrated using several NCL circuits. We found that for the MAC benchmarks, increase in the data path size causes verification times to increase exponentially. For future work, we plan to develop abstraction techniques to improve scalability of our equivalence verification approach.

4. ABSTRACTION TECHNIQUES TO IMPROVE SCALABILITY OF EQUIVALENCE VERIFICATION FOR NCL CIRCUITS[3]

4.1. Introduction

Asynchronous logic has been around for the past 50+ years; but, until recently, synchronous circuits have been good enough to meet industry needs, so asynchronous circuits were primarily utilized for niche markets and in the research domain. However, as transistor size continues to decrease, asynchronous circuits are being looked to by industry more and more to solve power dissipation and process variability issues associated with these smaller feature size circuits. NULL Convention Logic (NCL) is an asynchronous paradigm for the design of digital circuits and is characterized by dual-rail encoding (the use of two wires to encode Boolean data as opposed to just one wire) and the lack of need for a global clock. These features of NCL circuits have resulted in very desirable properties such as tolerance to high radiation exposure, tolerance to extreme temperature fluctuations, low power, less EMI, less noise, increased robustness, and design-reuse [41]. Owing to such properties, NCL circuits have carved out several niche application areas. NCL circuits have now been demonstrated to function without breaking down in space applications with high radiation exposure [42] and temperatures ranging from -196°C to 125°C [43, 44]. In System-on-Chip (SoC) design, NCL IPs can be placed in large multi-rate systems with minimal disruption to timing closure thus promoting IP reuse.

The design challenge for NCL circuits has been addressed to a large extent by the development of techniques and tools to synthesize NCL circuits from their synchronous counterparts [9, 45].

[3] The material in this chapter was co-authored by Vidura M. Wijayasekara, Anthony T. Rollie II, Ronald G. Hodges, Sudarshan K. Srinivasan, and Scott C. Smith. Vidura Wijayasekara was the primary developer of the conclusions that are advanced here.Vidura Wijayasekara also drafted and revised all versions of this chapter. Anthony Rollie II and Ronald Hodges served as undergraduate research assistants and contributed to the development of benchmarks. Sudarshan Srinivasan and Scott Smith served as proofreaders and checked correctness of the theories and algorithms developed by Vidura Wijayasekara.

Synthesis however does not guarantee that the resulting NCL circuits will be error free. There are numerous sources of bugs: bugs in the synthesis tools, manual tinkering of the circuit after synthesis, etc. To be employed in commercial designs, NCL circuits have to be verified. Due to the lack of global clock and high nondeterminism, verification of NCL circuits is a very challenging problem. The lack of verification technology will block the use of NCL circuits and the very fruitful exploitation of the unique properties of these circuits.

A very successful verification technology in the semiconductor industry is formal equivalence checking, where optimized circuits are checked for correctness against their original/parent counterparts. Wijayasekara et al. [46] developed the first known formal verification methodology to check the equivalence of NCL circuits against their synchronous counterparts. However, as will be shown in the results section, efficiency and scalability of verification was quite limited. In this work, we have developed two abstraction techniques to significantly improve the scalability of equivalence verification for NCL circuits. The abstraction techniques essentially break down the equivalence verification problem into smaller parts. The resolution of the smaller parts guarantee correctness of the entire circuit.

4.2. Abstraction of Combinational Units

In the equivalence verification of two circuits, say A and B, if there is a combinational circuit block that is identical in both circuits, the inner working of this combinational block can be abstracted, and hence, ignored in proving equivalence of A and B. The abstraction is achieved by replacing the combinational block in both circuits using an uninterpreted function (UF), which is like a black box function [47]. UF only satisfies the property of functional consistency: equal inputs produce the same outputs (given below).

$$\begin{aligned}&(x_1 = y_1) \wedge (x_2 = y_2) \wedge \ldots \wedge (x_n = y_n) \Rightarrow \\ &\qquad uf(x_1, x_2, \ldots, x_n) \quad = \quad uf(y_1, y_2, \ldots, y_n)\end{aligned}$$

This abstraction mechanism has been proven highly effective in verification of synchronous circuits. We have developed a technique to apply UF-based abstraction to the equivalence verification of NCL circuits with their synchronous counterparts. The challenge in the direct application of UF-based abstraction is that, while the functionality of a combinational block in an NCL circuit may be identical to the functionality of the corresponding block in the synchronous circuit, the implementation of the blocks in the two circuits will be very different.

There are three differences in the implementation of a combinational block in synchronous and NCL domains. First, NCL circuits use dual-rail encoding to represent values, as opposed to Boolean, which is used in synchronous. A one-bit value in synchronous will be represented using two wires in NCL: $\langle D^0, D^1 \rangle$. Boolean 0 and 1 are represented as $\langle 1, 0 \rangle$ and $\langle 0, 1 \rangle$, respectively. The combination $\langle 0, 0 \rangle$ is the NULL value, which represents absence of valid data; and $\langle 1, 1 \rangle$ is illegal, and will not occur in a properly operating circuit. This brings us to the second difference, which is that synchronization in NCL is achieved by alternatively propagating DATA and NULL wavefronts with handshaking, thereby eliminating the need for a global clock. In contrast, synchronous circuits always only propagate DATA and use a global clock for synchronization. Third, NCL circuits (both combinational and sequential elements) are constructed using threshold gates with hysteresis. The output of a three input threshold gate that has a threshold of two asserts when two or more inputs assert, and resets only when all inputs reset again. Hence, threshold gates have memory and the output depends on both inputs and current output. Moreover, NCL circuits need to be observable and input-complete. Input-completeness requires that at least one output avoid the transition $NULL \rightarrow DATA$ ($DATA \rightarrow NULL$) until all inputs are DATA(NULL). Observability is a property related to the internal gates of an NCL circuit: any change in the internal gate values should cause a change at the output. These constraints ensure delay-insensitivity.

Definition 18. *Let* $\langle y_1, y_2, ..., y_n \rangle = uf_{sync}(\langle {i_1}^1, {i_2}^1, ..., i_m^1 \rangle)$. $\langle Y_v^0, Y_v^1 \rangle$ *is*

(a) $\langle \neg y_v, y_v \rangle$ *if* $\bigwedge_{u=1}^{m} ({i_u}^0 \oplus {i_u}^1)$,

(b) NULL if $\bigwedge_{u=1}^{m} (\neg {i_u}^0 \wedge \neg {i_u}^1)$, *or*

(c) $\langle y_v^0, y_v^1 \rangle$ *otherwise,*

where $1 \leq v \leq n$.

We propose an NCL wrapper function (Definition 18), which is our novel contribution in this paper. The wrapper function (definition given above), as the name suggests, is to be wrapped around a synchronous function to enable that function to be embedded into an NCL environment.

The wrapper function transforms dual-rail data on the input side to boolean data, and vice versa on the output side, *while preserving all the properties of NCL, including handling NULL inputs and guaranteeing input-completeness.* Therefore, the wrapper function can be used to abstract a combinational circuit block in an NCL circuit using a synchronous UF.

In Definition 18, input to the wrapper function is $\langle {i_u}^0, {i_u}^1 : 1 \leq u \leq m \rangle$; current state of the output is $\langle {y_v}^0, {y_v}^1 : 1 \leq v \leq n \rangle$, and $\langle Y_v^0, Y_v^1 : 1 \leq v \leq n \rangle$ is the next state of the output. uf_{sync} is the UF symbol that abstracts away the corresponding synchronous combinational unit. Of dual-rail encoded signal $\langle D^0, D^1 \rangle$, D^0 and D^1 are mutually exclusive when transmitting DATA, and D^1 is equivalent to the Boolean value of DATA. Hence, $\langle {i_1}^1, {i_2}^1, ..., {i_m}^1 \rangle$ is forwarded as input to uf_{sync}, and dual-rail equivalent of an output y from uf_{sync} is simply $\langle \neg y, y \rangle$. If all dual-rail inputs to the wrapper function are DATA (all dual-rail inputs are mutually exclusive), dual-rail equivalent of output of uf_{sync} is forwarded to the NCL system. If all inputs are NULL, output is also set to NULL (required NCL behavior). For any other combination of inputs, output of the wrapper function is unchanged. Triggering changes at the output only when all inputs are DATA (or NULL) ensures input-completeness. Since uf_{sync} abstracts out all the internal logic from the combinational block, observability is irrelevant, and therefore ignored.

4.3. Abstraction of Completion Mechanism

In this section, we present another novel abstraction technique that we have developed, which is targeted at the NCL control logic. A single-bit NCL register has an output (k_o), which is used to make requests to registers in preceding stages. When the register output is NULL/DATA, k_o is 1/0. k_o is generated using an inverting TH12 threshold gate with the two output rails of a dual-rail register as inputs.

Definition 19. *A threshold gate THmn:*

- *has n inputs and a threshold $m : 1 \leq m \leq n$,*
- *if number of asserted inputs $\geq m$, then output is* 1,
- *if number of asserted inputs is zero, then output is* 0.

For an n-bit register, a completion detector k_c is generated, which is 1/0 when all the single-bit registers output NULL/DATA. Thus, the k_c signal is used to synchronize the completion of the registers.

Synthesizable threshold gates have an upper limit of four inputs [3]. Thus a tree of threshold gates, called a completion tree, is used to generate k_c. As the width of the register increases, the completion tree grows more complex.

We propose an abstraction technique for the generation of $k_c$where the tree for an n-bit register is replaced by a THn(2n) threshold gate with inversion at the output. The function of the gate is defined below.

Definition 20. *k_c is*

(a) 0 if $\sum_{j=1}^{n}(r_j^0 + r_j^1) = n$,

(b) 1 if $\sum_{j=1}^{n}(r_j^0 + r_j^1) = 0$, or

(c) k_c otherwise.

In the definition above, r_j^0 and r_j^1 are the dual-rail output of the j^{th} register. Case (a) corresponds to DATA completion. We check the sum of the asserted output rails ,$\sum_{j=1}^{n}(r_j^0 + r_j^1)$, to detect completion. Dual-rail output is DATA, when only one of r^0 and r^1 is 1. Therefore, sum is n when all registers have valid data. Here we exploit the fact that both rails of a dual-rail signal will not assert at the same time, which is also exploited in the completion detection circuit design. Case (b) corresponds to NULL completion: deassertion of all output rails. Hence, the sum should be zero. Case (c), the default case, corresponds to intermediate states between completions where the circuit holds the current output.

The completion tree abstraction is achieved by replacing the original completion tree circuit with the function from Definition 20. The use of this abstraction significantly reduces verification time as can be evidenced from the experimental results (given in the Results sections). The use of this abstraction has to be verified as well. To elaborate, a verification check is required that checks that the abstraction function from Definition 20 is equivalent to the completion tree from the original circuit that is replaced. This can be verified using techniques from [46].

4.4. Results

We used multiply and accumulate (MAC) circuits to demonstrate the effectiveness of the proposed abstraction techniques. Circuits involving multiplication are typically challenging, and therefore are good benchmarks to evaluate verification techniques. The benchmarks were obtained

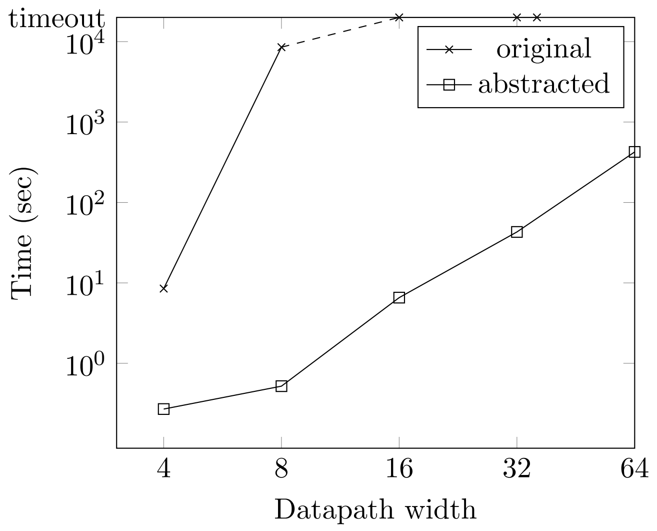


Figure 4.1. Comparison of Verification Times for MAC Circuits

by varying the datapath of the MAC circuits between 4-bits and 64-bits. The combinational multiplier circuit and the three completion trees in the MAC were abstracted away using our abstraction techniques. Combinational multiplier circuit can be verified independently using equivalence verification method for combinational circuits [46]. The benchmarks and verification proof obligations were specified using the SMT-LIB language [26] and were checked using the Yices Solver version 2.4.2 [48]. All experiments were performed on a 1.8 GHz AMD A6-6310 APU with a 2MB L2 cache.

The results are given in Table 4.1. We used a timeout of 20,000 seconds. As can be seen from the table, equivalence verification of the original MAC circuits times-out beyond the 8-bit datapath. Column "abstracted circuit" gives the verification times for the MAC circuits after applying combinational abstraction and completion tree abstraction. The "combinational abstraction" column gives the verification times to check the equivalence of the combinational units that were abstracted from the NCL and synchronous circuits. The "completion tree abstraction" column shows the verification times required to check the equivalence of the original completion tree from the circuit and the completion tree abstraction function (from Definition 20) that was applied to replace the original completion tree circuit. Results indicate significant improvements in the verification times. With the use of abstractions, equivalence verification could be extended up to and beyond a 64-bit MAC. Verification times of the original and abstracted circuits are plotted against data path width in Figure 4.1.

Table 4.1. Verification Times for MAC Before and After Abstractions

Model	time (s)			
	full circuit	pipeline	comb. unit	cmpl. tree
MAC4x4	8.501	0.167	0.1	N/A
MAC8x8	8528.824	0.184	0.292	0.046
MAC16x16	timeout	0.300	6.217	0.044
MAC32x32	timeout	0.660	42.106	0.042
MAC64x64	timeout	3.251	421.364	0.051

4.5. Conclusion

Based on the results, we conclude that the proposed abstractions are necessary for equivalence verification of complex NCL circuits with their synchronous counterparts.

5. EQUIVALENCE VERIFICATION FOR DESYNCHRONIZED CIRCUITS[4]

5.1. Introduction

The impact of persistent technology scaling results in a previously ignored set of design challenges such as manufacturing and process variability, and increased significance of wire delays. The challenges threaten to invalidate the effectiveness of synchronous design paradigms at the system-level. Asynchronous circuits possess several alluring properties—over their synchronous counterparts—that pose solutions to many of these challenges. Such properties include locally generated timing signals in the place of global clocks, potential performance speedups, robustness towards variability in the manufacturing process and operating conditions, etc. [40, 49, 50, 51, 52, 35, 53, 54, 55, 56]. However, design of asynchronous circuits has been a challenge and currently lacks support of Computer-Aided Design tools. Desynchronization [40, 57] is proposed as a design solution, where pipelined circuits and systems with a high degree of asynchronocity are synthesized from synchronous parents in a manner that exploits existing CAD tool support for synchronous designs. Desynchronization methods have in fact been successfully used to design and fabricate circuits that implement the DLX architecture and the DES encryption/decryption algorithm [40].

For desynchronization to be a feasible design solution, one of the critical challenges however is verification. Verification becomes a challenge when the desynchronized circuits are pipelined. For example, the controller of a desynchronized 5-stage pipeline can have more than $16*15^{10}$ states [40]. Functional verification using exhaustive testing is, therefore, hard. In this paper, we present a methodology for formal equivalence verification for desynchronized circuits. Equivalence verification has been a very successful approach in commercial design cycles for synchronous design.

[4] The material in this chapter was co-authored by Vidura M. Wijayasekara, Sudarshan K. Srinivasan, and Raj S. Katti. Vidura Wijayasekara was the primary developer of the conclusions that are advanced here.Vidura Wijayasekara also drafted and revised all versions of this chapter. Sudarshan Srinivasan and Raj Katti served as proofreaders and checked correctness of the theories and algorithms developed by Vidura Wijayasekara.

The methodology is developed in the context of desynchronization synthesis. The desynchronization paradigm was developed with the goal of synthesizing asynchronous circuits from synchronous circuits. The target of our equivalence verification framework is to check equivalence of desynchronized circuits and their parent synchronous circuits. We call the synchronous circuit that is used as input to desynchronization synthesis as the parent synchronous circuit. Such equivalence verification technology is new. We are not aware of any previous work that has developed such an equivalence verification framework. We also note here that such an equivalence verification methodology will positively impact the feasibility of using the desynchronization framework in commercial design cycles.

The transformations used in desynchronization to synthesize asynchronous circuits from synchronous circuits have been shown to be correct by construction. Why is then equivalence verification required? The parent synchronous circuit will have undergone significant mutations and there can be many sources of bugs that can creep into the synthesis process such as buggy synthesis tools and manual tinkering of the synthesized circuit. Commercial design cycles will require to verify the functional correctness of the asynchronous implementation, which could be a prohibitively expensive task. Partial verification methods such as checking if the connections in the desynchronized controller network are correct, will be insufficient to satisfy the stringent requirements of commercial design. Our equivalence verification methodology addresses this gap.

The notion of equivalence we use is Well-Founded Equivalence Bisimulation (WEB) refinement, which can be thought of as a notion of equivalence between two digital systems (a specification and an implementation). WEB refinement can be used even if there is a significant abstraction gap between the specification and implementation. There are two problems when using WEB refinement. First is finding the states of the desynchronized circuit that can be reached from the initial states of the circuit (reachable states). Identifying reachable states is important as unreachable states are often inconsistent and can cause spurious counter examples. The second problem is finding a refinement map, which is a function that maps reachable state of the implementation onto specification states. To elaborate on the challenges of the first problem: the desynchronized pipeline controller network is highly non-deterministic in nature and also exhibits a large state space. These two factors make it hard to trace the reachable states. There are two approaches to compute reachable states of the desynchronized controller network.

1. The first approach is based on symbolic simulation. The idea is to start from the set of reset states and perform symbolic simulation until no new states are discovered, *i.e.*, until a fixed point is reached. Or, start from the set of all states and perform symbolic simulation until a fixed point is reached where no new states are eliminated [58]. Approaches based on symbolic simulation of the implementation model cannot be used because the complexity of the desynchronized controller network requires a prohibitively large number of symbolic simulations of the model.

2. The second approach is to compute invariant properties of the desynchronized controller network that characterize the set of reachable states. We explored this approach and found that because of the large state space and the high degree of non-determinism of the controller networks, we could not find a systematic approach to generate invariants to characterize the reachable states of the controller networks. The primary problem is that the behavior of a desynchronized controller depends on the state of controllers on its output side, but, does not have any dependencies on the input side *i.e.*, the source controllers. This lack of dependency on the source side results in a high degree of nondeterminism of the resulting controller networks.

The second problem in equivalence verification is finding a suitable refinement map function. The challenge here is to find a general formulation for generating refinement maps that can be used for a large class of desynchronized circuits. Another requirement is that the formulation generates refinement maps that can be encoded in a decidable fragment of first-order logic. Such a formulation would provide a platform for highly automated equivalence verification that can exploit SMT solvers to discharge WEB refinement proof obligations.

Since verifiability is an important consideration for design, we propose changes to the desynchronized controllers introduced by Cortadella et al. [40] that would make equivalence verification a feasible solution by alleviating the aforementioned challenges. These changes add additional sequential dependencies between controller events so that the state space of each controller and resulting controller networks are simplified and reduced. Changes are two fold. First, all controllers in the controller network are replaced with the modified controllers, which we refer to as the Design For Verification Desynchronization (DFVD) controllers. Second, the controller network is modified, which we refer to as a *circular desynchronized pipeline.* We show that when desynchronized pipelines are modified to circular desynchronized pipelines and DFVD controllers are used, the resulting desynchronized pipelines become amenable to our verification approach. The specific contributions of our work are:

1. The controller used for desynchronization can hold zero, one or two tokens. The controller is initialized with one token and can transition to states with zero tokens or two tokens. Our contribution is the DFVD controller that satisfies the following property. If the controller currently holds one token, then the controller will hold on to that token until a new token is accepted on its input. This is not a property satisfied by the original controller used for desynchronization. Therefore, when the DFVD controller is initialized with one token, it will always remain in states where it has one or two tokens. This property of the DFVD controller makes it possible to compute reachable states of pipeline controller networks, which is a requirement for equivalence verification. However, the verifiability of the controller is achieved by trading with performance. We estimate that in the worst case, pipeline throughput is degraded by the delay of four transitions of a muller-C element.

2. We analyzed and deduced 15 properties of the DFVD controller. These properties (an important contribution of our work) can be used as rules and applied to systematically generate invariants and characterize the reachable states of any DFVD controller network.

3. We introduce design for verification: *circular* desynchronized pipeline systems. A desynchronized pipeline system with n stages can hold anywhere between 1 to $2n$ valid *unique* tokens, which correspond to information in the datapath that are not redundant, in a given state. The number of valid unique tokens in circular desynchronized pipeline states, on the other hand, remains constant. Therefore, when an n stage circular desynchronized pipeline system is initialized with one unique token in each DFVD controller (totaling to n unique tokens), the number of unique tokens in every reachable state of the system will always be n. This property makes it possible to build a simplified general refinement map function between the reachable states of the circular desynchronized pipeline and the states of the specification synchronous pipeline, which, also, always has n unique tokens.

4. We have also developed a equivalence verification procedure for circular desynchronized pipelined systems. Proving equivalence requires a refinement map, which (in this context) is a function that maps states of the implementation (desynchronized pipelined system) to states of the specification (pipelined synchronous machine). Defining the refinement map in this context requires identifying redundant information in the pipeline (which is possible in desynchronized pipelines). Our specific contribution here is to identify conditions of the controller network state that correspond to redundant information in the data path. These conditions are generic and can be applied to any DFVD controller network. The key here is that these conditions are applicable only for reachable states (which we have been able to characterize using the DFVD properties).

5. We developed 7 circular desynchronized processor models of varying pipeline length (between 5 and 7 stages) and controller complexity. The models used the DFVD controllers, and the controller networks were modified to circular pipelines. Our design for verification and verification approaches are demonstrated by checking the correctness of these models.

The rest of the paper is organized as follows. Related work is given in Section 5.2. Desynchronization and DFVD controllers are described in Section 5.3. The properties of the DFVD controllers and reachability analysis of desynchronization controller networks are described in Section 5.4. The desynchronized processor models used for experiments are described in Section 5.5. The equivalence verification procedure is detailed in Section 5.6. Circular desynchronized pipelines are described in Section 5.7. Computation of the refinement map function for circular desynchronized pipelines and required invarients are detailed in Section 5.8. Experimental results are given in Section 5.9, and we conclude in Section 5.10.

5.2. Related Work

Current verification technology for asynchronous circuits can be classified as property checking approaches [33][34] or methods based on trace theory [35]. Trace theory was originally developed to model and verify concurrent systems. In trace theory the implementation and the specification are modeled as trace structures. Later on the theory was adapted by asynchronous community to verify untimed models of asynchronous circuits. The drawback in using untimed models is that they force the designer to use more complex implementations for bounded delay asynchronous circuits (e.g. desynchronized circuits) than necessary [59]. Later developments of the trace theory sought to resolve this drawback and handle real-time constraints of bounded delay circuits by extending the theory to handle timed traces and using time petri-nets to model bounded delay circuits. While research has been done to prove the correctness and improve efficiency of these models, many trace theory approaches target the verification of gate-level asynchronous circuits. Our focus is on the verification of desynchronized pipelined circuits and systems that are described in RTL. Our approach, which is equivalence verification, is based on the theory of Well-Founded Equivalence Bisimulation (WEB), in which implementation and specification are modeled as transition systems. Verifying pipelines has been a challenge and warrants specialized techniques. Approaches based on property checking can be used for desynchronized pipelined circuits, but, are cumbersome because a large number of properties are required and also the properties themselves can be hard to write leading to erroneous specifications [35].

Loewenstein [36] verified some properties of a counter-flow pipeline using the HOL theorem prover. Counter-flow pipelines are asynchronous in nature with results flowing in the pipeline in a direction opposite to that of instruction flow. The desynchronized pipelines we verify do not use the counter-flow mechanism. Also, our correctness proofs are based on the use of decision procedures and are highly automated.

Verbeek [60] verified deadlock freedom of delay insensitive circuits composed of primitives from the Click library. Click library is a collection of handshake circuit elements for implementation of data-driven asynchronous circuits. Verbeek uses SAT/SMT instances of the circuit to verify the deadlock freedom. In this regard, Verbeek's approach is similar to ours in that we also use SMT to model the transition systems. However, we prove funtional equivalence between a desynchronized pipeline and its specification pipelined synchronous system.

Cortadella et al. [40] have used flow equivalence (FE) to prove the correctness of their desynchronization method, and FE is well suited for this purpose. However, they have not demonstrated verification based on FE. Why do we use refinement instead of FE? Refinement is a more general notion. For example, one requirement of FE is that the specification and implementation should have the same set of latches. If optimizations such as retiming or pipelining is applied after desynchronization, then the design cannot be related back to its synchronous specification using FE. This is because FE compare the values in latches to verify equivalence.

In previous work, we developed a refinement-based verification method for desynchronized pipelines [61]. The original desynchronization controllers were used. The approach for reachability is based on performing symbolic simulation of the implementation model, starting from reset states, until no new states are discovered, *i.e.*, until a fixed point is reached. However, this approach is not viable because the complexity of the desynchronized controller network requires a prohibitively large number of symbolic simulations of the model. As a result, we were only able to verify a small subset of the reachable states, and, therefore, the verification method is only partial. In the current work, our approach is to generate constraints (also known as inductive invariants) on the state variables that characterize the reachable states of the system. This approach is also not viable for the original desynchronization controllers. Hence we use the design for verification approach to develop the DFVD controller. For the DFVD controller network, the latter approach for reachability is viable as shown in Section 5.4.

Also, we verify the desynchronized pipeline (implementation) against its parent circuit: the pipelined synchronous circuit (specification). In our previous work [61], we verified the desynchronized pipeline against the non-pipelined synchronous specification. Verifying against the pipelined synchronous specification is a better approach because the desynchronization paradigm was developed with the goal to use desynchronization as a way of synthesizing asynchronous circuits from synchronous circuits. It is impossible to create refinement maps when there is so much variability in the number of valid unique tokens in a given state of the desynchronized pipeline. In our current work, we introduce circular desynchronized pipelines that enables generating the refinement maps from the desynchronized pipeline states to the pipelined synchronous specification states.

We proposed the idea of using completion functions to define the refinement map for desynchronized pipelines in [61]. We adopt this approach in our current work. However, the key difference is how redundant data is identified in the pipeline. In [61], we relied on observing the flow of tokens in the controller network, as symbolic simulation was used for reachability analysis. However, as stated earlier, using symbolic simulation for reachability analysis is not viable. In this work, we have deduced generic conditions of the state of controller network that identify redundancy in the data path, which can be used for complete safety verification and is described in Section 5.7.

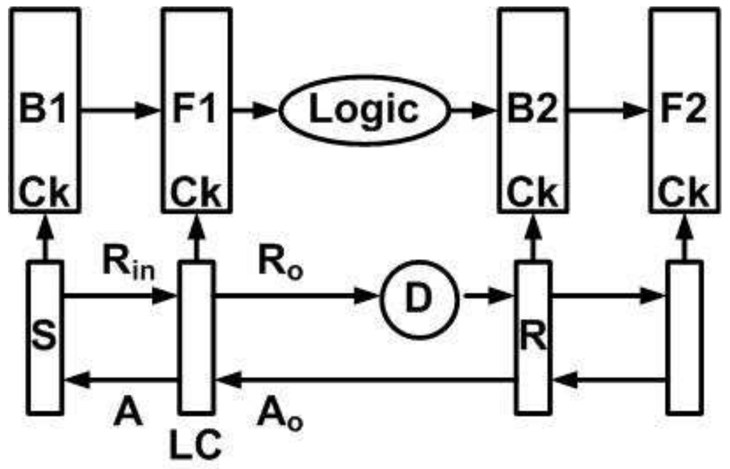


Figure 5.1. A Pipeline Stage with a Latch Controller Network.

5.3. Desynchronization and DFVD Controllers

Desynchronization is the process of converting a synchronous circuit into an asynchronous one by replacing the clock network with a network of handshaking latch controllers. The edge-triggered D-flipflops of the synchronous circuit are replaced by two D-latches which are transparent when their clock input is a 1 and are in the hold mode otherwise. The clock signals or triggers (Ck) for the latches are obtained by latch controllers with two inputs (R_{in}, A_o), and two outputs ($R_o, A = \neg Ck$). $R's$ denote a request signal and $A's$ denote an acknowledge signal. Consider a synchronous pipeline stage with a logic block whose inputs are provided by a flipflop and whose outputs are input to another flipflop. A desynchronized version of such a stage is shown in Figure 5.1. Each flipflop is converted into two latches shown on either side of the logic block. In a pair of consecutive latches, the left latch is the back latch and the right is the front latch, indicated by subscripts "b" (or "B") and "f" (or "F"), respectively. Each latch has a controller associated with it. The latch controller used by us is the semi-decoupled controller in [57]. If G is a signal then G^+ corresponds to a rising edge on G and G^- corresponds to a falling edge on G. We now describe the operation of the latch controller labeled, LC, in Figure 5.1 with the help of the two latch controllers , S (for sender), and R (for receiver). LC receives R_{in}^+ from S indicating the availability of data at the input of the LC latch (F1). LC sends A^+ to S indicating that the data has been captured by F1. LC then sends R_o^+ to R to indicate that its output is valid and will be stable until A_o^+ is received. S sees A^+ and puts out R_{in}^-. LC sees R_{in}^- and A_o^+ and puts out A^- and R_o^-. R puts out A_o^- when it receives R_o^-. The above description of the latch controller (called the 4-phase controller) can be converted to the following five logic equations.

1. $A^+ = R_{in} \wedge \neg R_o$
2. $A^- = \neg R_{in} \wedge R_o \wedge A_o$
3. $R_o^+ = A \wedge \neg A_o$
4. $R_o^- = \neg A$
5. $Ck = \neg A$

Note that the clock input to the latch is Ck, and the delay element D in Figure 5.1 mimics the delay of the logic block. This kind of desynchronization is similar to that performed in [40] and has been proven to work well.

One of the important aspects of desynchronization is that it leads to redundant tokens in the data path. When a copy of the token is passed from a source latch to a destination latch, the source latch holds on to the token until it receives an acknowledge signal indicating that the destination latch has accepted the token. Thus, for a period of time, the source is holiding on to a redundant token that has reached the destination. Redundancy leads to some issues with refinement-based verification, which we discuss in Section 5.6.

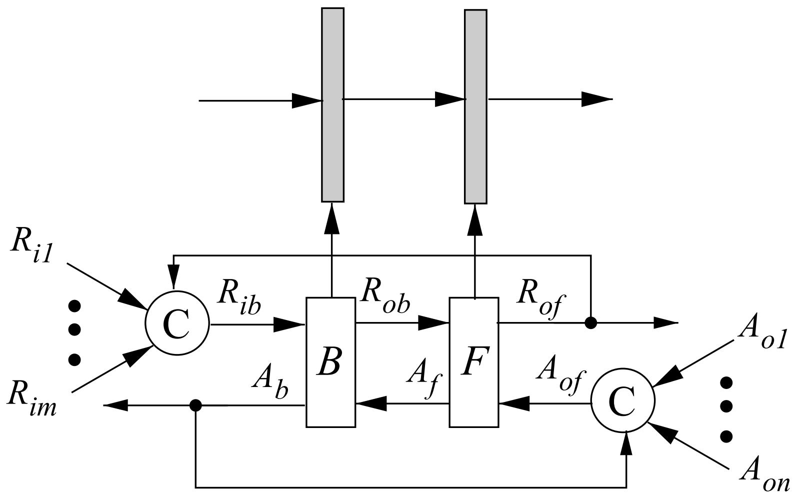


Figure 5.2. DFVD Controller

5.3.1. DFVD Controller

The proposed Design For Verification Desynchronization (DFVD) controllers for a pipeline latch pair is shown in Figure 5.2. The difference between the proposed controller and the controller in the desynchronized circuit of [40] is the feedback of R_{of} to the muller-C element that generates R_{ib} and the feed forward of A_b to the muller-C element that generates A_{of}. The connections are not part of the original controller. The A_b connection enforces the property that if the controller currently holds only one token in the F latch, then the controller will hold on to that token until it has received a new token in the B latch. The F latch will drop its token when it receives an acknowledge (A_{of}^+). Since A_b is connected to the muller-C element that generates A_{of}, unless latch B acknowledges the receipt of a new token by asserting A_b, the F latch will not drop its token.

An additional dependency is enforced by the R_{of} connection that allows the B latch to receive a new token only when the F latch has signaled a request on the output side. The new controller results in only minor delays but satisfies properties that allow for reachability analysis (see Section 5.4).

Table 5.1. Worst Case Delay Analysis of DFVD Controller

State Label	State $\langle A_b R_{ib} R_{ob} A_f R_{of} A_{of} \rangle$	Event	Delay
S1	$\langle$ 000100 $\rangle$	R_{ob}^-	N
S2	$\langle$ 000110 $\rangle$	R_{of}^+	N
S3	$\langle$ 010110 $\rangle$	R_{ib}^+	Y
S4	$\langle$ 110110 $\rangle$	A_b^+	N
S5	$\langle$ 100110 $\rangle$	R_{ib}^-	Y
S6	$\langle$ 100111 $\rangle$	A_{of}^+	Y
S7	$\langle$ 100011 $\rangle$	A_f^-	N
S8	$\langle$ 101011 $\rangle$	R_{ob}^+	N
S9	$\langle$ 101001 $\rangle$	R_{of}^-	N
S10	$\langle$ 101101 $\rangle$	A_f^+	N
S11	$\langle$ 001101 $\rangle$	A_b^-	N
S12	$\langle$ 001100 $\rangle$	A_{of}^-	Y

We now estimate the *worst case* increase in delay for the new controller by estimating the maximum delay between consecutive R_{ib}^+ transitions in the controller of latch B. This delay gives us the minimum time between consecutive sets of data getting stored into a latch. In the new controller circuit R_{ib} cannot change to 1 (or 0) unless R_i's (R_{i1}–R_{im}) and R_{of} are all 1 (or 0). Similarly A_{of} cannot change to 1 (or 0) unless A_o's (A_{o1}–A_{on}) and A_b are all 1 (or 0). The set of transitions that lead to *worst case* delay for the proposed controller circuit is shown in Table 5.1. This has been derived from the state diagram of an individual semi-decoupled 4-phase controller of [57] (see Figure 8 in [57]). The controller transitions from state S_i to S_{i+1}, starting at state S_1 and until it reaches S_{12}. From S_{12}, it transitions back to S_1. The events that causes the state transition is also shown in Table 5.1. Delays occur when a transition of R_{ib} or A_{of} has to occur.

From the state diagram it is clear that it takes 12 state transitions for two consecutive R_{ib}^+ transitions to occur. However without the new connections to R_{ib} and A_{of} it takes 8 state transitions for two consecutive R_{ib}^+ transitions to occur. Thus we obtain a *worst case* delay of 4 state transitions for the new controller to have two consecutive R_{ib}^+ compared to the existing semi-decoupled 4-phase controller of [57], which is usually negligible compared to the delay of pipeline processing logic in a stage. Also, note that many of the transitions of the additional muller-C elements can take place simultaneously with other events in the circuit and on average the performance degradation could be much lower.

5.4. Reachability Analysis of DFVD Controller Network

To perform verification, we need to compute the reachable states of the desynchronized pipeline controller network. Computing reachable states has two ends. First, unreachable states can be inconsistent w.r.t. the correctness property and flag spurious counter examples that hinder the verification process. Identifying reachable states of the implementation solves this problem as verification properties can now be checked only on the reachable states ensuring that spurious counter examples are eliminated. Second, our procedure for computing refinement maps for desynchronized pipelined machines is based on reachability analysis. ***Note that the reachability method eliminates unreachable states that hinder verification, which is what is required. The reachable states of the controller network may in fact only be a subset of the set of states computed by the reachability method.***

We now describe the general invariant generation rules. The first 8 rules (P1-P8) apply to the DFVD controller shown in Figure 5.2. Note that these rules are properties of the DFVD controller, and should be applied to each of the DFVD controllers in a DFVD pipeline controller network.

The acknowledge signal for the front and back latches A_f and A_b, respectively, also act as the clock for the front and back latches. When A_f or A_b are asserted, the corresponding latches are in a hold state, and when A_f or A_b are de-asserted, the corresponding latches are transparent (not holding any data tokens). Thus Property P_1 is a significant property as it implies that the DFVD controller will always be in a state where one or both of the latches is in a hold state. In other words, the DFVD controller will never reach a state where both latches are empty/transparent. This is not a property satisfied by the desynchronization controller proposed by [40], which allows the state where both latches are transparent. Property P_1 makes it possible for us to compute reachable states and define refinement maps in a systematic manner for desynchronized pipelines.

P_1: $A_b \vee A_f$

Property P_1 is not an invariant by itself, because there are states of the DFVD controller, which satisfy the property, but which can transition to states that do not satisfy P_1. Therefore, we need low-level properties P_2–P_8 that eliminate all such states.

P_2: $\langle A_b \wedge A_f \wedge R_{of} \rangle \rightarrow (\neg R_{ob})$

Properties P_3–P_5 identify the conditions under which the muller-C element corresponding to A_{of} should hold values of 0 and 1.

P_3: $\langle A_b \wedge A_f \wedge (\neg R_{of}) \rangle \rightarrow (\neg A_{of})$

P_4: $\langle (\neg A_b) \wedge A_f \wedge R_{of} \rangle \rightarrow (\neg A_{of})$

P_5: $\langle A_b \wedge (\neg A_f) \rangle \rightarrow A_{of}$

Properties P_6–P_8 identify the conditions under which the muller-C element corresponding to R_{ib} should hold values of 0 and 1.

P_6: $\langle A_b \wedge (\neg A_f) \wedge (\neg R_{of}) \rangle \rightarrow R_{ib}$

P_7: $\langle (\neg A_b) \wedge A_f \wedge (\neg R_{of}) \rangle \rightarrow (\neg R_{ib})$

P_8: $\langle A_b \wedge A_f \wedge R_{of} \rangle \rightarrow R_{ib}$

The conjunction of properties P_1–P_8 form an inductive invariant, which we have verified using the ACL2-SMT verification system [62] by proving that for every state of the DFVD controller that satisfies the conjunction of properties P_1–P_8, its successor also satisfies the conjunction of P_1–P_8.

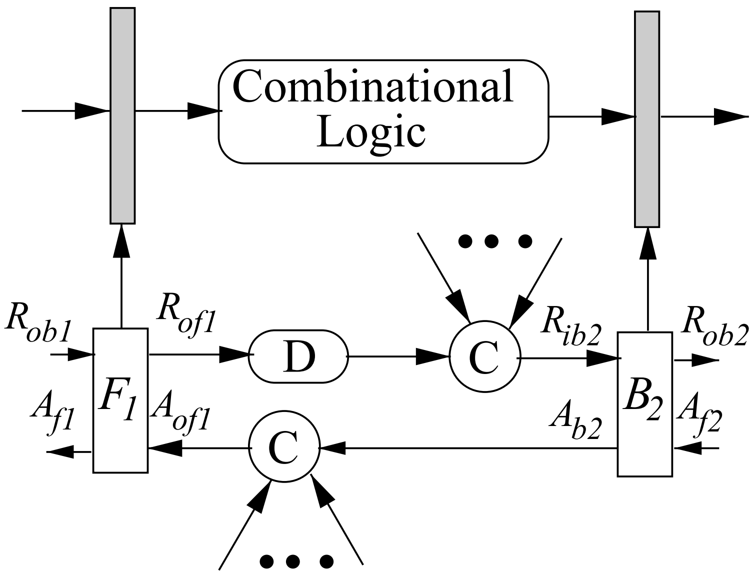


Figure 5.3. DFVD Controller Circuit for Data Transfer

Properties P_9–P_{15} apply to the circuit shown in Figure 5.3 that occurs in the desynchronized pipeline controller network when data is passing from one stage of the pipeline to another. In this situation, the front latch of the source stage is connected to the back latch of the destination stage. Hence the front controller of the source stage (labeled F_1 in the figure) is connected to the back controller of the destination stage (labeled B_2 in the figure).

Properties/rules P_9–P_{15} should be applied to every source to destination connection in the pipeline including feedback connections as well. The properties P_9–P_{15} are used to eliminate inconsistent states by identifying the conditions in which the muller-C elements corresponding to R_{ib2} and A_{of1} hold values of 1 and 0.

P_9: $\langle(\neg A_{f1}) \wedge (\neg A_{b2})\rangle \rightarrow (\neg R_{ib2})$

P_{10}: $\langle(\neg A_{f1}) \wedge R_{of1}\rangle \rightarrow R_{ib2}$

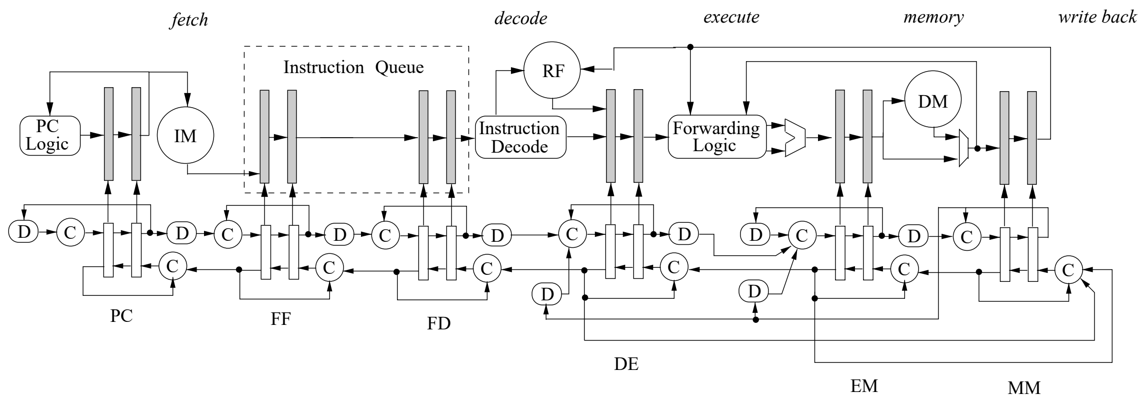


Figure 5.4. High-Level Organization of Desynchronized 6-Stage Pipelined Processor Model

P_{11}: $\langle A_{f1} \wedge (\neg A_{b2}) \wedge (\neg R_{of1})\rangle \rightarrow (\neg R_{ib2})$

P_{12}: $\langle A_{f1} \wedge A_{b2} \wedge R_{of1}\rangle \rightarrow R_{ib2}$

P_{13}: $\langle A_{f1} \wedge A_{b2} \wedge (\neg R_{of1})\rangle \rightarrow A_{of1}$

P_{14}: $\langle A_{f1} \wedge (\neg A_{b2}) \wedge R_{of1}\rangle \rightarrow (\neg A_{of1})$

P_{15}: $\langle (\neg A_{f1}) \wedge A_{b2}\rangle \rightarrow A_{of1}$

The conjunction of properties P_9–P_{15} also form an inductive invariant, which we have verified using the ACL2-SMT system by proving that for every state of the circuit shown in Figure 5.3 that satisfies the conjunction of properties P_9–P_{15}, its successor also satisfies the conjunction of P_9–P_{15}.

5.5. Desynchronized Pipelined Models

Five desynchronized pipelined processor models were developed and used as benchmarks to demonstrate the applicability and efficiency of the proposed verification solution for desynchronized systems. The models are specified in SMT-2 format. First, a 5-stage synchronous pipelined processor model based on the DLX pipeline [63] was constructed. Three desynchronized versions of the synchronous pipeline were developed, including DPM5-1, DPM5-2, and DPM5-5. In DPM5-1, one desynchronization controller is used to control all the stages of the pipeline using the idea of clustering [64]. Clustering is also used in DPM5-2, where two desynchronization controllers are employed (one controller for the fetch and decode stages, and the second controller for the execute, memory, and write back stages). DPM5-5 is a fully desynchronized model, where 5 controllers are used (one for each stage of the pipeline). The fetch stage in DPM5-5 is further pipelined (resulting in a short instruction queue) to create DPM6-6 and DMP7-7, both of which are fully desynchronized models employing one controller for each pipeline stage. The high-level organization of DPM6-6 is shown in Figure 5.4.

The models are specified at the term-level [47, 65], an abstraction level in which the bit-vector data path is abstracted using integers (also called terms in this context). Also, functions that operate on data are abstracted using Uninterpreted Functions (black box functions that only satisfy the property that equal inputs produce equal outputs). Term-level abstraction is used as it drastically improves the efficiency of verification.

5.6. Refinement-Based Verification

The goal of our verification procedure is to show equivalence between a pipelined desynchronized circuit/system and its pipelined synchronous specification. The notion of equivalence that we use is Well Founded Equivalence Bisimulation (WEB) refinement [15] and is based on stuttering bisimulation. Proving refinement guarantees that every behavior of the implementation is matched by behavior of the specification and vice versa. A detailed description of the theory of refinement can be found in [15]. It is enough to check the following correctness formula [16] to establish refinement (thereby establish equivalence) between an implementation and its specification.

Definition 1. *(Core WEB Refinement Correctness Formula)*

$$\begin{aligned} \langle \forall w \in \texttt{IMPL} \ :: \quad & s = r(w) \ \ \wedge \ \ u = Sstep(s) \ \ \wedge \\ & v = Istep(w) \ \ \wedge \ \ u \neq r(v) \\ \rightarrow \quad & s = r(v) \ \ \wedge \ \ rank(v) < rank(w) \rangle \end{aligned}$$

In the formula above, `IMPL` denotes the set of implementation states, *Istep* is a step of the implementation machine, and *Sstep* is a step of the specification machine. The refinement map r (a mechanism not found in stuttering bisimulation) is a function that maps implementation states to specification states thereby making it easy to compare systems at different abstraction levels. *rank*, used for deadlock detection, is a witness function from implementation states to natural numbers whose value decreases when there is stutter. The proof obligation that $s = r(v)$ is the safety component and guarantees that if the implementation makes progress, then the result of

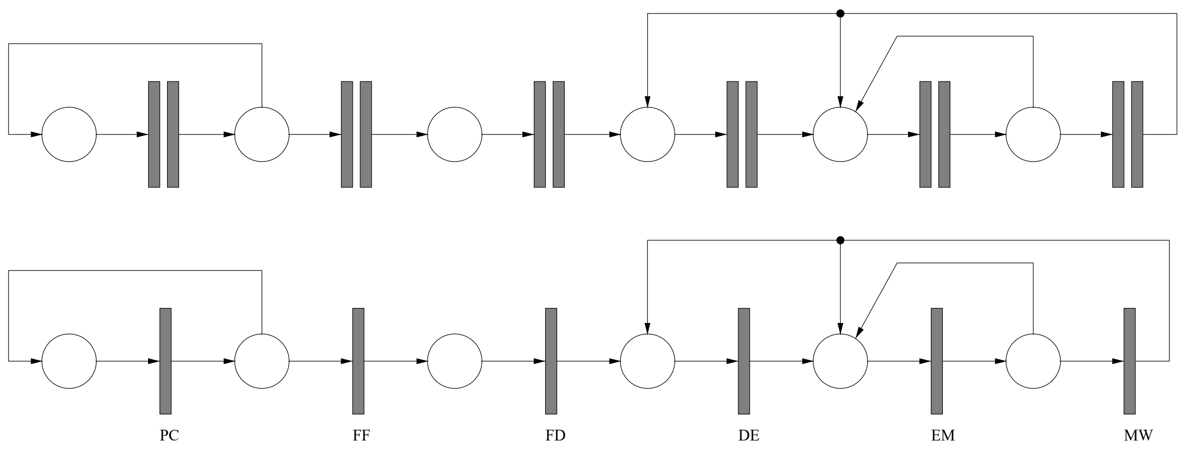


Figure 5.5. The 6-Stage Desynchronized Pipeline and the Corresponding Parent Synchronous Pipeline.

that progress is correct as given by the specification. The proof obligation that *rank*(*next-impl*) < *rank*(*impl*) is the liveness component and guarantees that the machine will not deadlock, *i.e.*, will always make forward progress. In this work, we solve the problem of safety verification for desynchronized pipelines and reserve liveness verification for future work.

The specific steps involved in a refinement-based verification methodology for checking safety are: (a) Compute the states of the implementation model that are reachable from reset (known as reachable states). ***We use the rules given in Section 5.4 to generate invariant properties that characterize the reachable states of any desynchronized pipeline controller network.*** (b) Construct a refinement map. (c) Use the models and the refinement map to state the safety component of the refinement-based correctness formula for the implementation model, which can then be automatically checked for the set of all reachable states using a decision procedure.

Therefore, to perform verification, in addition to the implementation and specification models, and reachability analysis, we also require a refinement map.

5.7. Circular Desynchronized Pipelines

For equivalence verification based on WEB refinement, a refinement map is required that maps every reachable state of the desynchronized circuit (implementation) to a synchronous circuit state (specification). Consider the example of the 6-stage pipelined circuit shown in Figure 5.4. The structure of the 6-stage desynchronized pipeline and corresponding parent synchronous pipeline is shown in Figure 5.5. There are several challenges to building a refinement map.

1. The desynchronized pipeline with DFVD controllers can have any where between 6 to 12 valid tokens in the pipeline, whereas the parent synchronous pipeline will always have 6 valid tokens. Here, we identify latches in hold state as valid tokens. Note that by this definition, flip-flops are always valid. The reason for the above is that all stages of the synchronous pipeline are synchronized with the global clock. Whereas, stages of the desynchronized pipeline proceed at differing speeds.

2. Valid tokens in many states of the desynchronized pipeline can be redundant. The synchronous pipeline will never have redundant tokens.

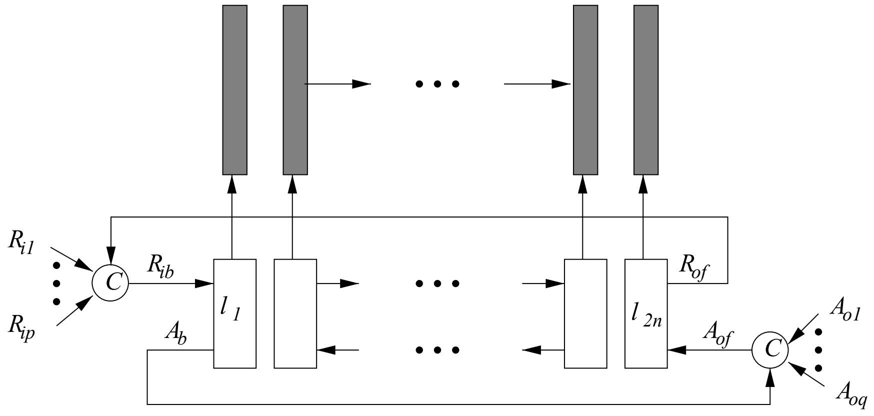


Figure 5.6. n-Stage Circular Desynchronized Pipeline.

Based on the above, an n stage desynchronzed pipeline state can have anywhere between 1 and $2n$ non-redundant tokens. The states of the parent synchronous pipeline will always have n non-redundant tokens. Defining refinement maps between desynchronized and parent synchronous pipelines, when there is so much variation in the number of tokens and how the tokens are distributed in desynchronized pipelines, is hard to resolve. States that have more than n non-redundant tokens need to be reduced to n, using flushing techniques, before mapping to a synchronous state. States with less than n non-redundant tokens has to be modified, by injecting new tokens to the pipeline, so that the state has n non-redundant tokens. A refinement map that can handle such variations, using techniques mentioned above, will be computationally very complex and cause efficiency of verification times to degrade. Therefore, we propose another design for verification technique to limit these variations. The technique is a circular desynchronized pipeline and is shown in Figure 5.6. For the purpose of explaining the properties of the circular desynchronized pipeline and the computation of the refinement map, we label latches of an n-stage desynchronized pipeline $\langle l_1, l_2, ..., l_{2n} \rangle$ and the registers of the parent synchronous pipeline $\langle r_1, r_2, ..., r_n \rangle$.

For any n-stage pipeline, the corresponding circular desynchronized pipeline is obtained by

1. connecting the R_o of the last latch (l_{2n}) to the muller-C element that generates R_i of the first latch (l_1).
2. connecting the A of l_1 to the muller-C element that generates A_o of l_{2n}.

Adding the above connections to the circuit synchronizes the output of l_{2n} with the input of l_1, and creates a loop like structure in the controller network. This synchronization ensures that the number of non-redundant tokens remain constant in all the reachable states of the circular desynchronized pipeline (see Property 5). This important property of the circular desynchronized pipeline simplifies computing the refinement map, since the the number of non-redundant tokens does not vary, as detailed in section 5.8. First, we present the conditions of the DFVD controller network states that result in redundancy among tokens. Then the following property (Property 4) is proved about the latches carrying redundant tokens in a desynchronized pipeline, which is used to derive subsequent properties.

Since redundancy is a result of desynchronization, the latches that have redundant data in a desynchronized pipelined machine state can be determined by observing the controller state. There are two conditions of the DFVD controller network states that identify redundant data in the pipelined system, which are given below.

Rd_1: $A_b \wedge R_{ob} \wedge A_f$

The first condition (Rd_1) occurs between the latch pair used to separate two stages of a pipeline and is depicted in Figure 5.2. The condition occurs when the B latch is holding its data (indicated by A_b), which has also been transmitted to the F latch (indicated by $R_{ob} \wedge A_f$).

Rd_2: $A_{f1} \wedge R_{of1} \wedge R_{ib2} \wedge A_{b2}$

The second condition (Rd_2) occurs between the F latch of a source latch pair (controller 1) and B latch of a destination latch pair (controller 2). The corresponding circuit is shown in

Figure 5.3. The condition occurs when the F latch of controller 1 is holding its data (indicated by A_{f1}), which has also been transmitted to the B latch of controller 2 (indicated by $R_{of1} \wedge R_{ib2} \wedge A_{b2}$).

Property 4 is expressed as a CTL property. In the property, l^v is true iff latch l holds a valid token and l^r is true iff latch l holds a valid token that is redundant. Also, l_s is the source latch and l_d is the destination latch.

Property 4.

$$AG\left\{l_s^r \rightarrow A\left[l_d^v U(\neg l_s^v l_d^v)\right]\right\}$$

Proof. When l_s is the B latch and l_d is the F latch of a pair of latches that seperate two stages, the condition for redundant latches is Rd_1 (where $b \leftarrow s$ and $f \leftarrow d$):

$$A_s \wedge R_{os} \wedge A_d$$

When l_s is an F latch of a source latch pair (controller 1) and l_d is a B latch of a destination latch pair (controller 2), condition for redundant latches is Rd_2 (where $f1 \leftarrow s$ and $b2 \leftarrow d$):

$$A_s \wedge R_{os} \wedge R_{id} \wedge A_d$$

For l_d to become invalid ($\neg l_d^v$), A_d of the corresponding latch controller (LC) should deassert. Using the logic equations that describe the LC (See section 5.3), we have

$$A_d^- = \neg R_{id} \wedge R_{od} \wedge A_{od}$$

Therefore, R_{id} needs to be deasserted to deassert A_d. R_{id} remains asserted until R_{os} deasserts because R_{os} is an input to the muller-C element that generates R_{id} (R_{os} and R_{id} are the same signal when l_s and l_d belong to the same DFVD controller). From the logic equations for LC:

$$R_{os}^- = \neg A_s$$

Hence, R_{os} cannot deassert until A_s deasserts. Deasserting A_s invalidates l_s ($\neg l_s^v$). $\square$

In the above property, the token in l_d remains valid as long as the token in l_s remains valid if l_s is holding a redundant token.

Property 5. *An n-stage circular desynchronized pipeline with m non-redundant tokens in the initial state, where $m \leq 2n$, will have exactly m unique valid tokens in every reachable state.*

Proof. The circular desynchronized pipeline design makes l_{2n} an input to l_1. Note that the changes are made to the controller network, and the datapath is not altered. The added dependence puts two constraints on the pipeline.

- Prevents l_1 from accepting a new token until l_{2n} is valid.
- Prevents any token in l_{2n} from retiring until l_1 accepts a new token.

New tokens are inserted to the pipeline at l_1. Once l_1 accepts a new token, l_{2n} becomes redundant. Therefore, the number of non-redundant tokens in the pipeline does not change. Because the token in l_{2n} is redundant and l_{2n} is an input to l_1, l_1 remains valid until l_{2n} becomes invalid (Property 4). Therefore, l_1 cannot accept a new token until the token in l_{2n} retires. □

In the above proof, it was mentioned that the token in l_{2n} becomes redundant once l_1 accepts a token. Note that the tokens are associated with the state of the controller network. In the actual datapath, the data in l_{2n} does not become redundant because l_1 is really accepting a new input. This is a special case, and we handle it by flushing the instruction in l_{2n} because it is not going to map to any stage of the synchronous pipeline.

5.8. Refinement Map for Circular Desynchronized Pipelines

In the initial state of a desynchronized pipeline, every F latch of the latch pairs has a non-redundant token and every B latch is empty. Thus, an n-stage desynchronized pipeline will have n non-redundant tokens at the initial state. If the desynchronized pipeline is circular, then the pipeline will have n non-redundant tokens in every reachable state (Property 5). Note, however, that there can be multiple redundant tokens in these reachable states. Therefore, a reachable state can have anywhere between n to $2n$ valid tokens, but exactly n non-redundant tokens.

The problem of computing a witness refinement map reduces to identifying where these n non-redundant tokens are located in the desynchronized pipeline, and mapping them onto the n stages of the synchronous pipeline. The challenge here is to develop a refinement map function that (1) is general and applicable to any circular desynchronized pipeline; (2) is applicable to every reachable state and, therefore, has to account for all possible combinations in which these tokens can be distributed in the desynchronized pipeline; (3) can be encoded in QF_UFLIA (unquantified linear integer arithmetic with uninterpreted sort and function symbols), so that the refinement proof obligations (which use the refinement map function) can be checked using an SMT solver for this logic.

We introduce the notion of projection predicates, which are used to construct refinement maps.

Definition 21. *A projection predicate $p_{i \leftarrow j}$ is a predicate that is true only when the refinement map projects latch l_j in the desynchronized pipeline to register r_i of the synchronous pipeline.*

Since there are n pipeline registers in the synchronous machine and $2n$ latches in the desynchronized machine, we have $1 \leq i \leq n$ and $1 \leq j \leq 2n$ in the above definition.

Next, we define projection functions $pf_{i \leftarrow j}$.

Definition 22. *A projection function $pf_{i \leftarrow j}$ is a function that computes the value of register r_i of the synchronous machine from the value of latch l_j in the desynchronized machine.*

We now provide a formula for the refinement map, which essentially gives the value of each pipeline register r_i, where $1 \leq i \leq n$, for a given desynchronized state.

$$r_i = \begin{cases} pf_{i \leftarrow 1}(l_1), & p_{i \leftarrow 1} \\ pf_{i \leftarrow 2}(l_2), & p_{i \leftarrow 2} \\ \vdots & \\ pf_{i \leftarrow 2n}(l_{2n}), & p_{i \leftarrow 2n} \end{cases}$$

Next, we describe how projection predicates and projection functions are computed. Note that for the above definition to be complete, at least one of the projection predicates should be true. Subsequently, we also state and prove this property.

5.8.1. Projection Predicates

We propose an inductive method to compute the projection predicates. The definition for $p_{1\leftarrow j}$, the base case, is given below.

Definition 23. *For all $j \in \mathbb{N}$ such that $1 \leq j \leq 2n$ where n is the number of synchronous stages,*

$$p_{1\leftarrow j} = \begin{cases} 0 & , n+1 < j \leq 2n \\ \neg l_j^r l_j^v \left(\bigwedge_{k=1}^{j-1} l_k^v \rightarrow l_k^r\right) & , 1 \leq j \leq n+1 \end{cases}$$

Out of the n non-redundant tokens in a given circular desynchronized pipeline state, the first (leftmost) non-redundant token maps to r_1. When the n non-redundant tokens are placed on the last n pipeline latches (from l_{n+1} to l_{2n}), l_{n+1} maps to r_1. Therefore, l_{n+1} is the furthest latch that can be mapped to r_1 resulting $p_{1\leftarrow j}$ to be false for $(n+1) < j \leq 2n$. For $1 \leq j \leq (n+1)$, $p_{1\leftarrow j}$ is true iff l_j holds a valid token that is non-redundant ($\neg l_j^r l_j^v$) and all valid tokens in the latches before l_j are redundant so that l_j is holding the first non-redundant token.

The inductive step for computing $p_{i\leftarrow j}$, which is derived from $\left\{p_{(i-1)\leftarrow 1}, p_{(i-1)\leftarrow 2}, \ldots, p_{(i-1)\leftarrow (j-i)}\right\}$, is defined below.

Definition 24. *For all $i, j \in \mathbb{N}$ such that $1 < i \leq n$ and $1 \leq j \leq 2n$,*

$$p_{i\leftarrow 1} = 0$$

for $1 < j \leq 2n$

$$\begin{aligned} p_{i\leftarrow j} = (\neg l_j^r l_j^v) \Bigg[& p_{(i-1)\leftarrow 1} \bigwedge_{k=2}^{j-1} (l_k^v \rightarrow l_k^r) + p_{(i-1)\leftarrow 2} \bigwedge_{k=3}^{j-1} (l_k^v \rightarrow l_k^r) \\ & + \ldots \\ & + p_{(i-1)\leftarrow (j-2)} \bigwedge_{k=j-1}^{j-1} (l_k^v \rightarrow l_k^r) + p_{(i-1)\leftarrow (j-1)} \Bigg] \end{aligned}$$

more concisely,

$$(\neg l_j^r l_j^v) \bigvee_{m=1}^{j-1} \left[p_{(i-1)\leftarrow m} \bigwedge_{k=m+1}^{j-1} (l_k^v \rightarrow l_k^r) \right]$$

If l_1 holds a valid non-redundant token, that token is the first non-redundant token of the pipeline state. Therefore, $p_{i \leftarrow 1}$ is always false for $1 < i \leq n$.

The first condition for l_j to map to r_i is that l_j must hold a valid non-redundant token. The second condition vary with the projection predicate values for r_{i-1}. Given l_m maps to r_{i-1} ($p_{(i-1) \leftarrow m}$), the goal is to find out if l_j holds the next non-redundant token after l_m. If $m \geq j$, $p_{i \leftarrow j}$ is false. Otherwise, l_j holds the next non-redundant token if all the valid tokens in the latches between l_m and l_j are redundant.

5.8.2. Projection Functions

$pf_{i \leftarrow j}$ projects the token in l_j to r_i. Projection functions can be categorized in to three types. The first is the direct projection when j is equal to $2i$ or $2i{-}1$; l_j and r_i are in the same pipeline stage. In direct projection, the fields of l_j are copied to r_i. The other two types of projections are forward projection and reverse projection. If $pf_{i \leftarrow j}$ is not a direct projection, the first step is to flush the inflight instructions that precede the instruction in l_j; non-redundant instructions are commited to the memory using completion functions. If $j < 2i{-}1$ (forward projection), the data in l_j are projected to r_i using partial-completion functions. Partial-completion functions compute the values of the fields in r_i using the values of the fields in l_j and the updated memory obtained by flushing the preceding instructions in the pipeline. If $j > 2i$, the data in l_j needs to be reverse projected to r_i. When data move forward through the pipeline stages, information are lost as a result of partial completion of the instructions. This is a challenge for reverse projection. To overcome this challenge, essential fields to recover lost information are retained throughout all the pipeline stages even though they are obsolete for the completion of the instruction. In the example pipeline, pc and un-decoded instruction are preserved upto the writeback stage.

5.8.3. Consistency Invarients

The consistency invariants guarantee that interdependent fields within a latch are consistent in the current state. Assume that the back latch of *ff* stage in the example desynchronized pipeline (Fig. 5.5) reverse projects to *pc* pipeline register in the parent synchronous pipeline. Out of the two fields in *ff* stage (program counter and instruction), only program counter field gets projected to *pc* stage. When the synchronous machine is stepped, data in *pc* moves to *ff* pipeline register, and the instruction field in the *ff* stage is read from instruction memory using *pc* as the address. The instruction field of the pipeline register *ff* should be consistent with the instruction field of the back latch of the *ff* stage in the desynchronized pipeline. This requirement is enforced by the following consistancy invarient on instruction (*ff.inst*) and program counter (*ff.pc*) fields of the desynchronized pipeline.

$$ff.inst = imem(ff.pc)$$

5.8.4. Duplicate Invarients

When a latch remains valid after passing on data to a destination latch, the overlapping fields of the two latches hold duplicate information. Duplicate invarients guarantee that the overlapping fields of such latch pairs are same. Every datapath connection that has overlapping fields at source and destination requires a duplicate invarient. For example, consider a reachable state of the 6-stage pipeline processor model (Fig. 5.4). The front latch of *em* (l_{emf}) connects to the back latch of *mw* (l_{mwb}). There is also a feedback loop—forwarding logic—from l_{emf} to the back latch of *em* (l_{emb}). Assume l_{emf} is valid and passed on data to l_{mwb}, but not to l_{emb}. In the next state, l_{emf} would send data to l_{emb} if all the source latches of l_{emb} have valid data. Following duplicate invarient between l_{emf} and l_{mwb} guarantee that the data forwarded to l_{emb} is consistent with the data in l_{mwb}.

5.8.5. Memory Invarients

Memory invarients are similar to consistancy invarients. In states where memory gets updated, memory invarients gurantee that memory array state is consistent with the latch states. If value v is written to address a of the memory array *mem*, then for every latch i, $1 \leq i \leq 2n$, that contains fields v and a the following memory invarient should hold:

$$mem(l_i.a) = l_i.v$$

In the example 6 stage desynchronized pipeline (Fig. 5.5), data memory (*dmem*) is synchronized with the back latch of the *mw* stage (l_{mwb}). Assume a desynchronized state, in which the instruction in l_{mwb} updates *dmem*, and that l_{mwb} reverse projects to the pipeline register of the *em* stage (r_{em}) in the parent synchronous state. When the instruction in r_{em} moves forward to pipeline register in *mw* stage, it will update the data memory of the synchronous pipeline. Updated

Table 5.2. SMT Statistics

Processor Model	Decisions	Conflicts	Memory Used (MB)	Time (s)
DPM5-1	1,022	165	1.15	0.02
DPM5-2	3,564	706	1.84	0.2
DPM5-5	26,161	8,372	3.43	3.78
DPM6-6	45,377	16,058	4.38	9.73
DPM7-7	73,395	27,993	5.31	16.96
DPM-B1-5-2	1,458	170	1.66	0.07
DPM-B2-5-2	1,661	253	1.76	0.1

data memory of the synchronous pipeline should be consistent with *dmem* in the desynchronized state.This requirement is guaranteed by the following invariant on *dmem*.

$$memwrite(mwb.inst) \wedge l^{v}_{mwb} \\ \Rightarrow dmem(mwb.result) = mwb.arg1$$

5.9. Results

Table 5.2 reports the results of safety verification of the five desynchronized processor models described in Section 5.5 against the *pipelined* synchronous specifications. The experiments were performed on a Intel(R) Celeron(R) CPU 540, with a cache size of 1024 KB. Verification was performed on Z3 theorem prover (version 4.3.1) by Microsoft(R) Research. All models were written in smt-2 format.

Buggy Models: Two buggy variations of the DPM5-2 model were also checked using the verification procedure and resulted in the Z3 theorem prover flagging counter examples that pointed to the source of the bug. The buggy models are DPM-B1-5-2 and DPM-B2-5-2. In DPM-B1-5-2, we injected a bug in the data path. In the forwarding path for source operand 2 from memory stage to execute stage, the destination operand address is compared with the source address of operand 1 instead of operand 2. In DPM-B2-5-2, we injected a bug in the desynchronized pipeline controller network. The DPM5-2 model has 2 DFVD controllers. The R_{ob} signal instead of the R_{of} signal of controller 1 is connected to the R_{ib} muller-C element of controller 2. The verification statistics are also reported for the buggy models in Table 5.2.

5.10. Conclusions

Formal verification methods have become an integral part of the design cycle to ensure reliable IC designs. Therefore, verifiability has become an important consideration for any design paradigm. In this work, we propose improved verifiability for desynchronization, which is achieved with a worst case performance penalty of 4 muller-C element delay in pipeline throughput.

For future work, we plan to address liveness verification of desynchronized pipelines and explore compositional methods to improve scalability. We also plan to explore design for verification solutions for desynchronization with lower performance degradation.

6. CONCLUSION

This chapter concludes the work in this study. The dissertation focused on developing equivalence verification methods for design paradigms aimed at digital circuits for nanotechnologies: (a) synchronous elastic circuits, (b) NCL circuits, and (c) desynchronized circuits.

6.1. Nanoscale Circuit

Digital circuits are omnipresent in today's society. They are found in smart phones, automobiles, medical devices, home appliances, etc. They are also utilized in telecommunication systems, traffic control systems, systems to control nuclear reactors, etc. As circuits become smaller, new applications such as medical devices implanted in the human body (e.g. pacemaker) are discovered continuously. Many of these applications are safety-critical: a failure can lead to catastrophic events. Since digital circuits are readily used in many applications, including safety-critical applications, it is paramount to verify the correctness of modern circuit designs. In fact, only 27% of the total effort of a commercial design cycle is actually spent on design, devoting the rest to error detection and prevention [66].

Commercial design processes largely use simulation based testing to verify designs. Testing methods scales well with design size. However, bugs in rarely exercised paths can escape detection: these are called *corner case* bugs. Therefore, commercial circuit design companies use formal methods in their verification process in an effort to find these corner case bugs that could escape traditional testing [67, 68, 69].

6.2. Final Remarks

As the ability of synchronous paradigm to produce reliable circuit designs economically is diminishing with technology scaling, circuit designers are showing more and more interest in asynchronous design approaches. A new class of design paradigms: (a) Synchronous elastic circuits (b) NCL circuits, and (c) Desynchronized circuits, are proposed that can exploit existing CAD flows to synthesize delay-tolerant circuits. In this study, an initial successful effort is made to develop formal equivalence verification methods for this new class of design paradigms. This has remained largely an open problem before the work presented in this dissertation, and one of the main challenges in integrating them to commercial designs. The developed methods are promising and efficiently deal with state explosion problem: a major obstacle in formal verification. However, there is a long way to go to develop these methods into tools of commercial strength. Possible future work is discussed next.

6.3. Recommendation for Future Work

Work done in this study can be extended in three directions.

1. Complex systems with multiple inputs and outputs can become highly non-deterministic. Heuristics needs to be developed to deal with this problem in reachability analysis.

2. Liveness aspect of WEB refinement theory needs to be addressed to verify the design is deadlock free. Efforts can be made to develop a generic *rank* function that is applicable to a large class of circuit structures.

3. compositional verification approaches needs to be explored to improve the scalability.

REFERENCES

[1] A. Taubin, J. Cortadella, L. Lavagno, A. Kondratyev, and A. Peeters, "Design automation of real-life asynchronous devices and systems," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 1, pp. 1–133, 2007.

[2] M. T. Bohr *et al.*, "Interconnect scaling-the real limiter to high performance ulsi," in *International Electron Devices Meeting.* Institute of Electrical & Electronic Engineers, Inc (IEEE), 1995, pp. 241–244.

[3] S. C. Smith and J. Di, *Designing Asynchronous Circuits using NULL Convention Logic (NCL)*, ser. Synthesis Lectures on Digital Circuits and Systems. Morgan & Claypool Publishers, 2009.

[4] A. Yakovlev, P. Vivet, and M. Renaudin, "Advances in asynchronous logic: From principles to gals amp; noc, recent industry applications, and commercial cad tools," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 1715–1724.

[5] K. M. Fant and S. A. Brandt, "Null convention logictm: a complete and consistent logic for asynchronous digital circuit synthesis," in *Application Specific Systems, Architectures and Processors, 1996. ASAP 96. Proceedings of International Conference on*, Aug 1996, pp. 261–273.

[6] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Latency insensitive protocols," in *CAV*, 1999, pp. 123–133.

[7] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 1904–1921, Oct 2006.

[8] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *DAC*, 2006, pp. 657–662.

[9] R. B. Reese, S. C. Smith, and M. A. Thornton, "Uncle - an rtl approach to asynchronous design," in *ASYNC*, J. Sparsø, M. Singh, and P. Vivet, Eds. IEEE Computer Society, 2012, pp. 65–72.

[10] N. Andrikos, L. Lavagno, D. Pandini, and C. P. Sotiriou, "A fully-automated desynchronization flow for synchronous circuits," in *2007 44th ACM/IEEE Design Automation Conference*, June 2007, pp. 982–985.

[11] E. Kilada and K. S. Stevens, "Control network generator for latency insensitive designs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 1773–1778. [Online]. Available: http://dl.acm.org/citation.cfm?id=1870926.1871354

[12] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial hdl synthesis tools," in *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, ser. ASYNC '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 114–. [Online]. Available: http://dl.acm.org/citation.cfm?id=785166.785308

[13] K. S. Stevens, Y. Xu, and V. Vij, "Characterization of asynchronous templates for integration into clocked cad flows," in *Asynchronous Circuits and Systems, 2009. ASYNC '09. 15th IEEE Symposium on*, May 2009, pp. 151–161.

[14] J. Cortadella, M. Galceran-Oms, M. Kishinevsky, and S. S. Sapatnekar, "Rtl synthesis: From logic synthesis to automatic pipelining," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2061–2075, Nov 2015.

[15] P. Manolios, "Mechanical verification of reactive systems," Ph.D. dissertation, University of Texas at Austin, August 2001, see URL http://www.cc.gatech.edu/~manolios/publications.html.

[16] ——, "Correctness of pipelined machines," in *Formal Methods in Computer-Aided Design–FMCAD 2000*, ser. LNCS, W. A. Hunt, Jr. and S. D. Johnson, Eds., vol. 1954. Springer-Verlag, 2000, pp. 161–178.

[17] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[18] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Coping with latency in soc design," *IEEE Micro*, vol. 22, no. 5, pp. 24–35, 2002.

[19] M. R. Casu and L. Macchiarulo, "Adaptive latency-insensitive protocols," *IEEE Design and Test of Computers*, vol. 24, pp. 442–452, 2007.

[20] C.-H. Li and L. P. Carloni, "Leveraging local intracore information to increase global performance in block-based design of systems-on-chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 2, pp. 165–178, 2009.

[21] G. Hoover and F. Brewer, "Synthesizing synchronous elastic flow networks," in *DATE*. IEEE, 2008, pp. 306–311.

[22] M. Galceran-Oms, A. Gotmanov, J. Cortadella, and M. Kishinevsky, "Microarchitectural transformations using elasticity," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 7, pp. 18:1–18:24, Dec. 2011.

[23] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary, "Synchronous elastic networks," in *FMCAD*. IEEE Computer Society, 2006, pp. 19–30.

[24] "Benchmarks and Tool," 2013, see URL http://venus.ece.ndsu.nodak.edu/~s.srinivasan/cav2013.tar.gz.

[25] P. Manolios, "A compositional theory of refinement for branching time," in *12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003*, ser. LNCS, D. Geist and E. Tronci, Eds., vol. 2860. Springer-Verlag, 2003, pp. 304–318.

[26] "SMT-LIB," 2012, see URL http://www.smtlib.org/.

[27] "Verific Design Automation, Inc." 2012, see URL http://www.verific.com/.

[28] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[29] C.-H. Li, R. L. Collins, S. Sonalkar, and L. P. Carloni, "Design, implementation, and validation of a new class of interface circuits for latency-insensitive design," in *IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2007)*. IEEE, 2007, pp. 13–22.

[30] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla, "Validating families of latency insensitive protocols," *IEEE Transations on Computers*, vol. 55, no. 11, pp. 1391–1401, 2006.

[31] D. Sokolov, I. Poliakov, and A. Yakovlev, "Analysis of static data flow structures," *Fundam. Inform.*, vol. 88, no. 4, pp. 581–610, 2008.

[32] S. K. Srinivasan, Y. Cai, and K. Sarker, "Refinement-based verification of elastic pipelined systems," *IET Computers & Digital Techniques*, vol. 6, no. 2, pp. 136–152, 2012.

[33] J. R. Burch, "Combining ctl, trace theory and timing models," in *Automatic Verification Methods for Finite State Systems*, ser. Lecture Notes in Computer Science, J. Sifakis, Ed., vol. 407. Springer, 1989, pp. 334–348.

[34] T. Yoneda and B.-H. Schlingloff, "Efficient verification of parallel real-time systems," *Formal Methods in System Design*, vol. 11, no. 2, pp. 187–215, 1997.

[35] C. J. Myers, *Asynchronous Circuit Design.* New York: Wiley, 2001.

[36] P. Loewenstein, "Formal verification of counterflow pipeline architecture," in *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, 1995.

[37] F. Verbeek and J. Schmaltz, "Verification of building blocks for asynchronous circuits," in *ACL2*, ser. EPTCS, R. Gamboa and J. Davis, Eds., vol. 114, 2013, pp. 70–84.

[38] A. Kondratyev, L. Neukom, O. Roig, A. Taubin, and K. Fant, "Checking delay-insensitivity: 104 gates and beyond," in *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, April 2002, pp. 149–157.

[39] S. K. Srinivasan and R. S. Katti, "Desynchronization: design for verification," in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 215–222.

[40] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1904–1921, 2006.

[41] S. C. Smith, "Gate and throughput optimizations for null convention self-timed digital circuits," Ph.D. dissertation, School of Electrical Engineering and Computer Science, University of Central Florida, 2001.

[42] L. Zhou, S. C. Smith, and J. Di, "Radiation hardened null convention logic asynchronous circuit design," *Journal of Low Power Electronics and Applications*, vol. 5, no. 4, pp. 216–233, 2015.

[43] J. Brady, A. M. Francis, J. Holmes, J. Di, and H. A. Mantooth, "An asynchronous cell library for operation in wide-temperature and ionizing-radiation environments," in *IEEE Aerospace Conference*, 2015.

[44] B. Hollosi, M. Barlow, G. Fu, C. Lee, J. Di, S. C. Smith, H. A. Mantooth, and M. Schupbach, "Delay-insensitive asynchronous alu for cryogenic temperature environments," in *IEEE International Midwest Symposium on Circuits and Systems*, 2008.

[45] R. B. Reece, "Uncle user manual," https://sites.google.com/site/asynctools/, (available February 2016).

[46] V. Wijayasekara, S. K. Srinivasan, and S. C. Smith, "Equivalence verification for NULL convention logic (NCL) circuits," in *32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19-22, 2014*. IEEE, 2014, pp. 195–201. [Online]. Available: http://dx.doi.org/10.1109/ICCD.2014.6974681

[47] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessor control,” in *Computer-Aided Verification (CAV ’94)*, ser. LNCS, vol. 818. Springer-Verlag, 1994, pp. 68–80.

[48] “Yices homepage,” 2007, see URL http://fm.csl.sri.com/yices.

[49] J. Cortadella, “Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers,” *IEICE Trans. Inf. Syst.*, vol. E80-D, no. 2, pp. 315–325, 1997.

[50] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic Synthesis of Asynchronous Controllers and Interfaces.* New York: Springer-Verlag, 2002.

[51] A. Martin and M. Nystrom, “Asynchronous techniques for system-on-chip design,” *Proc. IEEE*, vol. 94, no. 6, pp. 1089–1120, 2006.

[52] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. I. Pénzes, R. Southworth, and U. Cummings, “The design of an asynchronous mips r3000 microprocessor,” in *ARVLSI*, 1997, pp. 164–181.

[53] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective.* Prentice-Hall, 2003.

[54] S. M. Nowick, M. B. Josephs, and C. H. V. Berkel, “Special issue: Asynchronous circuits and systems,” *Proc. IEEE*, vol. 87, no. 2, pp. 217–396, 1999.

[55] J. Sparso and E. S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective.* Boston, MA: Kluwer, 2001.

[56] C. H. van Berkel and R. Saejis, “Compilation of communicating processes into delay insensitive circuits,” in *Proc. Int. Conf. Computer Design (ICCD)*, 1988, pp. 157–162.

[57] S. B. Furber and P. Day, “Four-phase micropipeline latch control circuits,” *IEEE Trans. VLSI Syst.*, vol. 4, no. 2, pp. 247–253, 1996.

[58] P. Manolios and S. K. Srinivasan, “A computationally efficient method based on commitment refinement maps for verifying pipelined machines.” in *Formal Methods and Models for Co-Design (MEMOCODE’05).* IEEE, 2005, pp. 188–197.

[59] T. Yoneda, B. Zhou, and B.-H. Schlingloff, "Verification of bounded delay asynchronous circuits with timed traces," in *Algebraic Methodology and Software Technology.* Springer, 1999, pp. 59–73.

[60] F. Verbeek and J. Schmaltz, "Verification of building blocks for asynchronous circuits," in *ACL2*, ser. EPTCS, R. Gamboa and J. Davis, Eds., vol. 114, 2013, pp. 70–84.

[61] S. K. Srinivasan and R. S. Katti, "Verification of desynchronized circuits," in *ISCAS'09*, 2009.

[62] S. K. Srinivasan, "Efficient verification of bit-level pipelined machines using refinement," Ph.D. dissertation, Georgia Institute of Technology, December 2007, see URL http://etd.gatech.edu/theses/available/etd-08242007-111625/.

[63] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface.* Elsevier Morgan Kaufmann, 2009.

[64] A. Davare, K. Lwin, A. Kondratyev, and A. L. Sangiovanni-Vincentelli, "The best of both worlds: the efficient asynchronous implementation of synchronous specifications," in *(DAC'04)*, 2004, pp. 588–591.

[65] M. N. Velev and R. E. Bryant, "Bit-level abstraction in the verfication of pipelined microprocessors by correspondence checking," in *FMCAD'98*, 1998, pp. 18–35.

[66] C. Baier, J.-P. Katoen *et al.*, *Principles of model checking.* MIT press Cambridge, 2008, vol. 26202649.

[67] B. Bentley, "Validating the intel(r) pentium(r) 4 microprocessor," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 244–248.

[68] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber *et al.*, "Replacing testing with formal verification in intel coretm i7 processor execution engine validation," in *Computer Aided Verification.* Springer, 2009, pp. 414–429.

[69] R. M. Gott, J. R. Baumgartner, P. Roessler, and S. I. Joe, “Functional formal verification on designs of pseries microprocessors and communication subsystems,” *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 565–580, July 2005.

[70] R. Gamboa and J. Davis, Eds., *Proceedings International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2013, Laramie, Wyoming, USA, May 30-31, 2013*, ser. EPTCS, vol. 114, 2013.

APPENDIX. LIST OF PUBLICATIONS

- Wijayasekara, V., & Srinivasan, S. K. (2013, October). Equivalence checking for synchronous elastic circuits. In Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on (pp. 109-118). IEEE.

- Wijayasekara, V. M., Srinivasan, S. K., & Smith, S. C. (2014, October). Equivalence verification for NULL Convention Logic (NCL) circuits. InComputer Design (ICCD), 2014 32nd IEEE International Conference on (pp. 195-201). IEEE.

- Dubasi, M. A. L., Srinivasan, S. K., & Wijayasekara, V. (2014). Timed refinement for verification of real-time object code programs. In Verified Software: Theories, Tools and Experiments (pp. 252-269). Springer International Publishing.

- Bilal, K., Malik, S. U. R., Khalid, O., Hameed, A., Alvarez, E., Wijaysekara, V., ... & Khan, U. S. (2014). A taxonomy and survey on green data center networks. Future Generation Computer Systems, 36, 189-208.