REFINEMENT-BASED VERIFICATION OF ELASTIC PIPELINED SYSTEM WITH EARLY EVALUATION

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

By

Yangwei Cai

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Electrical and Computer Engineering

May 2010

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Refinement-based Verification of

Elastic Pipelined System with Early Evaluation

By

Yangwei Cai

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

ABSTRACT

Cai, Yangwei, M.S., Department of Electrical and Computer Engineering, College of Engineering and Architecture, North Dakota State University, May 2010. Refinement-based Verification of Elastic Pipelined System with Early Evaluation. Major Professor: Sudarshan Srinivasan.

This thesis presents a formal verification procedure to check correctness of the synchronous elastic pipelined system that incorporates early evaluation against its synchronous parent system. Note that the goal of the verification procedure is not to establish the correctness of the algorithm for synthesizing elastic circuits, but instead, to find bugs and formally prove the correctness of elasticized designs with early evaluation. Dataflow through elastic architectures is complicated by the insertion of any number of elastic buffers in any place in the design. Elastic token-flow diagrams are introduced, which are used to track the flow of data in elastic architectures. We provide a method to construct such diagrams. The thesis also develops a highly automated and systematic procedure based on elastic token-flow diagrams that compute functions that map states of elastic systems to states of the synchronous parent systems. Such functions, known as refinement maps, are used to compare behaviors of elastic and synchronous systems and hence prove their equivalence. The effectiveness of this method is demonstrated by verifying 8 synchronous elastic pipelined processor models with early evaluation.

Keywords: formal verification, synchronous, elastic system, early evaluation, refinement

ACKNOWLEDGMENTS

I would like to express my deep-felt gratitude to my advisor, Dr. Sudarshan Srinivasan of the Department of Electrical and Computer Engineering at North Dakota State University, for his advice, encouragement, enduring patience and constant support.

I also wish to thank the other members of my committee, Dr. Raj. Katti and Dr. Chao You of the Department of Electrical and Computer Engineering, and Dr. Jing Shi at Department of Industrial and Manufacturing Engineering at North Dakota State University. Additionally, I would also thank Koushik Sarker. Their suggestions, comments and additional guidance were invaluable to the completion of this work.

Thanks Haiping and my brother Yangjun together with sister in-law for their support during my study for a master's degree over the past two years at North Dakota State University.

And finally, I want to thank the professors and staff of the ECE department of North Dakota State University for all their hard work and dedication, providing me the means to complete my master's degree.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

The constant scaling of technology has resulted in increased significance of wire delays in digital design. However, since precise information on wire lengths and delays are available late in the design process, accounting for these delays results in expensive re-designs making synchronous design infeasible for nanoscale systems. Latency insensitive (LI) [1–5, 10, 16–18] design is an emerging solution that addresses the wire delay challenge in the synchronous domain. The central idea is to use relay stations—which function like latches in a pipelined datapath—to break long wires that cause violations of timing requirements imposed by the clock. However, including such relay stations can alter the functionality of the system resulting in an incorrect design. The LI solution to this problem is to use a handshaking protocol known as the LI protocol to control the transfer of data between modules/stages. The protocol allows for insertion of buffers/relay stations without altering the functionality of the system. The resulting Latency Insensitive designs are still synchronous, *i.e.*, the system synchronization is implemented using a global clock. Synchronous Elastic Networks (SEN) [7, 9] is one effective approach to implement LI designs and also synthesize LI systems from synchronous parents.

One of the critical chellenges for any design paradigm to be feasible is verification. The verification challenge for hardware design has in fact been addressed by the International Technology Roadmap for Semiconductors (ITRS) 2007 report [8], which states that "Verification has become the dominant cost in the design process. ... Without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry." The 2007 report of the ITRS also predicts that in 15 years 40% of design errors will be exposed by formal verification methods.

An important component of the verification challenge for LI systems is the verification of pipelined LI systems such as pipelined microprocessors, because, pipelining is a key optimization that is used extensively in digital design at both component and system-levels.

To adress this problem, we develop a verfication solution for LI pipelined systems designed using the SEN approach, referred to as elastic systems hereafter.

Correctness Notion: The goal of the verification solution is to check the correctness of an elastic pipelined system against its synchronous specification. The notion of equivalence that we use is Well Founded Equivalence Bisimulation (WEB) refinement [12], which is based on stuttering bisimulation. Synthesis of elastic designs incorporates the insertion of additional elastic buffers in the data path to handle timing issues in the design. While the insertion of these buffers does not affect the functionality of the system, the timing behavior is altered. As a result, an elastic system can require several transitions to match a single transition of its synchronous parent system. This phenomenon is known as stutter and is accounted for by WEB refinement. We also choose to use WEB refinement as the methods we develop can be combined with existing methods in a compositional manner to verify elastic pipelined systems against high-level non-pipelined specifications, such as an instruction set architecture (ISA) specification.

A detailed description of the theory of refinement can be found in [12]. It is enough to prove the following correctness formula [11] to prove refinement (thereby establish equivalence) between an implementation and its specification.

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

$$
\begin{aligned}
&\langle \forall impl \in \texttt{IMPL} :: \\
&\quad spec = r(impl) \ \wedge \ next\text{-}spec = Sstep(spec) \ \wedge \\
&\quad next\text{-}impl = Istep(impl) \ \wedge \ next\text{-}spec \neq r(next\text{-}impl) \\
&\rightarrow \\
&\quad spec = r(next\text{-}impl) \ \wedge \ rank(next\text{-}impl) < rank(impl) \rangle
\end{aligned}
$$

In the above formula, *impl* is a pipelined machine state. *spec* is the ISA state obtained

by applying the refinement map r to *impl*. *next-spec* is the successor of *spec* obtained by stepping the ISA machine in state *spec* and *next-impl* is the successor of *impl* obtained by stepping the pipeline machine in state *impl*. The formula above states that for every pipelined machine state *impl* its corresponding ISA state *spec*, if the successors of *spec* and *impl*, namely, *next-spec* and *next-impl*, respectively, do not match, then applying r to *next-impl* should result in state *spec* and the rank of *next-impl* should decrease w.r.t. the rank of *impl*. The proof obligation that $spec = r(next\text{-}impl)$ is the safety component and guarantees that if the implementation makes progress, then the result of that progress is correct as given by the specification. However checking safety alone provides no guarantee that the implementation will always make progress, *i.e.*, will not deadlock. The proof obligation that $rank(next\text{-}impl) < rank(impl)$ is the liveness component and guarantees that the machine will not deadlock, *i.e.*, will always make forward progress.

The specific steps involved in a refinement-based verification methodology are: (a) Construct models of the specification and implementation. (b) Compute the states of the implementation model that are reachable from reset (known as reachable states). (c) Construct a refinement map. (d) Construct a *rank* function for the implementation system. (e) The models, the refinement map, and the rank function can now be used to state 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. We use the Bit-level Analysis Tool (BAT) system [13] to model the specification and implementation systems at the word-level and to state the correctness formula. We then use the decision procedure for bit-vectors incorporated in BAT to automatically check the correctness formula. The reachable states, refinement maps, and rank functions are computed manually using the procedures given in the thesis.

1.1. Contributions

There are three main contributions in this thesis. First, we show how to extend the approach presented in [15] to verify elastic pipelined systems with early evaluation [6], a LI design approach that allows a design stage to complete execution when all the data required for proceeding with the computation of the results in that stage are available. Note that if early evaluation is not used, the elastic protocol forces a design stage to wait until all inputs are available, even though some of the inputs may not be required for execution. Second, we describe a systematic procedure for synthesizing rank functions used for deadlock detection for elastic pipelined systems without and with early evaluation. Third, we show how to combine our approach with existing approaches to check verify equivalance of elastic pipelined systems with high-level non-pipelined specifications. The specific contributions of the approach are:

1. A method to compute elastic token-flow diagrams (described in Chapter 3) which can also track the flow and progress of data in elastic pipelined processor designs with early evaluation. These diagrams enables a systematic analysis of data flow in elastic architectures even after the insertion of any number of additional elastic buffers in any place in the data path. A method is provided to construct such diagrams. We also use token-flow diagrams for reachability analysis (described in Chapter 4). The network of elastic controllers is deterministic. However, when early evaluation is introduced the controller network becomes nondeterministic.

2. A method for computing refinement maps based on elastic token-flow diagrams (given in Chapter 5). The complexity in computing refinement maps arises because: (1) Each flip flop in the synchronous machines is replaced with two latches in the elastic machine. Therefore, an elastic storage element can have either 0, 1, or 2 valid data tokens, while in the synchronous system, every flip flop always has exactly one valid data token. (2) The insertion of additional elastic buffers in the design can significantly alter the data flow patterns of the architecture.

3. A method to synthesize rank functions for elastic pipelined systems without and with early evaluation, used to check liveness.

4. We show how to combine our approach with existing approaches to check the equivalance of elastic pipelined systems with early evaluation against high-level non-pipelined specifications.

5. We also show that the elastic token-flow diagrams can be used as a design aid to determine if the design will benefit from early evaluation (shown in section 5.3 of Chapter 5). This analysis is useful as the controllers required for early evaluation are more complex and expensive compared to the controllers required when early evaluation is not implemented.

1.2. Outline of Thesis

This thesis is organized as follows. Chapter 1 provides the background introduction, contribution and outline of the thesis. Chapter 2 describes Synchronous Elstic Network (SEN) and synchronous elastic pipelined processor models with early evaluation. Chapter 3 describes elastic token-flow diagrams which are obtained by simulating the flow of data tokens in the elastic controller network of elastic models with early evaluation. Chapter 4 describes token flow diagram for reachability analysis. Chapter 5 describes refinement maps and rank functions of elastic processor with early evaluation. Chapter 6 provides the results to show the efficient approach for checking the equivalence of elastic pipelined processors with early evaluation against their synchronous parents, also the conclusions and ideas for future research.

CHAPTER 2. ELASTIC PROCESSOR MODELS WITH EARLY EVALUATION

This chapter provides the introduction of elastic processor models based on SEN and early evaluation mechanism.

2.1. Synchronous Elastic Network

SEN is one effective approach to implement LI designs and also synthesize LI systems from synchronous parents. The idea with SEN is to replace each of the flip flops in the design with a latch pair called an elastic buffer (EB). Each EB is clocked using a controller known as the elastic controller, which implements the LI protocol. The controllers are synchronized with a global clock. One controller is used for all the EBs in a stage of the design. The EBs as stated earlier are constructed using two latches, called elastic half buffers (EHBs). The left and the right EHBs are called the master and slave EHBs, respectively. The global clock of the synchronous design is now replaced by a network of elastic controllers. The elastic controllers are connected so that they communicate with their neighbors. In the resulting elasticized design, elastic buffers can be inserted in any place in the datapath to break long wires.

Figure 1 shows an implementation of Elastic Half Buffer, which is a set of elastic modules and elastic channels. Elastic channels have two control wires implementing a handshake between the sender and the receiver. The wires are called valid, in the forward direction, and stop, in the backward direction. The role of these wires is similar to the one of the request/acknowledge wires in asynchronous systems. Depending on the state of the control wires, a channel can carry valid or invalid data items, that we will call tokens and bubbles, respectively. For simplicity in the explanation, we will initially assume that the elastic modules are combinational blocks and latches.

Figure 2 depicts an example of a naive elastic implementation for transmitting data between souce side and destination side of elastic channels. Each register has an associated

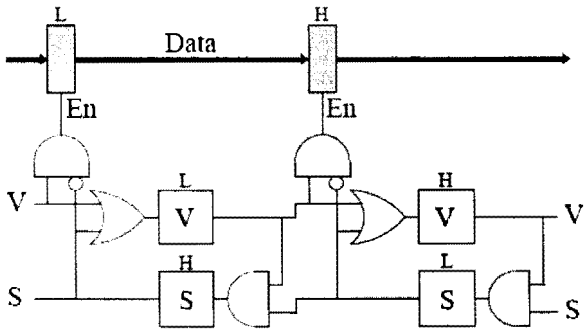

Figure 1: Elastic half buffer

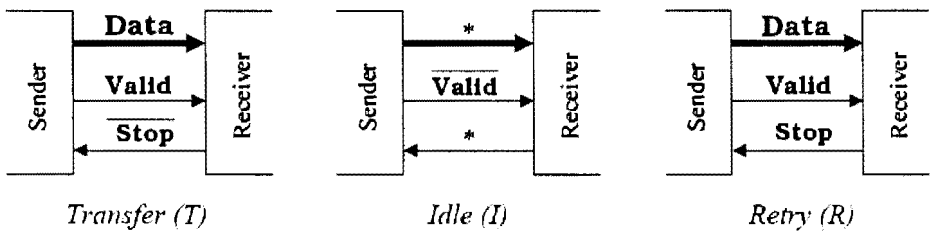

Figure 2: Self protocol

valid bit (V) that keeps track of the validity of the stored data. The clock signal is not explicitly shown and the enable signal (En) indicates when new data is stored into the register. The chain of AND gates manages the backpressure generated by the receiver when it is not able to accept data (Stop = 1). The scheme in Fig 2 is not scalable due to the long combinational path from the receiver to the sender. When the pipeline is full, i.e. all Vs are at 1, the delay of the Stop chain becomes critical.

Data transfer is performed by using the control signals $Valid(V)$ and $Stop(S)$ that determine three possible states in the channel (as shown in Figure 2):

Transfer (T), $(V \wedge \neg S)$: the sender provides valid data and the receiver accepts it.

Idle (I), $(\neg V)$: the sender does not provide valid data.

Retry (R), $(V \wedge S)$: the sender provides valid data but the receiver does not accept it.

Figure 3 shows how the EB interfaces with the source side and destination side

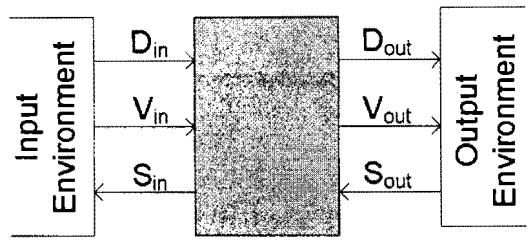

Figure 3: Interface of elastic buffer [7]

elastic channels. Communication between two EBs occurs via a elastic channel, which has 3 components including *data* and *valid* signal produced by the source EB and *stop* signal produced by the destination EB. Data is transferred through the elastic channel only if *valid* is asserted and *stop* is deasserted. We would like to emphasize here again that the transactions in the elastic channels are synchronized with a global clock. This is a significant difference from asynchronous hand-shaking based protocols, where no global clock is present. A detailed description of the elastic controllers can be found in [7, 9].

2.2. Elastic Processor Models with Evaluation

One of the drawbacks in SEN system is that it can often result in an increased number of cycles required by the design to perform operations when introducing additional elastic buffers. Early evaluation is an optimization of elastic design to improve performance [6].

We use the following example to motivate early evaluation. Consider the forwarding logic of the DLX pipeline (shown in Figure 6). The *em* stage receives input from three stages namely *de*, *em*, and *mm*. Let us assume that there are two additional elastic buffers in the path from *mm* to *em*. These additional buffers will result in a degradation of performance and the resulting design will require more clock cycles. But, there could be several situations in which the data from *mm* is not required for the execution in the execute stage to proceed. In elastic designs however, the execution in the execute stage is forced to wait until the data token from *mm* arrives. Early evaluation provides a mechanism to allow

execution to proceed even if the data token from *mm* has not arrived, provided that the it can be determined that *mm* data token is not required using the data from only *de* and *em*.

The mechanism used to implement early evaluation is based on anti-tokens. Supposing that in the execute stage, the token from *mm* has not yet arrived, but it is determined that in fact the token from *mm* is not required for execution. As stated earlier, the early evaluation mechanism allows execution to proceed. The logic to detect and allow execution to proceed is implemented using an eager *join* structure described in [6]. The problem however is that the token from *mm* is still in the path from *mm* to *em*, and has not been consumed yet. Therefore, this token can interfere with future executions in the execute stage. To nullify this *mm* token that has not yet arrived, the eager *join* generates an anti-token. The anti-tokens are propagated in the reverse direction of the tokens and when tokens and anti-tokens meet, they are both nullified. Thus the anti-token generated by the eager *join* in the execute stage will flow in the path from *mm* to *em*, but only now in the reverse direction. When the anti-token meets the token from *mm* both are nullified. As a result, the token from *mm* is prevented from interfering with future executions in the execute stage.

The flow of anti-tokens is implemented using the controllers similar to the elastic controllers (designed to propagate tokens). Only the direction in which the anti-tokens are propagated is reversed (shown in Figure 4). Thus in early evaluation, the elastic controller

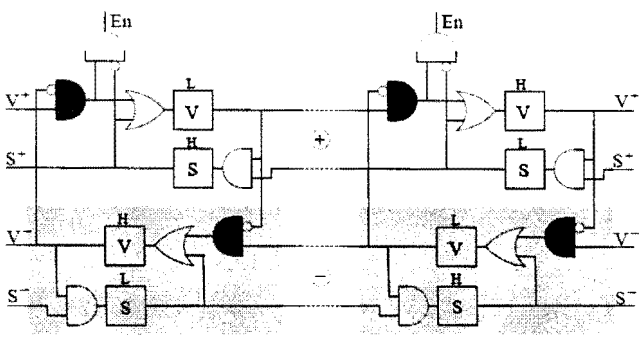

Figure 4: EHB with early evaluation

of an EB can hold up to two tokens or two anti-tokens. The controllers are also modified so that when tokens and anti-tokens meet, both are nullified. Figure 5 shows the interface of the elastic controller with early evaluation, which includes V^+ and S^+ as the output *valid* and *stop* for the tokens, and V^- and S^- as the output *valid* and *stop* for the anti-tokens. Also, V_{in}^+, S_{in}^+, V_{in}^-, and S_{in}^- are the inputs of the controller. The behavior of the controller is complex and we refer the reader to [6] for a detailed description.

We developed 8 elastic processor models (A0, A1, ..., A7) that implement the LI protocol with early evaluation. The models are used to demonstrate the analysis and verification techniques and also used for experimentation. The models are based on the 5-stage DLX model and are described using the BAT specification language at the bit-level with a data path width of 32 bits. A0 is the model without any additional EBs. The other models A1 through A7 were obtained by placing additional EBs at various points in the data path. Figure 6 shows the positions of the additional EBs (I1, I2, I4, and I5). Figure 7 shows the elastic controller network for the processor model given in Figure 6 (which also happens to be model A7. Table 1 shows the additional EBs in each of the models.

Anti-tokens are generated in the execute stage. The execute stage receives input from three stages: execute, memory, and writeback. The paths from memory to execute and writeback to execute are forwarding paths. In the elastic framework, if additional EBs are placed in the forwarding paths, the execute stage will have to wait for data from all three paths before it can complete execution. However, it is possible for the execute stage to complete execution even if it does not receive data from the writeback stage. Such

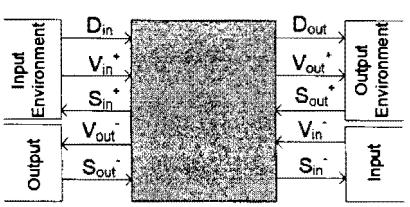

Figure 5: Interface of EB with early evaluation

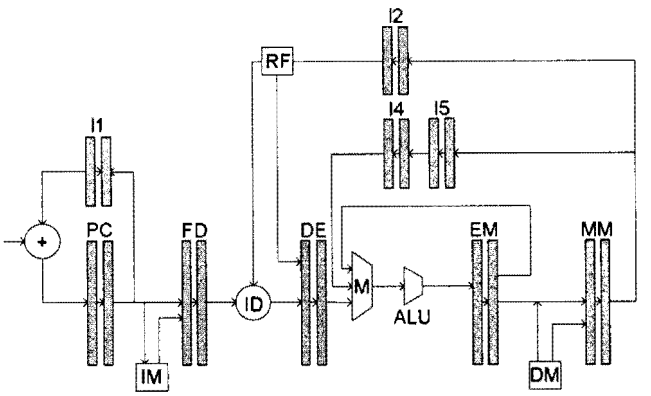

Figure 6: High-level organization of elastic 5-stage DLX processor that implements the early evaluation LI protocol. The model has four additional elastic buffers: 11, 12, 14, and 15

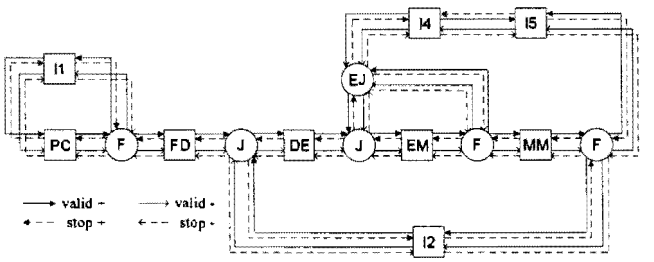

Figure 7: Network of elastic controllers for the elastic 5-stage DLX processor shown in Figure 6. The J and F blocks denote the join and fork circuits. the EJ block denotes the early join circuit, which is capable of generating anti-tokens. Valid+ and Stop+ are the valid and stop signals of the tokens. Valid- and Stop- are the valid and stop signals for the anti-tokens. Note that the directions of Valid- and Stop- are the reverse of the directions of Valid+ and Stop+, respectively.

Table 1: Token-flow Diagram: Reachability

Model	Additional EBs
A0	–
A1	11
A2	14
A3	12, 14
A4	12, 14, 15
A5	11, 14
A6	11, 12, 14
A7	11, 12, 14, 15

a condition occurs when the two operands required for execution are available from the memory stage. This condition can be determined by examining data from only the execute and memory stages. Thus, under this condition, it is not necessary to wait for data from the writeback stage to complete execution. However, the data token from writeback, which will arrive at a later time will have to be nullified. This is achieved by generating an anti-token that will flow from execute to writeback (in the reverse direction). When the anti-token encounters the data token from write back, both are nullified, ensuring that this data token does not interfere with future operations in the execute stage.

The early join block is used to identify the condition under which anti-tokens are required and also generates the required anti-tokens. The exact implementation of the early join block is given in [6]. In the models A0, ..., A7, an early join block is used to join the paths from the memory and writeback to the execute stage and is depicted as EJ in Figure 7. If data from writeback has not arrived and is not require for execution to proceed in the execute stage, then the EJ block generates an anti-token that flows from execute to writeback and also allows the execute stage to complete.

CHAPTER 3. EARLY EVALUATION ELASTIC TOKEN-FLOW DIAGRAM

In this chapter, we show how to construct token-flow diagrams, which are obtained by simulating the flow of data tokens in the elastic controller network, for elastic systems that implement early evaluation. The tokens are assigned numeric labels and the rules of simulation are modified so as to distinguish tokens that correspond to new data units that enter the system, from tokens of data units already present in the processor pipeline. These diagrams make it possible to analyze elastic networks and perform reachability analysis, compute refinement maps, and compute rank functions for elastic processors with early evaluation in a highly automated and sytematic manner.

This approach is similar, a modified and extended set of rules are required to capture the behavior of the elastic controller network [15] (without early evaluation). Two primary differences are: (1) The controller network that implements early evaluation propagates anti-tokens in addition to regular tokens. Also, the anti-tokens propagate in the reverse direction. (2) While the generation of new tokens is a deterministic, the generation of new anti-tokens is nondeterministic w.r.t. to the state of the controller network. Whether anti-tokens are generated or not in a given state of the controller network also depends on the state of the datapath.

The following notation is used to describe the rules. Each controller can hold up to a maximum of two tokens or two anti-tokens. A token or anti-token t is a natural number. A token or anti-token with value 0 corresponds to a bubble. The token diagram is based on the token-state of an early evaluation-based elastic controller network, which is defined as follows.

Definition 2. *The token-state of an early evaluation elastic controller network with n controllers is an n-tuple, where each element is a quad of the form* $\langle t_m, t_s, a_m, a_s \rangle$, *where* $t_m, t_s, a_m, a_s \in \mathbb{N}$.

If e is an early evaluation elastic controller, then $t_m.e$ and $t_s.e$ are the tokens in the master EHB and the slave EHB, respectively, and $a_m.e$ and $a_s.e$ are the anti-tokens in the master and slave EHB, respectively. $t'_m.e$, $t'_s.e$, $a'_m.e$ and $a'_s.e$ are the master EHB token, slave EHB token, master EHB anti-token, and slave EHB token, respectively, in the next state of the controller. $v_{inp}.e$ and $s_{inp}.e$ are the current values of the input valid and input stop signals for tokens of e. Similarly, $v_{inm}.e$ and $s_{inm}.e$ are the current values of the input valid and input stop signals for anti-tokens of e. The values for the valid and stop signals can be obtained from the circuit of the early evaluation elastic controller network. t_a and t_b are valid tokens, while a_a and a_b are valid anti-tokens. A token with a zero value corresponds to a bubble. The *new-token* function defined in Section 3 is used to update elastic controllers with tokens. For a given controller, a_i is its input anti-token. We use a global counter to generate new anti-tokens. The counter is initialized to 1. The function *new-anti-token* will use the current value of the counter and as the new anti-token, and will also update the counter for future usage. This ensures that no two anti-tokens generated will have the same value.

Rules 1 through 3 describe the transitions of the token-state of the controller from the state in which the controller does not have any tokens or anti-tokens.

1. $(t_m.e = 0 \wedge t_s.e = 0 \wedge a_m.e = 0 \wedge a_s.e = 0 \wedge ((\neg v_{inp}.e \wedge \neg v_{inm}.e) \vee (v_{inp}.e \wedge v_{inm}.e))) \rightarrow$

   $(t'_m.e = 0 \wedge t'_s.e = 0 \wedge a'_m.e = 0 \wedge a'_s.e = 0)$

2. $(t_m.e = 0 \wedge t_s.e = 0 \wedge a_m.e = 0 \wedge a_s.e = 0 \wedge (v_{inp}.e \wedge \neg v_{inm}.e)) \rightarrow$
   $(t'_m.e = 0 \wedge t'_s.e = \textit{new-token}(e, t_i) \wedge a'_m.e = 0 \wedge a'_s.e = 0)$

3. $(t_m.e = 0 \wedge t_s.e = 0 \wedge a_m.e = 0 \wedge a_s.e = 0 \wedge (\neg v_{inp}.e \wedge v_{inm}.e)) \rightarrow$
   $(t'_m.e = 0 \wedge t'_s.e = 0 \wedge a'_m.e = 0 \wedge a'_s.e = a_i)$

Rules 4 through 7 describe the transitions of the token-state of the controller from the

10. $(t_m.e = 0 \wedge t_s.e = 0 \wedge a_m.e = 0 \wedge a_s.e = a_a \wedge (\neg v_{inp}.e \neg v_{inm}.e)) \rightarrow$
$(t'_m.e = 0 \wedge t'_s.e = 0 \wedge a'_m.e = 0 \wedge a'_s.e = a_a)$

11. $(t_m.e = 0 \wedge t_s.e = 0 \wedge a_m.e = 0 \wedge a_s.e = a_a \wedge (\neg v_{inp}.e \wedge v_{inm}.e \wedge s_{inm}.e)) \rightarrow$
$(t'_m.e = 0 \wedge t'_s.e = 0 \wedge a'_m.e = a_i \wedge a'_s.e = a_a)$

12. $(t_m.e = 0 \wedge t_s.e = 0 \wedge a_m.e = 0 \wedge a_s.e = a_a \wedge (v_{inp}.e \wedge \neg v_{inm}.e)) \rightarrow$
$(t'_m.e = 0 \wedge t'_s.e = 0 \wedge a'_m.e = 0 \wedge a'_s.e = 0)$

13. $(t_m.e = 0 \wedge t_s.e = 0 \wedge a_m.e = 0 \wedge a_s.e = a_a \wedge ((v_{inp}.e \wedge v_{inm}.e) \vee (\neg v_{inp}.e \wedge v_{inm}.e \wedge$
$\neg s_{inm}.e))) \rightarrow$
$(t'_m.e = 0 \wedge t'_s.e = 0 \wedge a'_m.e = 0 \wedge a'_s.e = a_i)$

Rules 14 and 15 describe the transitions of the token-state of the controller from the state in which the controller has no tokens and two anti-tokens.

14. $(t_m.e = 0 \wedge t_s.e = 0 \wedge a_m.e = a_b \wedge a_s.e = a_a \wedge (\neg v_{inp}.e \wedge s_{inm}.e)) \rightarrow$
$(t'_m.e = 0 \wedge t'_s.e = 0 \wedge a'_m.e = a_b \wedge a'_s.e = a_a)$

15. $(t_m.e = 0 \wedge t_s.e = 0 \wedge a_m.e = a_b \wedge a_s.e = a_a \wedge \neg(\neg v_{inp}.e \wedge s_{inm}.e)) \rightarrow$
$(t'_m.e = 0 \wedge t'_s.e = 0 \wedge a'_m.e = 0 \wedge a'_s.e = a_a)$

The join block has two input tokens on the input side and one input anti-token on the output side. The join cannot generate anti-tokens. Rules 16 through 22 describe its behavior. Note that the join behaves as a join for tokens and as a fork for anti-tokens because anti-tokens flow in the opposite direction. The output of the join is a three-tuple, the first is the output token on the output side and the next two elements are the output anti-tokens on the input side. The *max* function computes the maximum of two input token or anti-token values. Note that rules 22 and 23 describe the situation where there is only one input token, in which case the token is not allowed to pass.

16. $J(t_a, t_b, 0) \rightarrow \langle max(t_a, t_b), 0, 0 \rangle$

17. $J(t_a, t_b, a_a) \rightarrow \langle 0, 0, 0 \rangle$

18. $J(0, 0, a_a) \rightarrow \langle 0, a_a, a_a \rangle$

19. $J(t_a, 0, a_a) \rightarrow \langle 0, 0, a_a \rangle$

20. $J(0, t_a, a_a) \rightarrow \langle 0, a_a, 0 \rangle$

21. $J(t_a, 0, 0) \rightarrow \langle 0, 0, 0 \rangle$

22. $J(0, t_a, 0) \rightarrow \langle 0, 0, 0 \rangle$

The fork block has two input anti-tokens on the output side and one input token on the input side. The fork cannot generate anti-tokens. Rules 23 through 29 describe its behavior. Note that the fork behaves as a join for anti-tokens and as a fork for tokens because anti-tokens flow in the opposite direction. The output of the fork is a three-tuple, the first is the output anti-token on the input side and the next two elements are the output tokens on the output side. Note that rules 28 and 29 describe the situation where there is only one input anti-token, in which case the anti-token is not allowed to pass.

23. $F(a_a, a_b, 0) \rightarrow \langle max(a_a, a_b), 0, 0 \rangle$

24. $F(a_a, a_b, t_a) \rightarrow \langle 0, 0, 0 \rangle$

25. $F(0, 0, t_a) \rightarrow \langle 0, t_a, t_a \rangle$

26. $F(a_a, 0, t_a) \rightarrow \langle 0, 0, t_a \rangle$

27. $F(0, a_a, t_a) \rightarrow \langle 0, t_a, 0 \rangle$

28. $F(a_a, 0, 0) \rightarrow \langle 0, 0, 0 \rangle$

29. $F(0, a_a, 0) \rightarrow \langle 0, 0, 0 \rangle$

The early join is similar to the join, the primary difference being that it can generate anti-tokens. In fact, the early join follows rules 16 through 20. The difference occurs when only one token is input and there are no anti-token inputs. In such a situation, the early join may or may not generate and anti-token and both possibilities should be considered. Rules 30 through 33 describe the behavior of the early join in this situation.

30. $EJ(t_a, 0, 0) \rightarrow \langle 0, 0, 0 \rangle$

31. $EJ(t_a, 0, 0) \rightarrow \langle 0, 0, \textit{new-anti-token} \rangle$

32. $EJ(0, t_a, 0) \rightarrow \langle 0, 0, 0 \rangle$

33. $EJ(0, t_a, 0) \rightarrow \langle 0, \textit{new-anti-token}, 0 \rangle$

CHAPTER 4. REACHABILITY ANALYSIS FOR ELASTIC SYSTEM WITH EARLY EVALUATION

For elastic controllers without early evaluation mechanism, its controller network is a deterministic system, which means all the next-states are deterministic, and the reachability analysis can be performed by simulating the elastic network starting from an inital state until a convergence is reached, *i.e.*, a state of the controller is reached that has already been visited as depicted in Figure 8. As shown in the figure, S_0 is the initial state. When the elastic controller network transitions from S_r, it goes back to S_k again, which is a state that has already been visited. Therefore the reachable sates of the contoller network are S_0 through S_r.

In the reset state of an in-order pipelined processor, all the pipeline latches are actually empty and do not contain any valid instructions. This is implemented by resetting the valid bits in the pipeline latches. However, the elastic controllers without early evaluation corresponding to the pipeline latches are initialized to the half state, *i.e.*, with one token and the controllers of the additional elastic buffers are initialized to the empty state. Such an initialization is required as the presence of these tokens enables data flow in the elastic system. Also note that once states S_0, ..., S_r are computed using the above initialization, states S_0, ..., S_{k-1} can be dropped from the set of reachable states as for all practical purposes the actual elastic system can be initialized to any one of the states S_k, ..., S_r in which the controller of the program counter has at least one valid token.

For the purpose of reachability analysis, we define the elastic-state of an elastic controller network as follows.

Definition 3. *The elastic-state of an elastic controller network with n controllers is an n-tuple, where each element is either empty, half, or full.*

An elastic-state of a controller network can be easily constructed from its token state by observing the number of valid (non-zero) tokens in each controller. We assume that

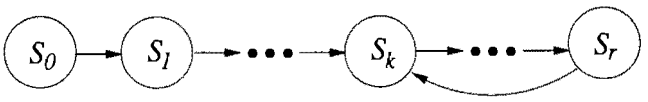

Figure 8: Reachability analysis of elastic controller networks

the *elastic-state* function performs such a construction. Reachability analysis is performed using the following procedure.

(1) Let EC_p be the ordered set of elastic controllers corresponding to the pipeline latches (including the program counter) in the elastic processor. The position of a controller in EC_p is based on its position in the pipeline and is given by the *pos* function. Let EC_a be the set of additional elastic controllers in no particular order. For the DLX example $EC_p = \{pc, fd, de, em, mm\}$, and $EC_a = \{l1, l2, l4, l5\}$. Then, the initial token state S_0^t is given using the following assignments to EC_p and EC_a: (a) $\langle \forall e \in EC_p :: (t_m.e = 0) \wedge (t_s.e = n + 1 - pos(e, EC_p)) \rangle$. (b) $\langle \forall e \in EC_a :: (t_m.e = 0) \wedge (t_s.e = 0)$. The tokens corresponding to memory components are initialized to the t_s value of the pipeline latch at the end of the pipeline stage in which they are updated. For the DLX example in our experiments, the memory components are the register file (*rf*) and the data memory (*dm*), which are initialized to $t_s.de$ and $t_s.mm$.

(2) The set of visited states V is initialized to $\{elastic\text{-}state(S_0^t)\}$. Initialize the current token-state S_c^t to S_0^t and the loop counter i to 0.

(3) Compute the next token-state S_{i+1}^t from S_i^t using the elastic token-flow procedure given in Chapter 3.

(4) If $elastic\text{-}state(S_{i+1}^t) \in V$, then terminate. Otherwise update $V = V \cup \{elastic\text{-}state(S_{i+1}^t)\}$.

(5) Increment i and goto step 3.

As for elastic controller network that incorporates early evaluation mechanism, its reachable states can also be performed using elastic token-flow diagrams as described (which is originally for elastic controller network without early evaluation) above with

two primary differences. First of all, the rules which are used to construct the elastic token-flow diagrams for elastic controller network with early evaluation are those described in Chapter 3. Secondly, the generation of anti-tokens is dependent on the state of the data path as well as the state of the elastic controller with early evaluation. Since the states of the data path could be extremely large, they are not used in the reachability analysis of the controller network. Instead we consider both situations. If it is possible for an anti-token to be generated, we consider the cases where the anti-token is and is not generated. The generation of anti-tokens is given by rules 30 through 33 in Chapter 3. Note that rules 30 and 31 fire simultaneously and similarly rules 32 and 33 fire simultaneously.

Figure 9 shows the reachable states and transitions of the elastic controller network of the A4 model. The elastic token-flow digram which are used to compute for reachable states of the A4 model is shown in Table 2. The initial state S_0 is initialized with tokens as described in Chapter 4. All anti-tokens are initialized to bubbles. In the A4 example, anti-tokens, when generated are usually nullified immediately by the corresponding tokens in the opposite direction. Therefore, their presence is not seen in this token-flow diagram except in state S_1, since only the S_1 state has anti-token and no tokens at the same time. Therefore, the table only shows tokens, except in the S_1 state where the anti-token is indicated by the $\ominus$ symbol.

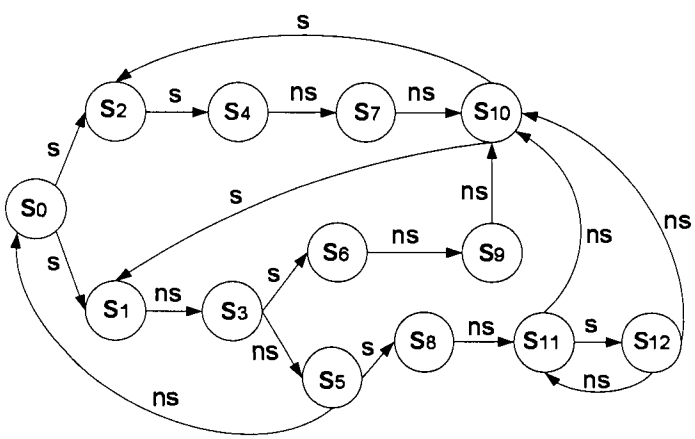

Figure 9: Reachability analysis for A4 model

Table 2: Token-flow Diagram for Early Evaluation Elastic: Reachability

State	pc	fd	de	em	mm	l2	l4	l5
S_0	7	6	5	4	3	0	0	0
S_1	8	7,6	0	5	4	3	$0/\ominus$	3
S_2	8	7,6	5	4	4	3	0	3
S_3	9,8	7	6	5	5	4	0	4
S_4	9,8	7	6,5	4	0	4	3	4
S_5	9	8	7	6	0	5	0	5
S_6	9	8	7,6	5	0	5	4	5
S_7	9	8,7	6	5	0	4	4	0
S_8	10	9	8,7	6	6	0	5	0
S_9	10	9,8	7	6	0	5	5	0
S_{10}	10,9	8	7	6	5	0	0	0
S_{11}	11	10,9	8	7	0	6	0	6
S_{12}	12,11	10	9,8	7	7	0	6	0

CHAPTER 5. REFINEMENT MAPS FOR ELASTIC SYSTEM WITH EARLY EVALUATION

5.1. Refinement Maps

Verifying that the elastic implementation refines its synchronous counterpart requires a function that maps states of the elastic system to states of the synchronous parent system. This function, known as the refinement map, can be thought of as an abstraction function that allows one to view an elastic system as a synchronous system. We introduce a procedure to compute such refinement maps for elastic pipelined processors.

In elastic systems, some inputs can take several cycles to reach their destination stage. Whereas, in synchronous systems, inputs are available to each pipeline stage at every cycle. This variability in the latency of inputs in elastic systems is identified by bubbles (tokens with a 0 value) in the elastic token-flow diagrams. If we were to construct the token-flow diagram for a synchronous machine, there would be no pipeline latch with bubbles, nor would there be pipeline latches with two tokens. Also, a new token will be introduced at every step. A synchronous token-state for a pipeline with n-stages and m memory components would be an $n+m$-tuple with one token for each latch and one token for each memory component. In the synchronous state, memory components tokens are assigned the token of the pipeline latch at the end of the pipeline stage in which they are updated. For the 5-stage synchronous DLX, a token state would be of the form ⟨pc, fd, de, em, mm, rf, dm⟩ and three possible successive token states for the synchronous DLX would be ⟨5, 4, 3, 2, 1, 4, 1⟩, ⟨6, 5, 4, 3, 2, 5, 2⟩, and ⟨7, 6, 5, 4, 3, 6, 3⟩. Herein lies the usefulness of token-flow diagrams as they clearly bring out the differences in data-flow between the synchronous and elastic systems.

One approach to define the refinement map is to roll back some or all the pipeline latches and memory components in an elastic state so that all the latches including the

program counter are in a half state and if t_{pc} is the token of the program counter, then the tokens of the other $n-1$ latches will have the following values: $\langle \forall e \in EC_p :: t_m.e = 0 \wedge t_s.e = t_{pc} + 1 - pos(e, EC_p) \rangle$ Projecting out the values of the slave elastic half buffers corresponding to the pipeline latches will give the corresponding synchronous state. Such an approach is similar to the commitment refinement map used for synchronous processor verification [11]. The extent to which each latch is rolled back depends on the state of the elastic controller network. Therefore, we define one mapping function for each reachable state of the elastic controller network. The overall refinement map selects and applies the appropriate mapping function for each controller network state. Given an elastic controller network state (S_r), the procedure to compute the mapping function is as follows.

1. Count the number of pipeline latches that are in the empty state in S_r. Let this count be n_e. Let r be the number of reachable states of the controller.

2. Starting from a token-state of S_r (such a state can be obtained from the token-flow diagram constructed for reachability analysis), construct the token-flow diagram for $n_e * r$ steps. This provides sufficient steps of the token-flow diagram to perform the analysis required to compute the mapping function.

3. Starting from the last token state in the diagram, search backwards in the *pc* column to find the first valid token, say t_{pc}. Construct a synchronous token-state corresponding to S_r using the following equation: $t_j = t_{pc} + 1 - j$, where j is the position of the latch in the pipeline, with $j = 1$ for the *pc*. Memory components would be assigned the token of the stage in which they are updated.

4. Rolling back the pipeline latches is hard to compute directly. We instead use history variables that record previous values of pipeline latches. Therefore, all that is to be determined to construct the mapping function is which history should be projected. This is computed by searching backward in each of the columns in the token-flow

diagram (where each column corresponds to a pipeline latch), to find the first token that matches the token in the synchronous token-state. If the match for a pipeline latch was found k rows going backward, then the k^{th} history variable is projected for that latch.

Note that anti-tokens are ignored in refinement map computation. Examples of computing refinement maps for some states of the A4 model are shown in Tables 3, 4, and 5. The circled tokens indicate when a match is found.

5.2. Liveness

As described before, checking for liveness is used to determine if the implementation (the elastic system) always makes forward progress, *i.e.*, it does not deadlock. This is achieved by defining *rank*, a function from states of the implementation to the natural numbers whose value decreases when the implementation stutters w.r.t. the specification. Another way to think of the rank function is that it is used to distinguish stutter from deadlock. During both stutter and deadlock, the implementation does not seem to make progress w.r.t. the specification. If the lack of progress is due to stutter, the rank function identifies this by decreasing. If the lack of progress is because of deadlock, the rank function will not decrease pointing to a deadlock problem. We now describe a method that can be used to synthesize rank functions for elastic systems with and without early evaluation.

The high-level idea is as follows. Remember that the implementation is the pipelined

Table 3: Token-flow Diagram: Refinement Map Construction for S_0 of Processor A4

State	pc	fd	de	em	mm	rf	dm
S_c	⑦	⑥	⑤	④	③	⑤	③
Sync	7	6	5	4	3	5	3

Table 4: Token-flow Diagram: Refinement Map Construction for S_1 of Processor A4

State	pc	fd	de	em	mm	rf	dm
S_{c-1}	⑦	6	⑤	④	③	5	③
S_c	8	7,⑥	0	5	4	⑤	4
Sync	7	6	5	4	3	5	3

Table 5: Token-flow Diagram: Refinement Map Construction for S_2 of Processor A4

State	pc	fd	de	em	mm	rf	dm
S_{c-1}	⑦	6	5	4	③	5	③
S_c	8	7,⑥	⑤	④	4	⑤	4
Sync	7	6	5	4	3	5	3

elastic system and the specification is the pipelined synchronous system. Therefore, the only reason for stutter between the implementation and specification is because of the elastic nature of the implementation. The transitions of the elastic implementation that will result in stutter can be determined by analyzing the elastic controller network.

The state diagram of the elastic controller network can be determined using elastic token-flow diagrams. Using the refinement map computations, we can determine which are the stuttering steps of the controller and which are the non-stuttering steps.

We then label the edges in the state diagram of the elastic controller network as "s" or "ns" if the edge corresponds to a stuttering step or non-stuttering step of the implementation. We then assign a rank (a natural number value) to all the states in the state diagram such that rank value decreases along a stuttering edge. The detailed procedure for synthesizing rank functions is described below. Note that the procedure can be applied to elastic systems with and without early evaluation. When early evaluation is not used, the elastic controller network is deterministic and its state diagram is just a loop. If early evaluation is present, the state diagram is more complex and can consist of multiple

overlapping loops. Figure 9 shows the state diagram of the A4 elastic processor including the stuttering edges and rank values for each of the states. The detailed method for rank computation is given below using two procedures. The first procedure is used to compute the stuttering transitions of an SEC. The input to the procedure is the State diagram of the Elastic Controller network (SEC) of the implementation machine.

Procedure for computing stuttering transitions of SEC:

1. Repeat steps 2 through 5 for each of the transitions (edges) in the SEC, which are all of the form $w \dashrightarrow v$.
2. Determine the token-state of w from the elastic token-flow diagram constructed for reachability analysis of the implementation system.
3. Compute the token-state of v from the token-state of w using the rules for elastic token-flow diagrams.
4. Apply the refinement map to both w and v to determine their corresponding synchronous token-states w_s and v_s, respectively.
5. If $w_s = v_s$, then mark the transition (edge in the SEC) as "s", else mark it as "ns".

Procedure for ranking states of SEC:

1. For each of the loops in the SEC, repeat steps 2 through 4.
2. Let p be the number of states in the loop and p_s be the stuttering transitions in the loop.
3. Choose any state w in the loop that has an incoming edge with an "ns" label. Assign the rank value of p to w.

4. Starting from the successor of w, assign the ranks of all the other states in the loop in the following way. If a state has an incoming edge with an s or ns label and its predecessor has a rank value of q, then assign the state a rank value of q-1 or q, respectively. Note that this approach ensures that the last transition in the loop from the predecessor of w to w will result in an increase in the value of rank from p-p_s to p. Also, for all stuttering transitions in the loop, the rank value will decrease by one and for all other non-stuttering transitions in the loop, the rank will remain the same.

5. The next issue is that the loops in the SEC will have overlapping states. Thus, some states will have multiple rank values. If a state v has more than on rank assigned, the highest rank is chosen. Also, the ranks of the states in the loops that did not assign v its highest rank will have to be adjusted by incrementing the ranks of the states in these loops with the difference of the highest rank of state v and the rank assigned by that loop to state v. This situation will not occur in systems that do not use early evaluation, because, the controller network is deterministic and so the SEC will have only one loop.

The rank of an elastic system state is given by the rank of the state of its elastic controller network.

5.3. Using Token-flow Diagrams for Elastic System Design

The reachability analysis performed using token-flow diagrams can be used for design of the early evaluation elastic system. In performing reachability analysis of various elastic processor models with anti-tokens, we found several of these designs that seem to benefit from early evaluation, actually do not. Consider the example of the A5 processor model. The reachability analysis of the A5 model is shown in Figure 10.

If the A5 elastic controller is initialized to the s_{11} or s_{12} state, then the controller will remain only in one of these states and no anti-tokens will be generated and therefore, early

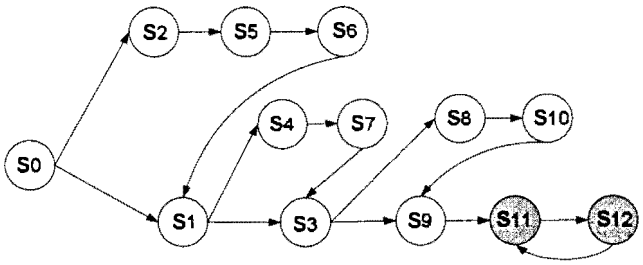

Figure 10: Reachability analysis for A5 model

evaluation is not required. Thus, if the A5 model is initialized to s_{11} or s_{12} states, then the more complex controllers required for early evaluation can be replaced with the basic elastic controllers. Two more examples that illustrate the use of the token-flow diagrams to eliminate early evaluation are models A6 and A7. The corresponding reachability analysis of these models are shown in Figures 11 and 12. Note that in all these three figures, the reachable states are shown as shaded circles.

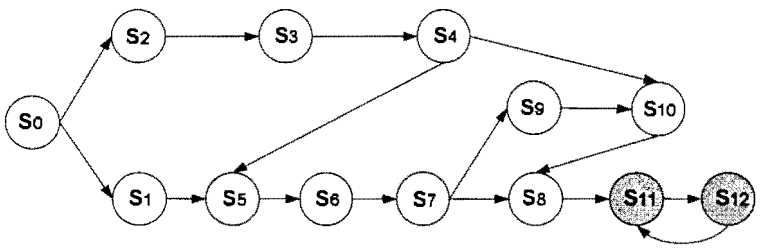

Figure 11: Reachability analysis for A6 model

Table 6 and Table 7shows the invariant constraint and mapping, rank functions for each of the reachable states of the elastic processors A0, A1, A2, A3, A4, A5, A6, A7. For anti-models A0 and A1, no anti-tokens be generated, since the condition for generating anti-tokens is never satisfied. In the invariant constraint column, if x is a pipeline latch, x^0, $x^{\frac{1}{2}}$, x^1, $x^{-\frac{1}{2}}$ and x^{-1} are used to indicate the empty/empty, half/empty, full/empty, empty/half and empty/full states of the latch, respectively. For the refinement map column in the table, we use the following notation. If x is a pipeline latch, x_y^h is used to indicate

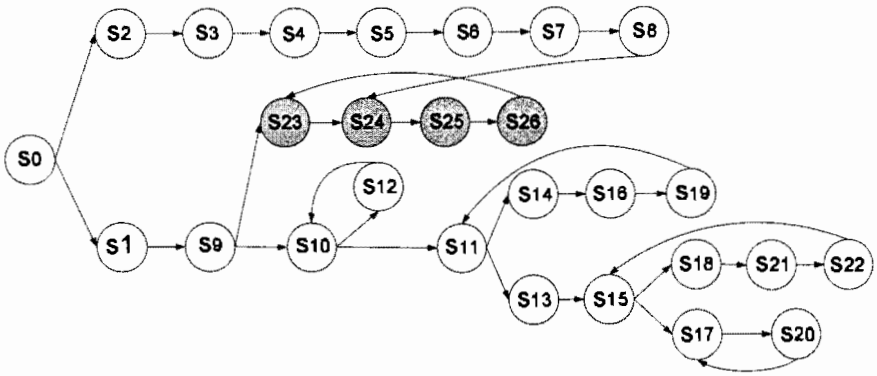

Figure 12: Reachability analysis for A7 model

the projected value for that latch, where h indicates the history value (0 for current, -1 for previous value, -2 for the value two cycles before, and 1 for the next value). y can either be s or m indicating that the projected value is from the slave EHB or the master EHB, respectively. The invariant constraints and mapping functions were obtained using the procedures described in Chapter 4 and Chapter 5.

Table 6: Invariants for Elastic Processor Models with Early Evaluation

Processor Model	Controller State	Invariant Constraint
A0	Invariant-1	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}}$
A1	Invariant-1	$pc^{\frac{1}{2}} \wedge fd^{0} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l1^{0}$
	Invariant-2	$pc^{0} \wedge fd^{\frac{1}{2}} \wedge de^{0} \wedge em^{\frac{1}{2}} \wedge mm^{1} \wedge l1^{\frac{1}{2}}$
A2	Invariant-1	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l4^{0}$
	Invariant-2	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{1} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$
	Invariant-3	$pc^{\frac{1}{2}} \wedge fd^{1} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$
	Invariant-4	$pc^{1} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l4^{0}$
A3	Invariant-1	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{0} \wedge l4^{0}$
	Invariant-2	$pc^{\frac{1}{2}} \wedge fd^{1} \wedge de^{0} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{\frac{1}{2}} \wedge l4^{0}$
	Invariant-3	$pc^{1} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$
	Invariant-4	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$
	Invariant-5	$pc^{\frac{1}{2}} \wedge fd^{1} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$
	Invariant-6	$pc^{1} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$
A4	Invariant-1	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{0} \wedge l4^{0} \wedge l5^{0}$
	Invariant-2	$pc^{\frac{1}{2}} \wedge fd^{1} \wedge de^{0} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{\frac{1}{2}} \wedge l4^{-\frac{1}{2}} \wedge l5^{\frac{1}{2}}$
	Invariant-3	$pc^{\frac{1}{2}} \wedge fd^{1} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{\frac{1}{2}} \wedge l4^{0} \wedge l5^{\frac{1}{2}}$
	Invariant-4	$pc^{1} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{\frac{1}{2}} \wedge l4^{0} \wedge l5^{\frac{1}{2}}$
	Invariant-5	$pc^{1} \wedge fd^{\frac{1}{2}} \wedge de^{1} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}} \wedge l5^{\frac{1}{2}}$
	Invariant-6	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{0} \wedge l5^{\frac{1}{2}}$
	Invariant-7	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{1} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}} \wedge l5^{\frac{1}{2}}$
	Invariant-8	$pc^{\frac{1}{2}} \wedge fd^{1} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}} \wedge l5^{0}$
	Invariant-9	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{1} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{0} \wedge l4^{\frac{1}{2}} \wedge l5^{0}$
	Invariant-10	$pc^{\frac{1}{2}} \wedge fd^{1} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}} \wedge l5^{0}$
	Invariant-11	$pc^{1} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{0} \wedge l4^{0} \wedge l5^{0}$
	Invariant-12	$pc^{\frac{1}{2}} \wedge fd^{1} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{0} \wedge l5^{\frac{1}{2}}$
	Invariant-13	$pc^{1} \wedge fd^{\frac{1}{2}} \wedge de^{1} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{0} \wedge l4^{\frac{1}{2}} \wedge l5^{0}$
A5	Invariant-1	$pc^{\frac{1}{2}} \wedge fd^{0} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l1^{0} \wedge l4^{\frac{1}{2}}$
	Invariant-2	$pc^{0} \wedge fd^{\frac{1}{2}} \wedge de^{0} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l1^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$
A6	Invariant-1	$pc^{\frac{1}{2}} \wedge fd^{0} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l1^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$
	Invariant-2	$pc^{0} \wedge fd^{\frac{1}{2}} \wedge de^{0} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l1^{\frac{1}{2}} \wedge l2^{1} \wedge l4^{\frac{1}{2}}$
A7	Invariant-1	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l1^{0} \wedge l2^{0} \wedge l4^{\frac{1}{2}} \wedge l5^{0}$
	Invariant-2	$pc^{0} \wedge fd^{1} \wedge de^{0} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l1^{\frac{1}{2}} \wedge l2^{0} \wedge l4^{0} \wedge l5^{0}$
	Invariant-3	$pc^{\frac{1}{2}} \wedge fd^{1} \wedge de^{0} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l1^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{0} \wedge l5^{\frac{1}{2}}$
	Invariant-4	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{0} \wedge l1^{\frac{1}{2}} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}} \wedge l5^{\frac{1}{2}}$

Table 7: Refinement Maps and Rank functions for Elastic Processor Models with Early Evaluation

Processor Model	Controller State	Refinement Map							Rank
		pc	fd	de	em	mm	rf	dm	
A0	Invariant-1	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^0	rf^0	dm^0	-
A1	Invariant-1	pc_s^0	fd_s^{-1}	de_s^{-1}	em_s^{-1}	mm_s^{-1}	rf^{-1}	dm^{-1}	0
	Invariant-2	$l1_s^1$	fd_s^0	de_s^{-1}	em_s^{-1}	mm_s^0	rf^{-1}	dm^{-1}	1
A2	Invariant-1	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^0	rf^0	dm^0	4
	Invariant-2	pc_s^{-1}	fd_s^{-1}	de_s^0	em_s^0	mm_s^{-1}	rf^{-1}	dm^{-1}	3
	Invariant-3	pc_s^{-1}	fd_s^0	de_s^0	em_s^0	mm_s^0	rf^0	dm^0	4
	Invariant-4	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^0	rf^0	dm^0	4
A3	Invariant-1	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^0	rf^0	dm^0	4
	Invariant-2	pc_s^{-1}	fd_s^0	de_s^{-1}	em_s^{-1}	mm_s^{-1}	rf^0	dm^{-1}	3
	Invariant-3	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^{-1}	4
	Invariant-4	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^0	4
	Invariant-5	pc_s^{-1}	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^{-1}	3
	Invariant-6	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^{-1}	4
A4	Invariant-1	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^0	rf^0	dm^0	20
	Invariant-2	pc_s^{-1}	fd_s^0	de_s^{-1}	em_s^{-1}	mm_s^{-1}	rf^0	dm^{-1}	19
	Invariant-3	pc_s^{-1}	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^{-1}	19
	Invariant-4	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^{-1}	19
	Invariant-5	pc_s^{-2}	fd_s^{-1}	de_s^0	em_s^0	mm_s^{-2}	rf^{-1}	dm^{-2}	18
	Invariant-6	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^0	19
	Invariant-7	pc_s^{-1}	fd_s^{-1}	de_s^0	em_s^0	mm_s^{-2}	rf^{-1}	dm^{-2}	18
	Invariant-8	pc_s^{-1}	fd_s^0	de_s^0	em_s^0	mm_s^{-2}	rf^0	dm^0	18
	Invariant-9	pc_s^{-1}	fd_s^{-1}	de_s^0	em_s^0	mm_s^{-2}	rf^{-1}	dm^{-1}	18
	Invariant-10	pc_s^{-1}	fd_s^0	de_s^0	em_s^0	mm_s^{-2}	rf^0	dm^0	18
	Invariant-10	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^0	rf^0	dm^0	20
	Invariant-12	pc_s^{-1}	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^0	18
	Invariant-13	pc_s^{-2}	fd_s^{-1}	de_s^0	em_s^0	mm_s^{-2}	rf^{-1}	dm^{-1}	17
A5	Invariant-1	pc_s^0	fd_s^{-1}	de_s^{-2}	em_s^{-2}	mm_s^{-2}	rf^{-1}	dm^{-2}	1
	Invariant-2	pc_s^{-1}	fd_s^{-2}	de_s^{-3}	em_s^{-3}	mm_s^{-3}	rf^{-2}	dm^{-3}	0
A6	Invariant-1	pc_s^0	fd_s^{-1}	de_s^{-2}	em_s^{-2}	mm_s^{-4}	rf^{-1}	dm^{-3}	1
	Invariant-2	pc_s^{-1}	fd_s^{-2}	de_s^{-3}	em_s^{-3}	mm_s^{-5}	rf^{-2}	dm^{-4}	0
A7	Invariant-1	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^{-2}	rf^0	dm^0	3
	Invariant-2	pc_s^{-1}	fd_s^0	de_s^{-1}	em_s^{-1}	mm_s^{-3}	rf^0	dm^{-1}	2
	Invariant-3	pc_s^{-2}	fd_s^0	de_s^{-2}	em_s^{-2}	mm_s^{-4}	rf^0	dm^{-2}	1
	Invariant-4	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^{-2}	rf^0	dm^{-2}	4

CHAPTER 6. RESULTS AND FUTURE WORK

6.1. Results

The verification results are shown in Table 8. The refinement proofs were automatically checked using the BAT decision procedure version 0.2 [13]. The Siege SAT solver (version 4) was used to solve the SAT problems generated by BAT [14]. The experiments were conducted on a 1.8GHz Intel (R) Core (TM) Duo CPU, with an L1 cache size of 2048KB. In the table, the "Siege" column gives the running times of the Siege SAT solver, which is used to as used As can be seen from the table, all the elastic 5-stage DLX-based processors were verified against the synchronous DLX within 1000 seconds.

Table 8: Verification Times and CNF Statistics for Anti Model

Processor Models	Verification Time [sec]		CNF Statistics		
	Siege	Total (BAT)	Variables	Clauses	Literals
A0	0.03	0.64	1,947	5,575	34,208
A1	11.72	14.73	7,191	29,949	180,830
A2	112.46	127.67	9.193	41,697	364,794
A3	65.48	57.73	6,472	27,643	302,561
A4	844.19	855.87	13.646	87.953	1730,012
A5	18.70	23.22	13,479	64,373	373,418
A6	62.49	65.96	13.743	65,229	370,418
A7	68.86	71.24	20.069	107,029	631,014

6.2. Future Work

Refinement-based verification requires a refinement map that relates states of the implementation and states of the specification. Defining efficient refinement maps often requires intuition about the design. We develop a methodology to compute refinement maps and rank functions in a highly automated manner for checking the equivalence of elastic pipelined system with early evaluation against synchronous parents system. The key idea

is to compute the reachable states of the elastic controller with early evaluation mechanism and use this information in computing refinement maps and rank functions. The reachable states are themselves computed using token-flow diagrams. The efficacy of the methods are demonstrated by verifying several elasticized pipelined processor models defined at the bit-level. For future work, we plan to develop a refinement-based verification tool for elasticized designs that will incorporate the developed methods. Also, using this tool, we will apply these methods to various other designs.

REFERENCES

[1] Julien Boucaron, Jean-Vivien Millo, and Robert de Simone, *Another glance at relay stations in latency-insensitive design*, Electr. Notes Theor. Comput. Sci. **146** (2006), no. 2, 41–59.

[2] Luca P. Carloni, Kenneth L. McMillan, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli, *A methodology for correct-by-construction latency insensitive design*, ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design (Piscataway, NJ, USA), IEEE Press, 1999, pp. 309–315.

[3] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli, *Theory of latency-insensitive design*, IEEE TCAD **20** (2001), no. 9, 1059–1076.

[4] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli, *Coping with latency in soc design*, IEEE Micro **22** (2002), no. 5, 24–35.

[5] Mario R. Casu and Luca Macchiarulo, *Adaptive latency-insensitive protocols*, IEEE Design and Test of Computers **24** (2007), 442–452.

[6] Jordi Cortadella and Michael Kishinevsky, *Synchronous elastic circuits with early evaluation and token counterflow*, DAC, 2007, pp. 416–419.

[7] Jordi Cortadella, Michael Kishinevsky, and Bill Grundmann, *Synthesis of synchronous elastic architectures*, DAC, 2006, pp. 657–662.

[8] ITRS, *International technology roadmap for semiconductors 2007 edition*, 2007, See URL `http://www.itrs.net/Links/2007ITRS/Home2007.htm`.

[9] Sava Krstic, Jordi Cortadella, Michael Kishinevsky, and John O'Leary, *Synchronous elastic networks*, FMCAD, 2006, pp. 19–30.

[10] Cheng-Hong Li, Rebecca Collins, Sampada Sonalkar, and Luca P. Carloni, *Design, implementation, and validation of a new class of interface circuits for latency-insensitive design*, MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, 2007, pp. 13–22.

[11] Panagiotis Manolios, *Correctness of pipelined machines*, FMCAD, 2000, pp. 161–178.

[12] Panagiotis manolios, *Mechanical verification of reactive systems*, Ph.D. thesis, University of Texas at Austin, August 2001, See URL `http://www.cc.gatech.edu/~manolios/publications.html`.

[13] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon, *Automatic memory reductions for rtl model verification*, ICCAD, 2006, pp. 786–793.

[14] Lawrence Ryan, *Siege homepage*, See URL `http://www.cs.sfu.ca/~loryan/personal`.

[15] Sudarshan K. Srinivasan, Koushik Sarker, and Raj S. Katti, *Verification of synchronous elastic processors*, IEEE Embedded Systems Letters **1** (2010), no. 1, 14–18.

[16] Syed Suhaib, Deepak Mathaikutty, David Berner, and Sandeep Shukla, *Validating families of latency insensitive protocols*, IEEE Transations on Computers **55** (2006), no. 11, 1391–1401.

[17] Christer Svensson, *Synchronous latency insensitive design*, ASYNC, IEEE Computer Society, 2004, p. 3.

[18] Muralidaran Vijayaraghavan and Arvind Arvind, *Bounded dataflow networks and latency-insensitive circuits*, MEMOCODE'09: Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign (Piscataway, NJ, USA), IEEE Press, 2009, pp. 171–180.

CURRICULUM VITAE

Yangwei Cai was born on Nov. 21, 1982. He got his bachelor's degree in Computer Science and Technology at Hebei University of Engineering in the summer of 2005 and master's degree in Detecting Technology and Automation Equipment at Nanjing Forestry University in 2008. He entered North Dakota State University to pursue a master's degree in Electrical Engineering (Computer Engineering) in the fall of 2008. Currently, he is doing PhD degree in Computer Engineering at University of South Florida from Fall 2010.