VERIFICATION OF SYNCHRONOUS ELASTIC PIPELINED SYSTEMS

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

By

Koushik Sarker

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

Verification of Synchronous Elastic Pipelined Systems

By

Koushik Sarker

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

SUPERVISORY COMMITTEE:

ABSTRACT

Sarker, Koushik, M.S., Department of Electrical and Computer Engineering, College of Engineering and Architecture, North Dakota State University, May 2010. Verification of Synchronous Elastic Pipelined Systems. Advisor: Dr. Sudarshan K. Srinivasan.

The constant shrinking of technology has lead to several design challenges that the synchronous design paradigm is unable to cope with. Elastic design is a novel and promising design paradigm that overcomes many of these challenges by using components that are insensitive to the latencies of its inputs.

Verification is a critical problem for any design paradigm. The complexity of elastic designs arises when the system is pipelined. We develop formal verification techniques to verify synchronous elastic pipelined systems. Note that the goal of verification 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.

We develop two formal verification procedures. The first procedure checks the correctness of elastic pipelined systems against their synchronous parent pipelined systems. The second procedure checks the correctness of elastic pipelined systems against their high-level non-pipelined specifications (such as an instruction set architecture). Dataflow through elastic architectures is complicated by the insertion of any number of elastic buffers in any place in the design. We introduce elastic token-flow diagrams, which are used to track the flow of data in elastic architectures. We provide a method to construct such diagrams. We also develop highly automated and systematic procedures based on elastic token-flow diagrams that compute functions that

map states of elastic systems to states of their specifications. Such functions, known as refinement maps, are used to compare behaviors of elastic and synchronous systems and hence prove their equivalence. We elasticized a 5-stage DLX processor that enables the insertion of buffers in its data path. We constructed several elastic processors by introducing up to 5 elastic buffers at various places in the data path and verified equivalence with both their synchronous parent pipelined systems and also with their instruction set architecture specifications.

ACKNOWLEDGEMENTS

I want to thank and pay my regards to my advisor Dr. Sudarshan K. Srinivasan for his guidance and for giving me the opportunity to work under him.

I would also like to thank Dr. Rajendra Katti for helping me in my research. I thank Dr. Cristinel Ababei and Dr. Vasant Ubhaya for agreeing to be on my defense committee. I also want to thank Yangwei Cai for helping me during my thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

The impact of persistent technology scaling results in a previously ignored set of design challenges such as manufacturing and process variability and increasing significance of wire delays. These challenges threaten to invalidate the effectiveness of synchronous design paradigms at the system-level. Several alternate design paradigms to deals with these challenges are being proposed. Synchronous Elastic Networks (SEN) [1, 2] has been proposed as an effective approach to design latency-insensitive systems. Another popular trend is latency-insensitive designs, which allows for variability in data propagation delays [3].

The design of latency insensitive circuits is becoming popular in current design techniques. These circuits can solve the time constraint problem in the digital circuit design. In current nano technologies, the layout of the circuit is generated first, and then we can compute the delay of the paths in the circuit. Situations are likely where the communication latencies need to be fixed after the layout of the chip has been generated. In this case, the designer has to make the necessary modifications in the circuit and afterwards generate the layout once again. Synchronous Elastic circuits are one kind of latency insensitive circuits and can resolve the communication delay constraints. The new latency insensitive design needs to be verified to make sure that it behaves in the same way as the conventional synchronous circuit does.

The goal of our research is to convert a conventional synchronous 5 stage pipeline microprocessor to an elastic model and then verify its functionality. There are

several protocols and algorithms which can change synchronous circuits to elastic circuits. We use the 'Synchronous Elastic Flow' (SELF) protocol [1]. For verification techniques, We use refinement map and rank function [4].

This chapter provides an overview of the problems that will be addressed in the rest of the thesis and gives a brief summary of the thesis contents.

1.1. Elastic Architecture

Cortadella et al. [1] presents SELF (Synchronous Elastic Flow) which can efficiently convert regular synchronous designs into elastic forms. The structure of the elastic system is a collection of elastic modules and elastic channels. All flip-flops are replaced by Elastic Buffers (EB). For every EB, there is a control block which results in changing or holding states of an EB. The Free State Machine (FSM) specification for the control block shows that states of the EB can be half, full or empty. The data transfer is indicated by the combination of valid and stop signals generated from the control block. They also demonstrated different circuit blocks for an elastic buffer, control circuits, multiple input output controllers that they refer as joins and forks. Register files and the instruction and data memories were considered as combinational units. They implemented the linear elastic pipeline with three versions of controllers. The authors claim that this technique is scalable and can be introduced at the level of functional units.

Krstic et al [2] gives a mathematical representation of the elastic systems. They introduced machines as a mathematical abstraction of circuits without combinational

cycles. The circuits are considered as stream functions. In order to define elastic behavior they demonstrated four parts of elastic machine - input output structure, persistence conditions, liveness conditions and the transfer determination conditions. Their work involves mathematical representations of the elastic behavior. But they did not describe any particular protocol like Cortadella et al. did. Their paper is good for understanding the behavior of elastic circuits, but cannot demonstrate any particular protocol to convert a conventional model to an elastic one.

The latency insensitive model proposed by Carloni et al. [3] involves the preliminary assumption that the system is completely synchronous. The circuit blocks are considered as a collection of modules having zero delay, i.e. a delay negligible with respect to the period of the common clock which is referred to as a virtual clock. After the final implementation is done, the operation of the circuit is controlled by a real clock which has a precise frequency. Their approach is to ignore time constraints during the early phases of the design when correct measures of the delay are not yet available. After the corresponding physical implementation is completed, they try to fix the problem, if there is any, by inserting the right amount of relay stations. There will be no modification in the functionality or layout of the individual modules since every module works according to the latency-insensitive protocol.

All these techniques are useful to design latency insensitive circuits. In our research work, we prefer to implement the SELF protocol proposed by Cortadella et al. Our goal is to transform a conventional synchronous 5 stage machine into its elastic

form by using SELF protocol. The second part of my work is to verify the new design. In the next section, I discuss about verification methodology.

1.2. Microprocessor Verification Techniques

Well-founded Equivalence Bisimulation (WEB) technique [4, 5] is used to verify the functional correctness of pipeline machines. The WEB technique involves refinement map which can be efficiently used for processor model verifications [6], which will be defined in greater detail in later chapters. Some important features of WEB refinement technique is that, it accounts for stuttering [6], preserves safety and liveness properties [7, 8] and demonstrates compositional reasoning for pipelined processor verification [9, 10, 11].

The Bit-level Analysis (BAT) tool [12, 13] was used for bit-level verification for the pipeline machines. The equivalence checking of elastic pipelined systems against their synchronous parents were done using BAT. BAT considers all the inputs and outputs as bit vectors. Memories are also considered as bit vectors in a special way. BAT takes in a machine description and LTL specification, and tries to find a counterexample. The machine specification has 4 required sections, :vars, :init, :trans, and :spec. The equivalence checking of the elastic pipelined systems against their instruction set architecture specification was performed using ACL2-SMT. The processor models were defied at bit level using BAT and term level using ACL2-SMT.

1.3. Thesis Contributions

A major part of the thesis involves the conversion a 5 stage DLX pipeline machine to its synchronous elastic model. We use Synchronous Elastic Flow (SELF) protocol to convert a conventional synchronous pipeline microprocessor to an elastic one. The elastic pipeline machine is tolerant to the inputs of variable latency in terms of clock cycle. Each of the individual stages is constructed using Elastic Buffers. The memories and register files are treated as combinational blocks. Each Elastic Buffer is controlled by an associated control block. The control block deals with two 1 bit signals – valid and stop. The valid signal indicates that the receiver is ready to accept data from the sender. The stop signal tells the sender not to transmit data if the receiver is not ready. We used 5 additional Elastic Buffers in different paths of the pipeline machine; these additional buffers introduce variable latency in the forwarding data paths.

We develop a formal verification procedure to check the correctness of synchronous elastic pipelined processors against their synchronous parent systems. 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. Dataflow through elastic architectures is complicated by the insertion of any number of elastic buffers in any place in the design. We introduce elastic token-flow diagrams, which are used to track the flow of data in elastic architectures. We provide a method to construct such diagrams. We also develop a highly automated and systematic procedure based on elastic token-flow diagrams that

computes 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. We elasticized a 5 stage DLX processor that enables the insertion of buffers in its data path. We constructed several elastic processors by introducing up to 5 elastic buffers at various places in the data path and verified equivalence with their synchronous parent processor.

We also develop a formal verification procedure to check that elastic pipelined processor designs correctly implement their instruction set architecture (ISA) specifications. The notion of correctness we use is based on refinement. Refinement proofs are based on refinement maps, which- in the context of this problem- are functions that map elastic processor states to states of the ISA specification model. Data flow in elastic architectures is complicated by the insertion of any number of buffers in any place in the design, making it hard to construct refinement maps for elastic systems in a systematic manner. We introduce token-aware completions functions, which incorporate a mechanism to track the flow of data in elastic pipelines, as a highly automated and systematic approach to construct refinement maps. We demonstrate the efficiency of the overall verification procedure based on token-aware completion functions using six elastic pipelined processor models based on the DLX architecture.

CHAPTER 2. ELASTIC CIRCUITS

2.1. Properties and Applications

The current trend in circuit design is constant technology scaling which is reducing the circuit area by a great deal. As a result, wire delays are becoming important and comparable to gate delays. This causes severe problems in the synchronous design paradigm. In current design techniques, the design layout is generated first. The time delays of the different circuit paths are computed after that. As a result, number of clock cycles required to transmit data over a particular block of circuits cannot be estimated because of the effect of the long wire delays.

After the final layout has been generated, it is possible to compute total gate and wire delays. But circumstances may arise where wire delays can contribute the major part of the circuit delay. In this situation, the designer has to go back to the initial design to make the necessary changes to reduce the effect of wire delay. The layout of the new design is generated to adjust the total delay in terms of number of clock cycles. This delay optimization process creates complexity as the initial design and the layout is changed several numbers of times.

Synchronous elastic circuits can overcome this problem. Elastic circuits are latency insensitive circuits. If latency is introduced in any part of the elastic circuit, they behave similarly like their synchronous parent circuits do. The comparative dataflow for

the synchronous and elastic circuits is shown in Table 2.1. This characteristic of elastic circuits allows different inputs to vary in time to reach a particular execution unit.

Clock Cycle	1	2	3	4	5
Synchronous System	t	t	t	t	t
Elastic System	t	*	*	t	t

Table 2.1. An example of token flow in synchronous and elastic system.

In conventional synchronous circuits, valid data is present in every clock cycle. On the contrary, invalid data cycles may appear in case of elastic circuits. We refer to the valid data as tokens (t) and invalid data as bubbles (*). The insertion of bubbles does not alter the functionality of the elastic system.

Synchronous elastic circuits overcome the wire delay problems. The core idea of the synchronous elastic circuit is to break the long wire into small chunks and insert additional buffers in between them. 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.

2.2. Synchronous Elastic Flow (SELF)

Cortadella et al. presents Synchronous Elastic Flow (SELF) [1] protocol which can efficiently convert regular synchronous designs into elastic forms. The new model consists of elastic buffers, control blocks and elastic channels which are implemented using the handshaking of valid and stop bits. SELF protocol controls the data transfer between two modules using valid and stop bits. Data is transferred over a channel when the sender is providing valid data and the receiver is ready to accept it. No data transfer occurs when there is not a valid bit. The sender sends the same data once again when the sender provides valid data but when the receiver is not ready to accept it. SELF protocol also demonstrates special control blocks known as 'join' and 'fork' which are used when the circuit has multiple input and output channels. We have used join to combine two control signals in multiple input channels; the output of the join is valid only when both the inputs are valid. We have used 'eager fork' in multiple output channels. An output valid signal is generated for a receiver whenever it is ready to accept data. The register file, instruction memory and data memory are considered as combinational units. The control layer connections have been made according to the original connections that we have for the data path.

2.3. Elastic Buffers

Elastic Buffers (EB) are characterized as unbounded First In First Out (FIFO) latches. An Elastic Buffer, shown in Figure 2.1, is constructed using two transparent

latches, termed as Elastic Half Buffers (EHB), which operate on the different edges of a clock cycle. Each EB is associated with a control block which generates the enable signals for the EHBs. The enable signals are AND-ed with the clock phases to synchronize the circuit operation with the clock input. The orientation is made in the way that the low phase of the clock is AND-ed with the master enable signal and the high phase is AND-ed with the slave enable signal. The implementation of an EB involves constructing the data-path and the control block. The descriptions of these two major blocks are given below:

2.3.1. Data Path of Elastic Circuit

Token transfer in an elastic circuit is caused by two different controlling bits termed as valid and stop bits. The valid bit propagates in a forward direction along with the token transfer. On the other hand, the stop bit propagates in the backward direction and acknowledges the status of the receiving end whether the receiver can accept the token. The propagation delay of these valid and stop bits determines the capacity of the EB.

The forward delay (d_f) is caused by the latency of the valid bit to propagate forward. The backward delay (d_b) is caused by the latency of the stop bit to propagate in the backward direction. According to the unbounded specification given by Cortadella et al, the values of d_f and d_b should be greater than zero [1]. If d_f and d_b become zero, the EB will no longer be active in the circuit, it will act like just a communication wire.

Carloni et al [4] demonstrated the constraint $d_f = d_b = 1$ to be satisfied in order to optimize the performance and distribute the EBs in the long communicational channels. The capacity of the EB is an important issue for the elastic circuit design. The number of tokens that can be stored in an EB is equal to its capacity. The following property defines the capacity of an EB:

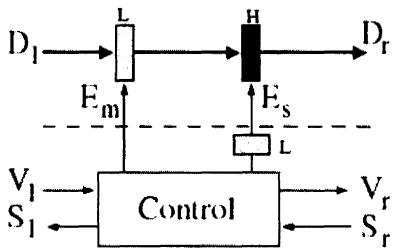

Figure 2.1. Block diagram of a latch based EB [1].

EB capacity property *The capacity of an EB, C, must satisfy the following constraint:* $C \geq (d_f + d_b)$. *[1]*

For simplicity and convenient use of EBs, both the values of d_f and d_b are assumed to be one, therefore, making the minimal capacity of the EB to two. In the experiments of this research work, all EBs have been constructed using two storage cells, mainly two back to back transparent latches with different polarity.

The design methodology of the elastic circuit data is to convert all flip-flops to two back to back latches, one receiving the data as input and the other releasing the stored data as output. We call these latches the master and slave latch, respectively. The master and slave latch have their individual enable signals which are generated from the

control block of an EB. The control block and its input-output signals are described in the following sub-section.

2.3.2. Control of Elastic Circuit

The control block of an EB governs the token flow in the elastic circuit. The control block is constructed using simple combinational gates and 1-bit control latches. The values of the control latches are used to prove the invariant property of the elastic machines, the details of the invariant properties are discussed in the later chapters. Figure 2.2 shows two different implementations of Elastic Controllers; in our research work, we used the circuit shown in Figure 2.2 (b).

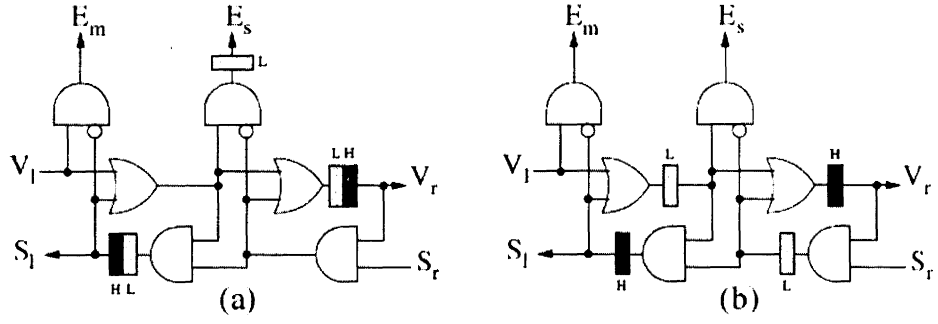

Figure 2.2. Two different implementations of elastic controllers [1].

The controller has one valid input (V_l) and one stop input (S_l). It generates a forward propagating output valid signal (V_r) and backward propagating output stop signal (S_r). The enable signals for the master and slave EHBs are indicated by E_m and E_s, which are AND-ed with the low and high phase of the clock respectively.

The controllers are synchronized with the clock and are connected in accordance with connections between EBs in the data path. Each controller has the following three possible states: [1, 14, 15]

1. Empty: The EB has no valid data token in storage. Therefore, the master and slave EHB is empty.

2. Half: The corresponding EB has one valid data token. The slave EHB keeps the valid data and the master EHB remains empty.

3. Full: The corresponding EB has two valid data tokens. Both the master and slave EHB have valid data. In this case, the controller generates an output stop signal to the immediate sender.

2.3.3. Advanced Join and Fork

Advanced structures such as 'join' and 'fork' [1, 14] are used when there are multiple input output channels. Figure 2.3 shows the implementation of a 'join' block. 'Join' needs to be used if there are multiple inputs coming to a particular unit. The output of the 'join' will be valid only if both the input valid signals are true. Multiple 'joins' can be used in a nested structure if there are more than 2 inputs for a particular block.

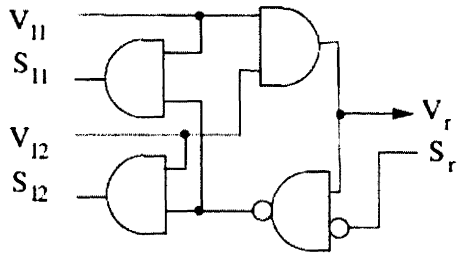

Figure 2.3. Join structure for multiple input channels [1]

The implementation of a fork is shown in Figure 2.4. Two latches of the fork are helpful to trace the output valid signals. The value of a particular fork latch is set to 1 if the corresponding output valid output is true. Unlike the join, the fork can transfer data to a particular receiver whenever the receiver is ready to accept the token. The back to back connection of a join and a fork does not create a combinational cycle and therefore this type of connection was used in this research work.

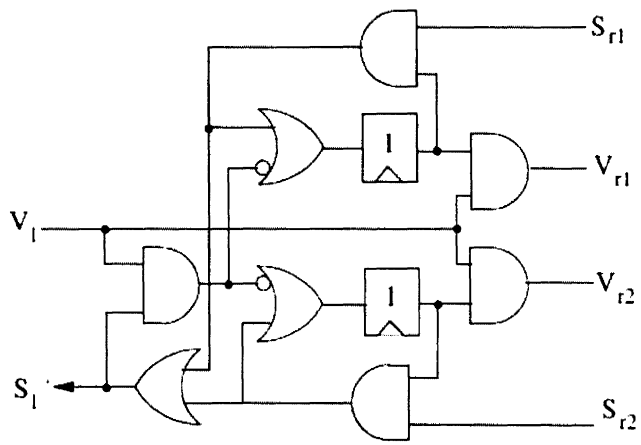

Figure 2.4. Fork structure for multiple output channels [1].

The core idea of Synchronous Elastic Flow (SELF) protocol in elastic circuits is to replace all latches with Elastic Buffers (EB), include control blocks for each of the EBs and use special control blocks for multiple inputs and outputs channels. Dataflow through elastic architectures is complicated by the insertion of any number of elastic buffers in place in the design. Using the SELF protocol, we elasticized a 5-stage DLX processor that enables the insertion of additional buffers in the datapath.

CHAPTER 3. PRELIMINARIES ON REFINEMENT

3.1. Core Theorem of WEB Refinement

The refinement problem can be described as follows. Given a high-level specification, say *S*, and a low-level implementation, say *I*, show that *I* correctly implements *S*. Refinement proofs are relative to a refinement map *r*, a function that maps implementation states to specification states. Our notion of refinement is based on stuttering bisimulation: for every pair of states *w*, *s* such that w is an implementation state and $s = r(w)$, we have that for every infinite path σ starting at *s*, there is a "matching" infinite path δ starting at *w*, and conversely. That σ and δ "match" implies that applying *r* to the states in δ results in a sequence that is equivalent to σ up to finite stuttering (repetition of states). Stuttering is a common phenomenon when comparing systems at different levels of abstraction and occurs when the implementation takes multiple steps to match a single step of the specification.

A detailed description of the theory of refinement can be found in [5]. It is enough to prove the correctness formula [4] given in Definition 1 (shown below) in order to establish refinement. In the formula below, IMPL denotes the set of implementation states, *Istep* is a step of the implementation machine, and *Sstep* is a step of the specification machine. *rank*, used for deadlock detection, is a witness function from implementation states to natural numbers whose value decreases when there is stutter.

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

$(\forall impl \in \text{IMPL} ::$

$spec = r\ (impl) \ \wedge \ next\text{-}spec = Sstep\ (spec) \ \wedge$

$next\text{-}impl = Istep(impl) \ \wedge \ next\text{-}spec \neq r\ (next\text{-}impl)$

$\rightarrow \quad spec = r\ (next\text{-}impl) \ \wedge \ rank\ (next\text{-}impl) < rank\ (impl)\ )$

In the above formula, *impl* is an implementation machine state. *spec* is the specification machine state obtained by applying the refinement map *r* to *impl*. *next-spec* is the successor of *spec* obtained by stepping the specification machine in state *spec* and *next-impl* is the successor of *impl* obtained by stepping the implementation state machine in state *impl*. The formula above states that for every implementation machine state *impl* its corresponding specification 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 with respect to the rank of *impl*. The proof obligation that *spec* = *r* (*next-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-impl*) < *rank*(*impl*) is the liveness component and guarantees that the machine will not deadlock, *i.e.*, will always make forward progress.

The correctness formula given above is expressible in a decidable fragment of first-order logic. Therefore, the correctness of pipelined machines as given by this formula can be automatically checked using a decision procedure for that logic.

3.2. Equivalence Checking

The goal of our verification procedure is to show equivalence between an elastic processor and its synchronous parent. The notion of equivalence that we use is Well Founded Equivalence Bisimulation (WEB) refinement [5], which is based on comparing behaviors of implementation (elastic) and specification (synchronous) systems. 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. A detailed description of the theory of refinement can be found in [5]. It is enough to prove the following correctness formula [4] to prove refinement (thereby establish equivalence) between an implementation and its specification.

Definition 2. *(Core WEB Refinement Correctness Formula)*

$$
\begin{aligned}
&(\forall w_1 \in \textit{MA-elas} :: \quad s_1 = r_1(w) \quad \wedge \quad u_1 = \textit{MA-sync-step}(s_1) \quad \wedge \\
&\qquad\qquad v_1 = \textit{MA-elas-step}(w_1) \ \wedge \ u_1 \neq r_1(v_1) \\
&\quad \rightarrow \qquad\qquad s_1 = r_1(v_1) \ \wedge \ \textit{rank}(v_1) < \textit{rank}(w_1))
\end{aligned}
$$

In the formula above, *MA-elas* denotes the set of elastic pipeline processor states, *MA-elas-step* is a step of the implementation machine, and *MA-sync-step* is a step of the specification machine. The refinement map r_1 is a function that maps the elastic pipeline processor states to the synchronous pipeline machine states. *rank*, used for deadlock detection, is a witness function from elastic pipeline machine states to natural numbers whose value decreases when there is stuttering.

WEB based refinement proofs were further used to check the equivalence between an elastic pipeline processor and an Instruction Set Architecture (ISA) model. We present a novel highly automated formal verification solution for latency-insensitive pipelined microprocessors developed using the SEN approach. The idea is to show that the elastic processor correctly implements all behaviors of its instruction set architecture (ISA) model, which is used as the high level specification for the processor. The notion of correctness that we use is Well Founded Equivalence Bisimulation (WEB) refinement, a detailed description of which can be found in [5]. It is sufficient to prove that the elastic processor (implementation) and its ISA (specification) satisfy the following core WEB refinement correctness formula to establish that the elastic processor refines i.e. correctly implements its ISA:

Definition 3. *(Core WEB Refinement Correctness Formula)*

$$(\forall w_2 \in \textit{MA-elas} :: \quad s_2 = r_2(w_2) \ \wedge \ u_2 = \textit{ISA-step}(s_2) \ \wedge \ v_2 = \textit{MA-elas-step}(w_2) \ \wedge \ u_2 \neq r_2(v_2) \rightarrow s_2 = r_2(v_2) \ \wedge \ \textit{rank}(v_2) < \textit{rank}(w_2))$$

In the formula above, IMPL denotes the set of implementation states, *MA-elas* is a step of the elastic pipeline machine, and *ISA-step* is a step of the Instruction Set Architecture (ISA) machine. The refinement map r_2 is a function that maps elastic pipeline machine states to ISA states. In fact, the refinement map can be thought of as an instrument to view the behaviors of the elastic pipeline machine at its specification level, in this case, the ISA. Therefore this technique allows verification tools to easily compare the behaviors of the two systems. *rank* is used for deadlock detection. Our focus in this work is to check safety, *i.e.*, to show that if the implementation makes progress, then, the result of that progress is correct as specified by the high-level specification.

3.3. Verification Methodology

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). This is a crucial step as applying the verification method to all syntactically possible states (including unreachable states) often leads the verification tool to flag spurious bugs (or spurious counter examples) and thereby hindering verification from proceeding. We use invariant properties to compute 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) and Applicative Common Lisps (ACL2) system for modeling and verification. For equivalence checking between elastic processor and synchronous processor, we used BAT for modeling and verification purpose. For equivalence checking between elastic pipeline processor and ISA, Modeling and verification is also performed using ACL2-SMT, a system developed by combining the ACL2 theorem prover (version 3.3) with the Yices decision procedures (version 1.0.10).

CHAPTER 4. ELASTIC PROCESSOR MODELS

4.1. Synchronous Parent Processor

The elastic processor models are based on the 5-stage synchronous DLX pipeline processor. The bit-level models are described using Bit-Level Analysis Tool (BAT). The models were also defined at the term-level using the Applicative Common Lisp 2 (ACL2) programming language. The models were obtained by elasticizing a synchronous 5-stage DLX processor using Synchronous Elastic Flow (SELF) protocol approach [1]. The synchronous 5 stage DLX pipeline processor shown in Figure 4.1 has 5 basic pipeline latches (*pc, fd, de, em, mm*) which store data after completion of the functions of each corresponding stage.

The program counter (pc) points to the current instruction to be fetched from the Instruction Memory (imem). The processor has one level of instruction fetch cycle and instruction decode cycle. The Arithmetic Logic Unit (ALU) is capable of processing some binary arithmetic and logical operations. The results processed by ALU can be stored in the data memory (dmem) or register files (RF).

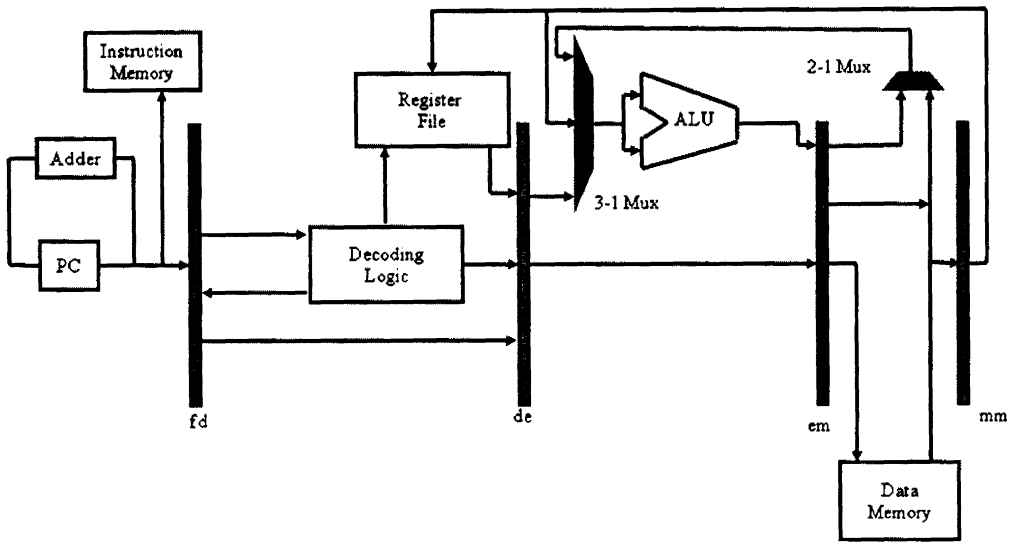

Figure 4.1: High level organization of conventional 5-stage DLX processor.

4.2. Conversion to Elastic Processor Models

The following steps are taken in order to convert the conventional 5 stage DLX processor into an elastic processor:

(1) The five basic latches of the conventional synchronous DLX processor (*pc, fd, de, em and mm*) are converted to Elastic Buffers (EB). EB is split into master and slave Elastic Half Buffers (EHB).

(2) For every EB, a control block is implemented. The control block operates the associate EB by generating enable signals for the EHBs. The control block also communicates with other EBs by interchanging the valid & stop signal with them.

(3) For multiple input communication channels, the 'join' block is inserted at the input of an EB.

(4) 'Fork' block is inserted at the node of multiple output channels.

(5) The control block connectivity is done in accordance with connections between pipeline stages in the data path.

(6) Register file, data memory and instruction memory were considered combinational units and no change was made to them.

The simple model of the elastic 5 stage DLX processor is shown in Figure 4.2.

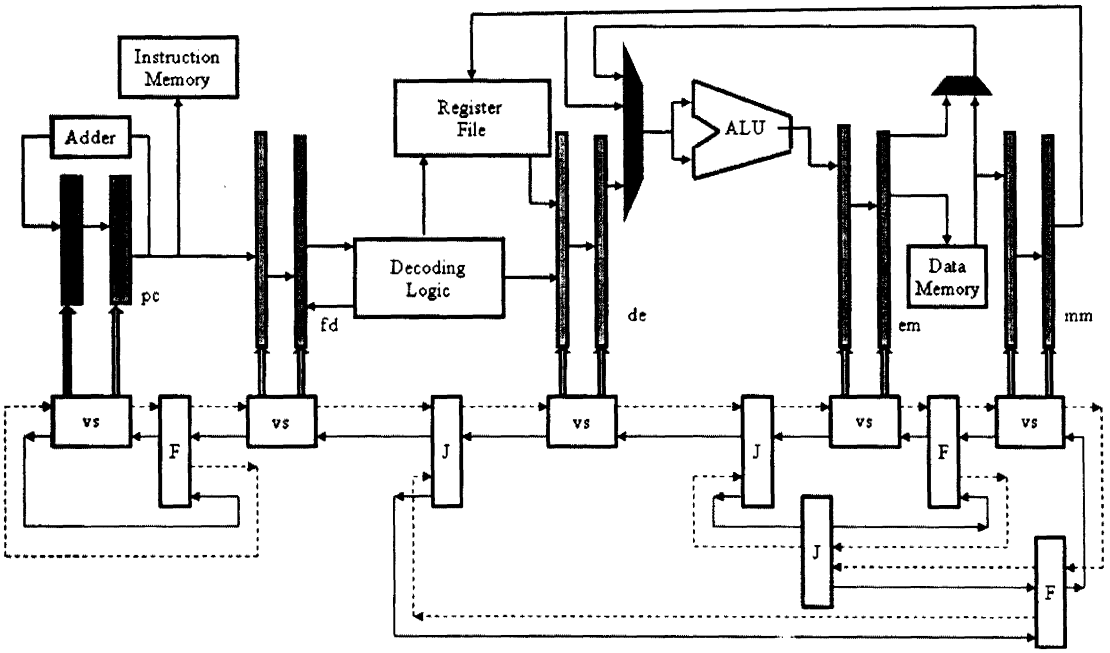

Figure 4.2: High level organization of elastic 5-stage DLX processor.

4.3. Elastic Processor Models

The elastic 5-stage DLX processor model with five additional elastic buffers (*l1, l2, l3, l4 & l5*) is shown in Figure 4.3. The model is defined at the bit-level using the Bit-level Analysis Tool specification language [6] and at term level using ACL2. The data path width is 32 bits. The model was obtained by first elasticizing a synchronous 5-stage DLX processor using the Synchronous Elastic Flow (SELF) protocol approach [1].

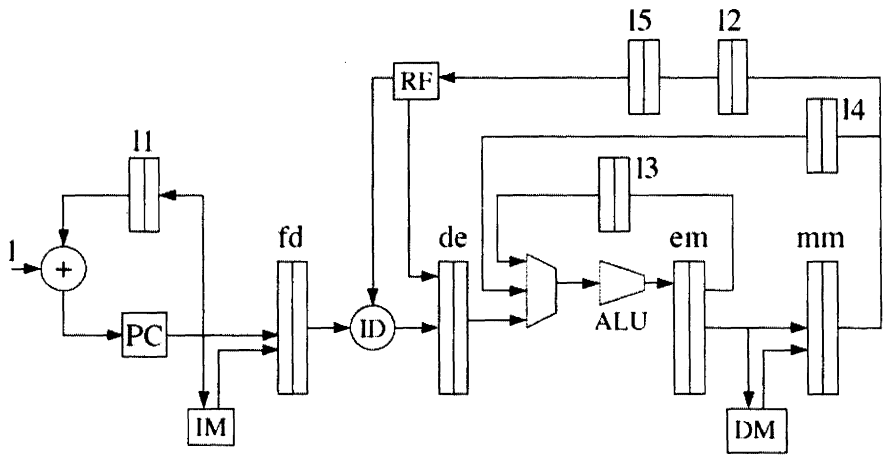

Figure 4.3: High level organization of elastic 5 stage DLX processor M5.

The main idea is to replace all flip flops with elastic buffers. The clock network is replaced by a network of elastic controllers, where each controller is used to control the elastic buffers in a pipeline stage and synchronize with the controllers of adjacent pipeline stages. The network of elastic controllers for the DLX processor with five additional elastic buffers in the data path is shown in Figure 4.4.

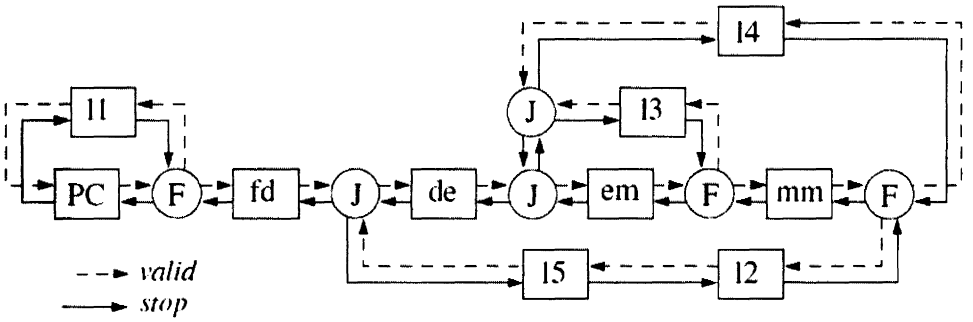

Figure 4.4: Network of elastic controllers for the elastic 5 stage DLX processor.

The controllers are synchronized with the clock, and are connected in accordance with connections between pipeline stages in the data path. Each controller has three possible states, *empty*, *half*, and *full*, which indicate that the corresponding elastic buffer has 0, 1, and 2 valid data tokens, respectively. We call the processor model obtained by elasticizing the synchronous DLX M0. The main advantage of the elastic processor is that it permits the insertion of additional elastic buffers at any place in the data path to break long wires. We therefore inserted additional elastic buffers *l1, l2, l3, l4* and *l5* at various places in the model. We inserted *l1* in model M0 to get model M1. We then inserted *l2* in model M1 to get M2. We derived models M3, M4, and M5 in a similar manner. The model M5 is shown in Figure 4.3. The figure also shows the positions of the additional elastic buffers and how they are connected with the elastic buffers corresponding to the pipeline latches (namely *pc*, *fd*, *de*, *em*, and *mm*). These models are used to demonstrate the effectiveness of our verification approach.

4.4. BAT Model for the Elastic Processors

Bit-level Analysis Tool (BAT) is used to define the elastic processors at the bit level [16]. All the latches were represented by bit vectors and are included in the current state of the processor, *mastate*, which combines all the bits stored in the data path and control latches. The current state of the synchronous and elastic DLX processor is represented by *mastate_syn* and *mastate_elas* respectively.

Synchronous DLX pipeline *mastate* is the bit vector that holds all the current tokens in 5 basic latches (*pc, fd, de, em & mm*) and 2 forwarding path latches (*dearg1 & dearg2*). When the synchronous DLX is stepped forward, new tokens are updated in each of these latches creating a new bit vector value of the *mastate*.

The elastic DLX pipeline *mastate* structure is more complex as each of the latches is split into 2 parts which are referred as Elastic Half Buffers (EHB). It also includes the control block and fork circuit latches. Each of the control blocks has four 1-bit latches. Each of the fork circuits has two 1-bit latches. All these combine a larger *mastate* bit vector for the elastic pipeline machine.

The implementation of *mastate* gets more complex if additional EB's are placed in different parts of the data path. The complexity of *mastate* for M1, M2, M3, M4 and M5 arises because of two reasons. Firstly, insertion of *l1, l2, l3, l4* and *l5* one after another gradually increases the size of the *mastate*. Secondly, some additional history variables have to be included in the *mastate*, as we often need to roll back the processor to track the previous states of an EB. As an elastic processor can take several transitions

to match a single synchronous machine state, it is often necessary to pull up the history values to compare elastic states with the synchronous ones.

In Bit-level Analysis Tool (BAT), a particular EB is implemented using two functions named as *step_half* and *step_full*. The *step_half* function makes some changes in the token state of an EB when the clock edge transition takes place. The following function pc_step_half is a similar type of function. The inputs to the pc_step_half are the current token states, control input bits and the clock edge; the function produces an gives an output of 68 bits which includes all the next state values for the pc.

```
(pc_step_half (68)   ((pc_in 32) (pc_w 68) (v1 1) (s2 1) (rise 1))
(local
 ((fall (not rise))
 (pc_low  (getppc-low    pc_w))
(pc_high (getppc-high   pc_w))
(pc_vs   (getppc-vs     pc_w))
(l1     (ex_l1     pc_vs))
(h1     (ex_h1     pc_vs))
(l2     (ex_l2     pc_vs))
(h2     (ex_h2     pc_vs))
(l1_next (cond (fall (or v1 h1)) (0b1 l1)))
(h1_next (cond (rise (and l1 l2)) (0b1 h1)))
```

(h2_next (cond (rise (or l1 l2)) (0b1 h2)))

(l2_next (cond (fall (and s2 h2)) (0b1 l2)))

(em (and v1 (not h1) fall))

(es (and l1 (not l2) rise))

(pc_low_next (cond (em pc_in) (0b1 pc_low)))

(pc_high_next (cond (es pc_low) (0b1 pc_high)))

(pc_w_next

(cat pc_low_next pc_high_next l1_next h1_next l2_next h2_next)))

pc_w_next))

The pc_step_half is recalled twice using the function pc_step_full, which ensures the transition of an EB to its next token state. The function also updates the controller latches to their next state values.

(pc_step_full (68)

((mastate 740) (v1 1) (s2 1))

(local

((ppc (get-ppc mastate))

(pc_high (getppc-high ppc))

(pc_in (mod+ pc_high 1))

(pc_next (pc_step_half pc_in ppc v1 s2 0b0))

(pc_final (pc_step_half pc_in pc_next 0b0 0b0 1b1)))

pc_final))

All the pipeline stage EBs (*pc, fd, de, em, mm*) and additional EBs in the data path (*l1, l2, l3, l4, l5*) are implemented in similar way. The controller connectivity is made according to the original data path connections.

CHAPTER 5. TOKEN FLOW DIAGRAM

5.1. Elastic Token Flow Diagrams

In this section, we introduce elastic token-flow diagrams, which are obtained by simulating the flow of data tokens in the elastic controller network. The data tokens are assigned numeric labels and the rules of simulation are modified so as to distinguish data tokens that correspond to new data units that enter the system, from data 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 in a highly automated and systematic manner.

A token t is a natural number. A token with value 0 corresponds to a bubble. The token-flow diagram is based on the token-state of an elastic controller network, which is defined as follows.

Definition 4. *The token-state of an elastic controller network with n controllers is an n-tuple of pairs* (t_m, t_s) *where* $t_m, t_s \in \mathrm{N}$.

A token-flow diagram is a table where each row corresponds to a token-state and each successive row is obtained by simulating the numbered tokens in the elastic design that is currently being analyzed. The following nine rules are used to simulate these numbered tokens in an elastic network. We use the following notation. If e is an elastic

controller, then $t_m.e$ and $t_s.e$ are the tokens in the current state of the controller corresponding to the master elastic half buffer (EHB) and slave EHB. $t'_m .e$ and $t'_s .e$ are the tokens in the next state of the controller corresponding to the master EHB and slave EHB. $v_i.e$ and $s_i.e$ are the current values of the input valid and input stop signals to e. The values for the valid and stop signals can be obtained from the circuit of the elastic controller network. t_a and t_b are valid tokens. A token with a zero value corresponds to a bubble. When the program counter is updated, it corresponds to a new instruction/data unit/token introduced into the processor. If t_i is the input token to controller, the *new-token* function is used to distinguish how the token is updated when it enters the controller of the program counter as opposed to other controllers.

$$\textit{new-token}\ (e, t_i) = \begin{cases} e = pc & t_i + 1, \\ \textit{otherwise} & t_i. \end{cases}$$

The J operator corresponds to the join of data flowing from two EBs to one. The rules are given below:

1) $(t_m.e = 0 \wedge t_s.e = t_a \wedge \neg\ v_i.e \wedge \neg\ s_i.e) \rightarrow (t'_m .e = 0 \wedge t'_m.e = 0)$

2) $(t_m.e = 0 \wedge t_s.e = t_a \wedge v_i.e \wedge \neg\ s_i.e) \rightarrow (t'_m .e = 0 \wedge t'_s .e = \textit{new-token}\ (e, t_i))$

3) $(t_m.e = 0 \wedge \dot{t}_s.e = ta \wedge \neg\ v_i.e \wedge s_i.e) \rightarrow (t'_m.e = 0 \wedge t'_s.e = t_a)$

4) $(t_m.e = 0 \wedge t_s.e = t_a \wedge v_i.e \wedge s_i.e) \rightarrow (t'_m.e = \textit{new-token}\ (e, t_i) \wedge t'_s.e = t_a)$

5) $(t_m.e = t_a \wedge t_s.e = t_b \wedge \neg\ s_i.e) \rightarrow (t'_m.e = 0 \wedge t'_s.e = t_a)$

6) $(t_m.e = t_a \wedge t_s.e = t_b \wedge s_i.e) \rightarrow (t'_m.e = t_a \wedge t'_s.e = t_b)$

7) $(t_m.e = 0 \wedge t_s.e = 0 \wedge \neg\ v_i.e) \rightarrow (t'_m.e = 0 \wedge t'_s.e = 0)$

8) $(t_m.e = 0 \wedge t_s.e = 0 \wedge v_i.e) \rightarrow (t'_m\ .e = 0 \wedge t'_s.e = \textit{new-token}\ (e, t_i))$

9) $J(t_a, t_b) = \begin{cases} t_a > t_b & t_a, \\ otherwise & t_b \end{cases}$

Rules 1 through 4 correspond to the half state of a controller, rules 5 and 6 correspond to the full state of a controller, and rules 7 and 8 correspond to the empty state of a controller. Rule 9 describes how two input tokens are merged. In in-order pipelines, in a join, tokens with smaller values correspond to feedback paths, and the token with the largest value corresponds to an instruction. Rule 9 is used to destroy tokens corresponding to instructions that have completed and also helps avoid the duplication of tokens.

5.2. Reachability

The elastic controller network is a deterministic system. Therefore, reachability analysis can be performed by simulating the elastic network starting from an initial state until a convergence is reached, *i.e.*, a state of the controller is reached that has already been visited as depicted in Figure 5.1. In the figure S_0 is the initial state. When the controller network transitions from S_r, it goes back to S_k, a state that has already been visited. Therefore the reachable states of the controller network are S_0 through S_r.

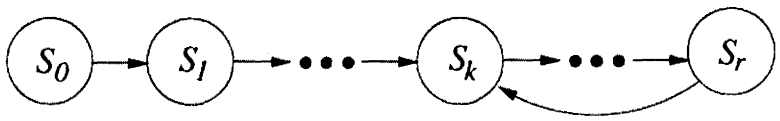

Figure 5.1. Reachability analysis of elastic controller network.

While regular simulation can be used for reachability analysis, we use the elastic token-flow diagram instead as information from this diagram about the reachable states is later used in computation of the refinement map and the rank function. In the reset state of an inorder 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 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 enable data flow in the elastic system. Also note that once states S_0, ..., S_m 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_m in which the controller of the program counter has at least one valid token.

5.3. Reachability Analysis

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 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*, *l3*, *l4*, *l5*}. Then, the initial token state S'_0 is given using the following assignments to EC_p and EC_a.

$$\{\forall e \in EC_p :: (t_m.e = 0) \wedge (t_s.e = n + 1 - pos\ (e, EC_p))\}$$

$$\{\forall e \in EC_a :: (t_m.e = 0) \wedge (t_s.e = 0)\}$$

2) The set of visited states V is initialized to {*elastic-state* (S'_0)}. Initialize the current token-state S'_c to S'_0 and the loop counter i to 0.

3) Compute the next token-state S'_{i+1} from S'_i using the elastic token-flow procedure given in Section 5.1.

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

5) Increment i and goto step 3.

Table 5.1 shows the elastic token-flow diagram generated for reachability analysis of the M5 DLX-based elastic processor. Each entry in the table gives the tokens present in an elastic controller. The reachability analysis demonstrates the following things:

State	pc	fd	de	em	mm	l1	l2	l3	l4	l5
S_0	5	4	3	2	1	0	0	0	0	0
S_1	0	5,4	3	0	2	5	2	1	0	1
S_2	6	5,4	0	3	0	0	0	2	1	2
S_3	6	5	4	0	3	6	3	2	2	0
S_4	7	6	5	4	0	0	0	3	0	3
S_5	0	7,6	5	0	4	7	4	3	3	0
S_6	8	7	6	5	0	0	0	4	0	4

Table 5.1. Token flow diagram: Reachability

- An entry 0 indicates that no tokens are present, *i.e.*, $t_m.e = 0$ and $t_s.e = 0$.
- An entry with a single token t indicates that $t_m.e = 0$ and $t_s.e = t$. An entry with two tokens of the form t_1, t_2 indicates that $t_m.e = t_1$ and $t_s.e = t_2$.

For the M5 processor the reachable states of the elastic controller network are states *elastic-state* (S_4) and *elastic-state* (S_5). Note that *elastic-state* (S_4) = *elastic-state* (S_6).

CHAPTER 6. REFINEMENT

6.1. 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. The example of constructing such refinement maps have been shown in Table 6.1 and Table 6.2 for state 1 and state 2 of M5 processor respectively.

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 would be an n tuple with one token for each latch. For the 5-stage synchronous DLX, a token state would be of the form (pc, fd, de, em, mm) and three possible successive token states for the synchronous DLX would be (5, 4, 3, 2, 1), (6, 5, 4, 3, 2), and (7, 6, 5, 4, 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.

State	PC	FD	DE	EM	MM
S_{c-1}	0	8,7	6	0	⑤
S_c	⑨	⑧	⑦	⑥	0
Sync	9	8	7	6	5

Table 6.1. Refinement map construction for state 1 of processor M5.

State	PC	FD	DE	EM	MM
S_{c-2}	0	8,7	6	0	⑤
S_{c-1}	⑨	8	7	⑥	0
S_c	0	9,⑧	⑦	0	6
Sync	9	8	7	6	5

Table 6.2. Refinement map construction for state 2 of processor M5.

One approach to define the refinement map is to roll back some or all the pipeline latches 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:

$(\forall e \in EC_p :: t_m.e = 0 \wedge t_s.e = t_{pc} + 1 - pos(e, ECp))$.

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 [4]. 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*.

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.

6.2. Token-Aware Completion Functions

Flushing [17] is one standard approach used to compute refinement maps for pipelined processors. In this approach, partially executed instructions in the pipeline latches are forced to complete, without allowing the machine to fetch any new instructions. Projecting out the programmer visible components - which include the program counter, register file, instruction memory, and data memory for the models we consider - in the resulting state will give the corresponding ISA state. Completion functions [18, 19, 20] were proposed as a computationally efficient approach to construct flushing refinement maps. One completion function for each pipeline latch in the machine is used to compute the effect on the programmer visible components of completing any partially executed instruction in that latch. The completion functions are composed to form the flushing refinement map. Note that older instructions in the pipeline are completed before younger instructions.

For the DLX example, let *fdc*, *dec*, *emc*, and *mmc* be the completion functions for the latches *fd*, *de*, *em*, and *mm*, respectively. The ISA state s corresponding to a synchronous DLX processor state w ($pc^w, fd^w, de^w, em^w, mm^w, rf^w, im^w, dm^w$) *is* ($pc^s, rf^s, im^s, dm^s$) = $(fdc(dec(emc(mmc(pc^w, rf^w, im^s, dm^w), mm^w), em^w), de^w), fd^w)$.

When we try to apply the completion functions approach to elastic pipelined processors, two issues arise. First, in some states of the elastic processor, instructions

can be duplicated in the data path, *i.e.*, an instruction can reside in two pipeline latches. Such a situation can occur at a fork when the instruction in a buffer before the fork has proceeded along one path of the fork, but the other path is blocked. The latch before the fork has to retain the instruction until both paths are cleared. A direct application of the completion functions-based map to such a state will result in completing the same instruction twice leading to an erroneous refinement map. Second, Elastic half buffers (EHBs) need not have valid tokens. The contents of such EHBs should be ignored and should not be used to update the programmer visible components. Modeling and verification were performed using ACL2-SMT [21].

We introduce token aware completion functions as a method to compute flushing-based refinement maps for elastic pipelined processors. The idea being that EHBs which are either holding duplicate instructions or are in an empty state should not be completed. This is achieved by first computing the reachable states of the elastic controller network. We use token-flow diagrams proposed in [22] to compute the reachable states of the system. The output of the token-flow diagrams is a set of token-states, one token-state for each reachable state. In a token-state, each EHB is assigned a numbered token, which is essentially a natural number. A value of "0" indicates a bubble, *i.e.*, the EHB is empty. Also, EHBs with the same instruction will be assigned the same token numbers. Thus, using the token-state, duplicate instructions and empty EHBs can be identified.

The token-aware completion functions approach [23] works by first computing a two dimensional array, we call *token-array*. Each row in the array corresponds to a

reachable state of the elastic controller network. Each element in a row is a binary value. The number of elements in a row is $2n$, where n is the number of pipeline latches in the elastic system. If *token-array* $(i, j) = 1$, then the contents of EHB H_j in the reachable state S_i should be completed. If *token-array* $(i, j) = 0$, then the contents of EHB H_j in the reachable state S_i should be ignored when computing the refinement map. Given the set of token-states (which are the reachable states represented using numbered tokens) of the elastic controller network of an elastic system, Procedure 1 computes the *token-array* for the elastic system.

Procedure 1:

In: S_R, set of token-states of the elastic controller network and PH, the ordered set of pipeline half buffers. The number of token states ($|SR|$) is r. The number of pipeline half buffers ($|PH|$) is $2n$, where n is the number of pipeline latches. The order of the pipeline half buffers is determined by the position of the buffer in the pipeline, *i.e.*, buffers closer to the end of the pipeline have a higher index.

Out: *token-array* for the elastic system.

(1) Initialize i to 0.

(2) Initialize V_t (the set of visited tokens) to $\{0\}$. The token number "0" represents a bubble. Note that initializing V_t to $\{0\}$ causes the procedure to assign a "0" value to the empty EHBs in the *token-array*.

(3) Initialize j to $2n$.

(4) Let $t = token\ (S_i, PH_j)$, where *token* is a look-up function that gives the token number for EHB PH_j in token-state S_i.

(5) *token-array* $(i, j) = \neg(t \in V_t)$.

(6) Assign $V_t = V_t \cup \{t\}$: adds the token number of EHB PH_j to the visited token set.

(7) If $j\text{-}1 \neq 0$, decrement j and goto step 4.

(8) If $i\text{-}1 \neq 0$, decrement i and goto step 2.

Procedure 2 takes as input the *token-array* and computes the flushing refinement map for the elastic system using completion functions.

Procedure 2:

In: Elastic processor state w: $(P_1, ..., P_m, H_i, ..., H_{2n})$. $P_i, ..., P_m$ are the programmer visible components and $H_1, ..., H_{2n}$ are the pipeline half buffers.

Out: ISA state s obtained by applying the flushing refinement map to w.

(1) Let $S_{2n+1} = (P_1, ..., P_m)$.

(2) Initialize i to $2n$.

(3) r = *reachable-state*(w), gives the number of the reachable elastic controller network state of w, assuming that the reachable states are numbered.

(4) $Si = \begin{cases} (completion\ (S_{i+1}, H_i) & token\text{-}array(r, i) \\ S_{i+1} & otherwise \end{cases}$

(5) If $i\text{-}1 \neq 0$, decrement i and goto step 3.

(6) $s = S_1$

Example: The elastic controller network of the M5 processor model has two reachable states S_1 and S_2. The token-states T_1 and T_2 (given as a vector of token numbers for the

EHBs in M5 in the order (*pc|fd|de|em|mm|l1|l2|l3|l4|l5*) corresponding to these reachable states S_1 and S_2, respectively are: (0,7|0,6|0,5|0,4|0,0|0,0|0,0|0,3|0,0|0,3) and (0,0|7,6|0,5|0,0|0,4|0,7|0,4|0,3|0,3|0,0) [23]. Note that there are two tokens in the token-states for each EB, one corresponding to the master EHB and the other corresponding to the salve EHB. The completion function based refinement map obtained using Procedures 1 and 2 for any state w of processor model M5 whose elastic controller network is state S_1 is: $(pc^s, rf^s, im^s, dm^s) = fdc\ (dec\ (emc\ (mmc\ (pc^w, rf^w, im^s, dm^w), l5^w{}_s), em^w{}_s), de^w{}_s), fd^w{}_s)$.

The completion function based refinement map obtained using Procedures 1 and 2 for any state w of processor model M5 whose elastic controller network is state S_2 is: $(pc^s, rf^s, im^s, dm^s) = fdc\ (fdc\ (dec\ (mmc\ (pc^w, rf^w, im^s, dm^w), mm^w{}_s), de^w{}_s), fd^w{}_s), fd^w{}_m)$.

CHAPTER 7. EXPERIMENTAL RESULTS

The verification and equivalence checking between conventional synchronous pipeline and elastic pipeline were performed using Bit-level Analysis Tool (BAT). The processor models were defined at the bit level using BAT. The verification and equivalence checking between the elastic processor models and the Instruction Set Architecture (ISA) model were performed ACL2-SMT system. In ACL2-SMT, the elastic processor models were defined at term level.

Figure 7.1 shows the invariant constraints for each of the reachable states of the elastic processors M0, M1, ..., M5. In the invariant constraint column, if x is a pipeline latch, x^0, $x^{1/2}$, and x^1 are used to indicate the empty, half, and full states of the latch, respectively.

Figure 7.2 represents mapping function and rank for each of the reachable states of the elastic processors M0, M1, ..., M5. For the refinement map column in the table, we use the following notation. If x is a pipeline latch, x^h_y is used to indicate 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. Note that sometimes it is easy to compute the next value of the program counter, and this value can be used to compute the mapping functions.

Elastic Processor Model	Elastic Controller State	Invariant Constraint
M0	State-1	$pc^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}}$
M1	State-1	$pc^{\frac{1}{2}} \wedge l1^{0} \wedge fd^{0} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}}$
	State-2	$pc^{0} \wedge l1^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{0} \wedge em^{\frac{1}{2}} \wedge mm^{1}$
M2	State-1	$pc^{0} \wedge l1^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{0} \wedge em^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{\frac{1}{2}}$
	State-2	$pc^{\frac{1}{2}} \wedge l1^{0} \wedge fd^{0} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge mm^{1} \wedge l2^{\frac{1}{2}}$
M3	State-1	$pc^{\frac{1}{2}} \wedge l1^{0} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge l3^{0} \wedge mm^{\frac{1}{2}} \wedge l2^{0}$
	State-2	$pc^{0} \wedge l1^{\frac{1}{2}} \wedge fd^{1} \wedge de^{\frac{1}{2}} \wedge em^{0} \wedge l3^{\frac{1}{2}} \wedge mm^{1} \wedge l2^{\frac{1}{2}}$
M4	State-1	$pc^{\frac{1}{2}} \wedge l1^{0} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge l3^{0} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$
	State-2	$pc^{0} \wedge l1^{\frac{1}{2}} \wedge fd^{\frac{1}{2}} \wedge de^{1} \wedge em^{0} \wedge l3^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{0} \wedge l4^{\frac{1}{2}}$
M5	State-1	$pc^{\frac{1}{2}} \wedge l1^{0} \wedge fd^{\frac{1}{2}} \wedge de^{\frac{1}{2}} \wedge em^{\frac{1}{2}} \wedge l3^{0} \wedge mm^{0} \wedge l2^{\frac{1}{2}} \wedge l5^{0} \wedge l4^{\frac{1}{2}}$
	State-2	$pc^{0} \wedge l1^{\frac{1}{2}} \wedge fd^{1} \wedge de^{\frac{1}{2}} \wedge em^{0} \wedge l3^{\frac{1}{2}} \wedge mm^{\frac{1}{2}} \wedge l2^{0} \wedge l5^{\frac{1}{2}} \wedge l4^{\frac{1}{2}}$

Table 7.1. Invariants for elastic processor models.

Elastic Processor Model	Elastic Controller State	Refinement Map							Rank
		pc	*fd*	*de*	*em*	*mm*	*rf*	*dm*	
M0	State-1	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^0	rf^0	dm^0	-
M1	State-1	pc_s^0	fd_s^{-1}	de_s^{-1}	em_s^{-1}	mm_s^{-1}	rf^{-1}	dm^{-2}	0
	State-2	$l1_s^1$	fd_s^0	de_s^{-1}	em_s^{-1}	mm_s^0	rf^{-1}	dm^{-1}	1
M2	State-1	$l1_s^1$	fd_s^0	de_s^{-1}	em_s^{-1}	mm_s^{-1}	rf^0	dm^{-2}	0
	State-2	pc_s^0	fd_s^{-1}	de_s^2	em_s^{-2}	mm_s^{-2}	rf^{-1}	dm^{-3}	1
M3	State-1	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^0	rf^0	dm^0	1
	State-2	$l1_s^0$	fd_s^0	de_s^0	em_s^{-1}	mm_s^0	rf^0	dm^{-1}	0
M4	State-1	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^0	1
	State-2	$l1_s^0$	fd_s^{-1}	de_s^0	em_s^{-1}	mm_s^{-2}	rf^{-1}	dm^{-1}	0
M5	State-1	pc_s^0	fd_s^0	de_s^0	em_s^0	mm_s^{-1}	rf^0	dm^0	1
	State-2	$l1_s^0$	fd_s^0	de_s^0	em_s^{-1}	mm_s^{-2}	rf^0	dm^{-1}	0

Table 7.2. Refinement maps and ranks for elastic processor models.

The token values in the synchronous token-state for the other latches should then be suitably adjusted. The invariant constraints and mapping functions were obtained using the procedures described in chapter 5 and chapter 6.

The verification results are shown in Table 7.3. The refinement proofs were automatically checked using the BAT decision procedure version 0.2 [16]. The experiments were conducted on a 1.8GHz Intel (R) Core(TM) Duo CPU, with an L1 cache size of 2048KB. As can be seen from the table, all the elastic 5-stage DLX-based processors were verified against the synchronous DLX within 25 seconds, thereby demonstrating the high efficiency of our approach.

Elastic Processor Models	Verification Times [sec]		CNF Statistics		
	Siege	Total	Variables	Clauses	Literals
M0	0.04	0.49	1,723	5,161	29,648
M1	15.74	19.25	9,688	43,633	252,084
M2	18.70	22.57	10,245	46,736	263,765
M3	13.20	16.18	8,538	36,735	210,251
M4	13.86	18.91	9,100	40,357	221,767
M5	13.95	17.15	8,706	36,673	205,527

Table 7.3. Verification time for proving equivalence properties between elastic processor and synchronous processor.

The verification time need for equivalence checking between elastic processor and instruction set architecture is shown in Table 7.3. The token-aware completion functions approach was used to verify safety for six elastic pipelined processors M0, ..., M5. The results are shown in Table 7.4.

Elastic Models	Yices		Total Time (sec)
	Bool Vars	Time (sec)	
M0	887	0.22	1.07
M1	1,101	0.46	2.29
M2	1,889	1.05	5.07
M3	2,811	1.32	6.17
M4	3,096	3.29	16.65
M5	3,605	4.16	24.42

Table 7.4. Verification time for proving equivalence properties between elastic processor and instruction set architecture.

Verification was performed using the ACL2-SMT system. The ACL2-SMT system incorporates a translator that reduces the correctness theorem to a decision problem in the form of a formula in a decidable logic that Yices can handle. The decision problem is then checked by Yices. Column "Bool Vars" gives the number of Boolean variables in the decision problem. The experiments were conducted on a 1.8GHz Intel (R) Core(TM) Duo CPU, with an L1 cache size of 2048KB. As can be seen from the table, each of the elastic 5-stage DLX-based processors were verified against the high-level instruction set architecture (ISA) within 25 seconds, thereby demonstrating the high efficiency of our approach.

CHAPTER 8. CONCLUSION

We have introduced highly automated and efficient approaches for checking the equivalence of elastic pipelined processors against their specifications. The methods were demonstrated using 6 elastic 32-bit DLX-based processors defined at the bit-level. We have developed a method for checking the correctness of elastic pipelined processors against their high-level instruction set architectures. The approach was demonstrated by verifying 6 DLX-based elastic processor models.

For future work, a tool flow can be developed that will incorporate these verification methods. The work can be further extended to explore the verification of elastic processors that incorporate features such as out-of-order execution and register renaming.

REFERENCES

[1] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In E. Sentovich, editor, Design Automation Conference (DAC'06), pages 657–662. ACM, 2006.

[2] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In Formal Methods in Computer-Aided Design (FMCAD' 06), pages 19–30. IEEE Computer Society, 2006.

[3] 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, 20(9):1059–1076, 2001.

[4] P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, Formal Methods in Computer-Aided Design– FMCAD 2000, volume 1954 of LNCS, pages 161–178. Springer-Verlag, 2000.

[5] P. Manolios. Mechanical Verification of Reactive Systems. PhD thesis, University of Texas at Austin, August 2001. See URL http://www.cc.gatech.edu/_manolios/~ publications.html.

[6] P. Manolios, S. Srinivasan. Refinement Maps for Efficient Verification of Processor Models. In Proceedings of the conference on Design, Automation and Test in Europe (DATE '05), pages 1304 - 1309. IEEE Computer Society, 2005.

[7] P. Manolios, S. Srinivasan. Automatic verification of safety and liveness for pipelined machines using WEB refinement. In ACM Transactions on Design Automation of Electronic Systems, vol. 13, no. 3, July 2008.

[8] P. Manolios, S. Srinivasan. Automatic verification of safety and liveness for xscalelike processor models using web refinements. In Design, Automation and Test in Europe Conference and Exposition (DATE '04), pages 168-175, IEEE Computer Society, 2004.

[9] P. Manolios. A compositional theory of refinement for branching time. In Proc. 12th IFIP WG 10.5 Adv. Res. Work. Conf. (CHARME), 2003, vol. 2860, pp. 304-318.

[10] P. Manolios and S. K. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In Proc. Int. Conf. Comput.- Aided Des. (ICCAD), 2005, pp. 863–870.

[11] P. Manolios and S. K. Srinivasan. A refinement-based compositional reasoning framework for pipelined machine verification. In IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 16 , no. 4, pages 353-364, April 2008.

[12] P. Manolios, S. K. Srinivasan, and D. Vroon, "BAT: The bit-level analysis tool," 2006 [Online]. Available: http://www.cc.gatech.edu/~manolios/ bat/

[13] P. Manolios, S. K. Srinivasan, and D. Vroon, "BAT: The bit-level analysis tool," in Proc. Int. Conf. Comput.-Aided Verification (CAV), 2007, pp. 303–306.

[14] J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and Design of Synchronous Elastic Circuits. In TAU, Proceedings of the ACM/IEEE International Workshop on Timing Issues 2006.

[15] M. Kishinevsky, J. Cortadella, B. Grundmann, S. Krstic, J. O'Leary. In Computer Science- Theory and Applications, Vol 3967/2006, April 2006.

[16] P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for rtl model verification. In S. Hassoun, editor, ICCAD, pages 786–793. ACM, 2006.

[17] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In Computer-Aided Verification (CAV '94), volume 818 of LNCS, pages 68-80. Springer-Verlag, 1994.

[18] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In N. Halbwachs and D. Peled, editors, Computer-Aided Verification–CAV '99, volume 1633 of LNCS. Springer-Verlag, 1999.

[19] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. Computer-Aided Verification , CAV '98, volume 1427 of LNCS, Vancouver, BC, Canada, June/July 1998. Springer-Verlag.

[20] R. Hosabettu. PVS specification and proofs of all the examples verified with the completion function approach, 1999. Available at http://www.cs.utah.edu/~hosabett/pvs/processor.html.

[21] S. K. Srinivasan. Efficient Verification of Bit-Level Pipelined Machines Using Refinement. PhD thesis, Georgia Institute of Technology, December 2007. See URL http://etd.gatech.edu/theses/available/-etd-08242007-111625/.

[22] S. K. Srinivasan, K. Sarker, and R. S. Katti. Verification of Synchronous Elastic Processors. IEEE Xplore: Embedded Systems Letters, volume. 1, no. 1, pages 14-18, May 2009.

[23] S. K. Srinivasan, K. Sarker, and R. S. Katti. Token-Aware Completion Functions for Elastic Processor Verification. Journal of Electrical and Computer Engineering, vol.2009, Article ID 480740, 5 pages, 2009. doi:10.1155/2009/480740.