Theory of composable latency-insensitive refinements

by

Muralidaran Vijayaraghavan

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author .....
Department of Electrical Engineering and Computer Science
February 27, 2009

Certified by .....
Arvind
Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chair, Department Committee on Graduate Students

Theory of composable latency-insensitive refinements

by

Muralidaran Vijayaraghavan

Submitted to the Department of Electrical Engineering and Computer Science
on June, 2009, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Simulation of a synchronous system on a hardware platform, for example an FPGA, can be performed using a hardware prototype of the system. But the prototype may not meet the resource and timing constraints of that platform. One way to meet the constraints is to partition the prototype hierarchically into modules, and to refine the individual modules while preserving the overall behavior of the system. In this thesis we formalize the notion of a refinement that preserves the behavior of the original modules – we call such refinements *latency-insensitive refinements.* We show that if these latency-insensitive refinements of the modules obey certain conditions, then these refinements can be composed together hierarchically in order to obtain the latency-insensitive refinement of the original system. We call the latency-insensitive refinements that obey these conditions as *composable latency-insensitive refinements.* We also give a procedure to automatically transform a module to a latency-insensitive refinement while obeying the conditions that enable it to be composed hierarchically. The transformation serves as a starting point for making further refinements and optimizations, and thus, gives a methodology to design hardware simulators for synchronous systems.

Thesis Supervisor: Arvind
Title: Professor

Acknowledgments

First and foremost, I would like to thank Arvind for being a dedicated advisor and an inspiring teacher. It was his encouragement and persistence that made me rigorize the concepts to make this thesis possible.

I would also like to thank Joel Emer for introducing me to this idea and enabling me to have hands on experience on FPGA simulators in Intel.

I would then like to thank everyone in CSG for their advice and support.

Last, but not the least, I would like to thank my family for their love and encouragement and my friends who made me feel at home in Cambridge.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Micro-architecture designs have to be evaluated in order to get a precise evaluation of the performance. Architects rely on simulators in order to carry out the performance evaluation. These simulators are typically written in a sequential language like C or C++. Various benchmarks are run on these simulators in order to evaluate the performance of the micro-architecture. A typical design goes through several such iterations by the architects before deciding a particular micro-architecture.

Simulators are judged by the following three characteristics:

1. Simulation speed – This is the speed at which the simulator simulates the micro-architecture. This is different from the speed of the micro-architecture that is modeled. If two simulators are modeling the same micro-architecture, one is said to be faster than the other if it runs the benchmarks faster.

2. Simulation accuracy – This is the measure of the accuracy of the performance numbers obtained from the simulator with respect to the original micro-architecture. A more detailed model gives more accurate numbers.

3. Development time of the simulator – This is the amount of time it takes to encode the simulator.

A software simulator has a short development time compared to a detailed hardware description code. Furthermore, architects usually have their own simulation

Simulator detail (accuracy)	Order of magnitude of simulator speed (in Thousands of instructions per second)
Low-detail model	100
Medium-detail model	10
High-detail model	1

Table 1.1: Trade-off between speed and accuracy in a software simulator

infrastructure which helps to develop software simulators rapidly. But the problem with software simulators is that they are slow. Micro-architectures are fine-grained parallel structures. Mapping them accurately into a software simulator drastically reduces the speed of simulation as it sequentializes all the parallel events. Architects use less detailed models in order to speed up the simulation time. As a consequence of this, simulation accuracy decreases. Table 1.1 shows the trade-off between speed and accuracy in software simulators used by Intel [10].

One can think of speeding up highly detailed software simulators by running it on several cores, thus exploiting the parallelism. But micro-architectures of processors and memory systems are fine-grained parallel systems. In a detailed simulator, each component communicates with several other components every simulated cycle. Exploiting this granularity of parallelism in multicores is not possible without incurring a huge communication cost. So using parallel software simulators on multicores is not going to speed up the simulation of micro-architectures, in fact the communication overhead may be so high, it might make it slower than sequential simulators.

Field Programmable Gate Arrays (FPGAs) are programmable logic devices that can be programmed using hardware description code. FPGAs, unlike multicores, offer a fine-grained parallel execution substrate which can potentially be used for detailed simulation of micro-architectures. This allows us to have orders of magnitude improvement in simulation speed, while not sacrificing the accuracy. The development time for a simulator on an FPGA could be substantial, but the improvement in simulation speed as well as accuracy, more than makes up for it.

The synchronous circuit description of a micro-architecture design can be directly implemented on an FPGA – this is called *prototyping* the design on an FPGA. The prototype can be directly used for simulation and performance studies. But the disad-

vantage with this approach is that the prototype has to meet the resource and timing constraints of the FPGA platform without any modifications, or else the simulation will not be accurate. There are several commonly used hardware structures that map inefficiently on FPGAs. For example, the Content Addressable Memory (CAM) consumes a lot of FPGA resources and significantly increases the critical path of the entire design. Multi-ported register file is another such example. FPGAs have a limited amount of single-ported or dual-ported address-indexed memory, but no CAMs or multi-ported memory, which is why these structures do not map well onto FPGAs.

FPGA platforms often have other peripherals like the Dynamic Random Access Memory (DRAM), and may contain multiple FPGAs connected by a communication network. These peripherals and communication network can be used for prototyping a complete system. For example, in order to prototype a multi-core architecture along with the memory subsystem, one could implement the individual cores in each of the FPGAs, use the FPGA platform's communication network to model the on-chip interconnect, and use the platform's DRAM to model the memory subsystem. But the issue with this is that the speed of the FPGAs with respect to the peripherals or the network can be vastly mismatched compared to the speeds in the system which we want to model. This again boils down to not being able to map components of the system well into an FPGA platform, though, unlike the previous problem, this can not be solved by relaxing the resource and timing constraints.

This brings us to the general problem we are trying to solve. Given an synchronous design, we refine the design so that it meets the resource and timing constraints, while preserving the behavior of the original design. In this thesis, we formally define what is meant by a refinement preserving the behavior of a synchronous design – we call such refinements as *latency-insensitive refinements*. We abstract the synchronous design as a Finite State Machine (FSM).

The rest of the thesis is organized as follows. Chapter 2 formalizes the definition of *latency-insensitive refinements*. In this chapter, we also state and prove the conditions that the refinements of individual modules of the design should obey in order to be able to compose these refinements hierarchically. This is important be-

cause synchronous circuits are typically designed as a hierarchy of modules, and so refining the design involves refining the individual modules and hierarchically composing them back. In Chapter 3, we give a methodology to automatically transform a module into a refinement, which satisfies these conditions. We also discuss how further refinements and optimizations can be made to this transformation. Finally, in Chapter 4, we conclude.

Chapter 2

Latency-Insensitive refinements

2.1 Definition of latency-insensitive refinements

A latency-insensitive refinement of an FSM is a system which simulates the observable behavior of the FSM. We make an assumption that only the outputs of the FSM are observable; if some internal state of the FSM is observable, then we can add an output which contains information about the internal state, thus making it observable. Furthermore, we assume that the FSM does not have any combinational loop.

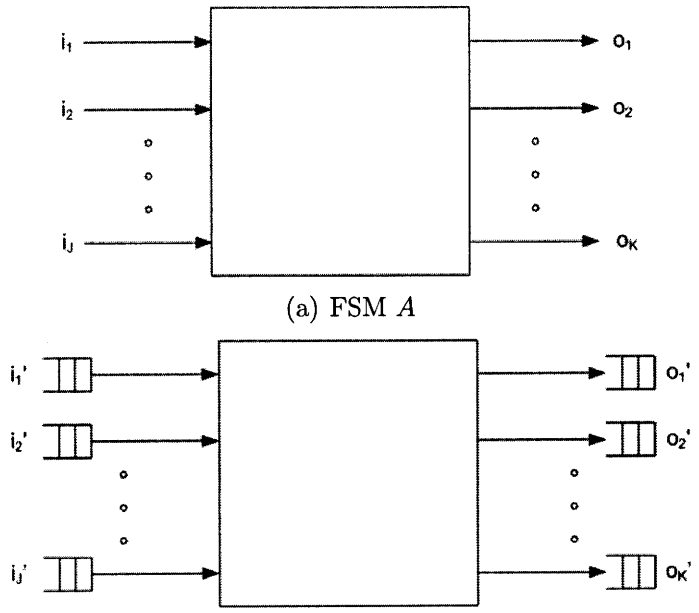
We formally define the latency-insensitive refinement of an FSM below.

Consider an FSM A (Figure 2-1(a)) with inputs $i_1, i_2, \ldots, i_J$ and outputs $o_1, o_2, \ldots, o_K$. Let A' (Figure 2-1(b)) be such that, for each input i_j in A, there is a corresponding First In First Out (FIFO) input buffer i'_j in A'; and for each output o_j in A, there is a corresponding FIFO output buffer o'_j in A'. Every buffer of A' can be enqueued only if it is not full, and dequeued only if it is not empty.

A' is said to be a latency-insensitive refinement of A if

1. **A' is cycle-accurate with respect to FSM A**

   Let $\overline{i_j}$ denote the cycle by cycle sequence of values on input i_j of A. Similarly, let $\overline{o_k}$ denote the cycle by cycle sequence of values on output o_k of A. Let $\overline{i'_j}$ denote the sequence of values enqueued into input buffer i'_j of A' and let $\overline{o'_k}$ denote the sequence of values dequeued from output buffer o'_k of A'. If A' is a



(a) FSM A

(b) Latency-insensitive refinement A'

Figure 2-1: Latency-insensitive refinement of an FSM

cycle-accurate refinement of A, then

$$\forall j, \overline{i'_j} = \overline{i_j} \Rightarrow \forall k, \overline{o'_k} = \overline{o_k} \tag{2.1}$$

2. **A' is deadlock-free**

   A' is deadlock-free if it has the following property. If A' has a source which eventually enqueues into each of the input buffer when it is not full and a sink which eventually dequeues from each of the output buffer when it is not empty, then no input buffer will remain full forever, and no output buffer will remain empty forever.

   This means that the source can keep enqueuing into each input buffer and the sink can keep dequeuing from each output buffer, thus guaranteeing forward progress.

2.2 Definition of dependency

Consider a system which has an interface similar to A' shown in Figure 2-1(b), though this system need not be a latency-insensitive refinement. We define the notion of dependency between an output buffer o'_k and an input buffer i'_j as follows.

Definition 2.2.1. An output buffer o'_k is said to be *dependent* on input buffer i'_j if, $\exists n \geq 0$, and there exists $n-1$ values enqueued into i'_j and n values enqueued into every other input buffer, such that the n^{th} value dequeued from o'_k changes if the n^{th} value enqueued into i'_j changes. We also say that i'_j *influences* o'_k.

Lemma 2.2.0.1. *If i_j and o_k are input and output buffers, respectively, of an FSM A, and i'_j and o'_k are the corresponding input and output buffers, respectively, of A' which is a latency-insensitive refinement of A, if o'_k is dependent on i'_j in A' then o_k is combinationally connected to i_j in A.*

Proof. If o_k is not combinationally connected to i_j in A, then $\forall n \geq 0$, there exists no cycle by cycle sequence of $n-1$ value of i_j and no cycle by cycle sequence of n values of the rest of the inputs, for which the value of o_k in the n^{th} cycle changes when the value of i_j in the n^{th} cycle changes. But this will contradict the fact that A' is cycle-accurate with respect to A, as in A' the value dequeued from o'_k changes if the value enqueued in i'_j changes. □

2.3 Composable cycle-accurate refinements

Large FSMs are usually designed by designing smaller FSMs and composing these smaller FSMs hierarchically. We describe the rules of composition below.

1. **Parallel Composition of FSMs**

   Figure 2-2 shows the parallel composition of two FSMs A_1 and A_2 to get A. Let us represent it as follows.

   $$A = A_1 + A_2$$

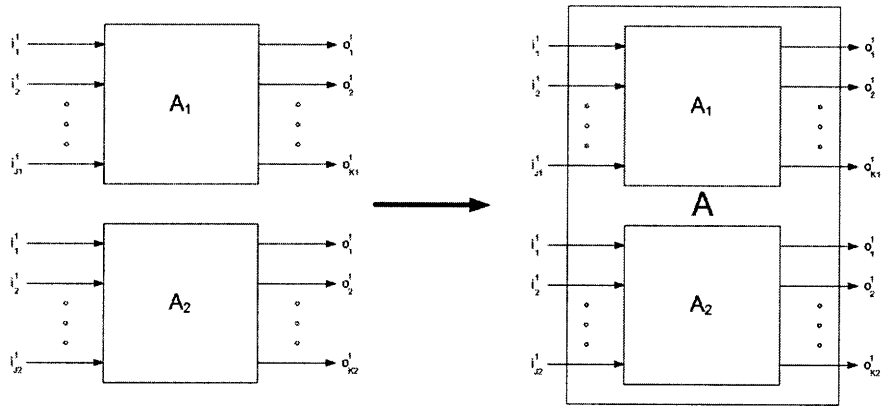


Figure 2-2: Parallel composition of FSMs

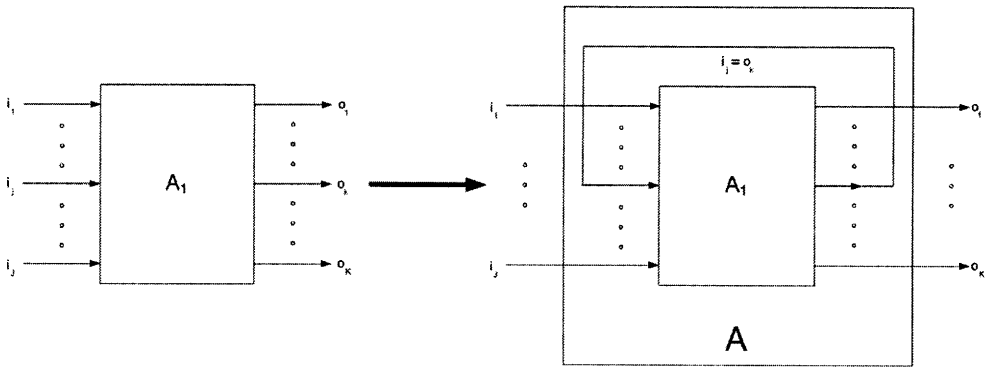


Figure 2-3: (j, k) Iterative composition of FSMs

2. **Iterative composition of FSMs**

   Figure 2-3 shows the (j, k) iterative composition of an FSM A_1 to get A. Input i_j is connected to output o_k. Let us represent it as follows.

   $$A = (j, k) \cdot A_1$$

   The (j, k) iterative composition of an FSM A_1 is legal only if output o_k is not combinationally connected to the input i_j. This prevents introduction of a combinational loop during composition.

The two rules do not introduce any combinational loops during composition. So if the original smaller FSMs do not have combinational loops, their composition will not introduce any combinational loops in the final composed FSM.

It is desirable to be able to compose the latency-insensitive refinements of FSMs so that the composition of the refinements of FSMs is a latency-insensitive refinement of the composition of the FSMs. In order for this to be true, the latency-insensitive refinements of the FSMs should obey the following conditions.

Condition 2.3.1. ***Dependency condition***

$\forall n \geq 0$, if every input buffer of the refinement has been enqueued n times and every output buffer has been enqueued n times, now if all input buffers that an output buffer depends is enqueued once more, and the output buffer is not full, then the output buffer will eventually be enqueued once more.

In particular, Dependency condition 2.3.1 is true even if the source has enqueued some of the input buffers only n times and stops, or if the sink stops dequeuing from some of the output buffers so as to make them full, after being enqueued n times. We illustrate the subtleties of the condition through the following examples.

1. Example 1: If a refinement obeying the condition has an output buffer o that does not depend on any input buffer, if all the output buffers have been enqueued n times and if all the input buffers have been dequeued n times, now even if all

input buffers are empty, and if o is not full, then it will be enqueued once more. This will happen even if none of the input buffers are ever enqueued henceforth.

2. Example 2: If a refinement obeying the condition has two output buffers o_1 and o_2, if all the output buffers have been enqueued n times, and if all the input buffers have been dequeued n times, now if some of the input buffers which o_1 depends on are empty, if none of the input buffer which o_2 depends on is empty, and if o_2 is not full, then it will be enqueued once more. This will happen even if the input buffers that o_1 depends on are never enqueued henceforth. o_2 should not depend on whether o_1 has been enqueued, thus there is no forced order in enqueuing o_1 and o_2.

Condition 2.3.2. ***Cleanup condition***

$\forall n \geq 0$, if every output buffer of the refinement has been enqueued n times, and if every input buffer has been enqueued n times, then eventually every input buffer will be dequeued n times.

In particular, Cleanup condition 2.3.2 is true even if the sink stops dequeuing from some of the output buffers so as to make them full.

Definition 2.3.1. We call a latency-insensitive refinement obeying Dependency condition 2.3.1 and Cleanup condition 2.3.2 as *composable latency-insensitive refinements.*

Theorem 2.3.1. *If a system A' (Figure 2-1(b)) obeys Dependency condition 2.3.1 and Cleanup condition 2.3.2, then A' is deadlock free.*

Proof. We prove that $\forall n \geq 0$, if n values are enqueued into every input buffer, then n values will be enqueued eventually into every output buffer provided there is a sink to dequeue the output buffers whenever they are not empty, and n values will be dequeued eventually from every input buffer. This implies that the output buffers will never remain empty forever and the input buffers will never remain full forever. This proves that A' is deadlock-free.

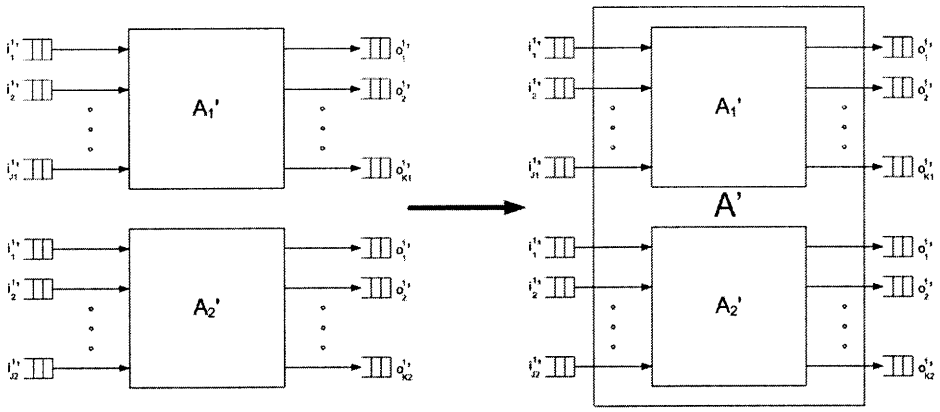


Figure 2-4: Parallel composition of composable latency-insensitive refinements

We prove this by induction over n. For $n = 0$, the hypothesis is trivially true. Let us assume that the hypothesis is true for $n \leq m$. We now prove that the hypothesis is true for $n = m + 1$. If $m + 1$ values are enqueued into every input buffer, for every output buffer, all the input buffers that it depends on have been enqueued $m + 1$ times. Thus, by Dependency condition 2.3.1, all the output buffers will eventually be enqueued once more, i.e. $m + 1$ times in total. Now, by Cleanup condition 2.3.2, since all output buffers are enqueued $m + 1$ times, and all input buffers are enqueued $m + 1$ times, all input buffers will eventually be dequeued $m + 1$ times. This proves the induction hypothesis for $n = m + 1$. □

Thus, in order to prove that a system A' is a composable latency-insensitive refinement of an FSM A, it is enough to prove that A' is cycle-accurate with respect to A, and A' obeys Dependency condition 2.3.1 and Cleanup condition 2.3.2.

2.3.1 Composing composable latency-insensitive refinements

We now describe the rules of composition for composable latency-insensitive refinements.

1. **Parallel Composition of composable latency-insensitive refinements**

   Figure 2-2 shows the parallel composition of two latency-insensitive refinements

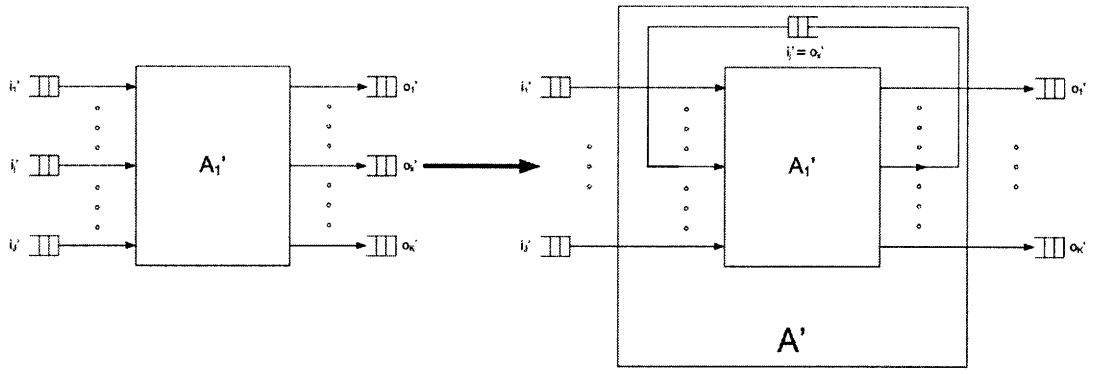


Figure 2-5: (j, k) Iterative composition of composable latency-insensitive refinements

A'_1 and A'_2 to get A'. Let us represent it as follows.

$$A' = A'_1 \oplus A'_2$$

2. **Iterative composition of composable latency-insensitive refinements**

   Figure 2-3 shows the (j, k) iterative composition of A'_1 to get A'. Input buffer i'_j is merged with output buffer o'_k. Let us represent it as follows.

$$A' = (j, k) \odot A'_1$$

   The (j, k) iterative composition of a refinement A'_1 is legal only if output buffer o'_k does not depend on input buffer i'_j.

Theorem 2.3.2. *Composable latency-insensitive refinements of FSMs are composable, i.e.*

- *If A'_1 and A'_2 are composable latency-insensitive refinements of FSMs A_1 and A_2, then $A' = A'_1 \oplus A'_2$ is a composable latency-insensitive refinement of $A = A_1 + A_2$.*
- *If A'_1 is a composable latency-insensitive refinement of FSM A_1, then $A' = (j, k) \odot A'_1$ is a composable latency-insensitive refinement of $A = (j, k) \cdot A_1$.*

Proof. Proving the theorem for parallel composition of the refinements is trivial as the individual refinements A'_1 and A'_2 are simply juxtaposed in the parallel composition without any interaction.

Let us now prove the theorem for iterative composition of a refinement.

Lemma 2.3.2.1. *If A is legal, then A' is legal.*

Proof. If A is legal, then, in A_1, o_k is not combinationally connected to i_j from the definition of (j,k) iterative composition of an FSM. From Lemma 2.2.0.1, o'_k is not dependent on i'_j in A'_1. Thus (j,k) iterative composition of A'_1 is legal. Thus A' is legal. □

Lemma 2.3.2.2. *In A', $\forall n \geq 0$, if every input buffer of A' has been enqueued n times and every output buffer has been enqueued n times, then the merged buffer $i'_j = o'_k$ will be enqueued and dequeued n times eventually.*

Proof. We prove this by induction on n. For $n = 0$, the induction hypothesis is trivially true. Let us assume that the hypothesis is true for $\forall n \leq m$. We prove that it is true for $n = m + 1$. Consider the refinement A'_1 inside A'. In A'_1, all its input buffers except i'_j has been enqueued $m+1$ times, and all its output buffers have been enqueued $m + 1$ times. Output buffer o'_k in A'_1 can not depend on i'_j by definition of an iterative composition. So, all the input buffers that output o'_k depends on have been enqueued $m + 1$ times. Thus, o'_k will be enqueued $k + 1$ times, since A'_1 obeys Dependency condition 2.3.1. Since the buffers i'_j is merged with o'_k, all the input buffers of A'_1 have now been enqueued $m + 1$ times, and all its output buffers have been enqueued atleast m times. Hence, again since A'_1 obeys Dependency condition 2.3.1, all its output buffers will be enqueued once more, i.e. $m + 1$ times in total. Now, since A'_1 obeys Cleanup condition 2.3.2, all its input buffers will be dequeued once more, in particular i'_j will be dequeued once more, i.e. $m + 1$ times in total. Thus we have proved the induction hypothesis. □

Lemma 2.3.2.3. *A' obeys Dependency condition 2.3.1*

Proof. By Lemma 2.3.2.2, the merged buffer $i'_j = o'_k$ will be enqueued n times, given that every input buffer of A' has been enqueued n times and every output buffer has been enqueued n times. Let all the input buffers that an output buffer o'_l depends on be enqueued $n + 1$ times. Consider the following two cases.

1. In A'_1, if o'_l did not depend on i'_j, then by Dependency condition 2.3.1, o'_l will be enqueued $n+1$ times.

2. In A'_1, if o'_l depended on i'_j, then the input buffers that o'_k depends on must be a subset of the input buffers that o'_l depends on. This is because in FSM A_1, o_l is combinationally connected to i_j, which is connected to o_k. If o_k is combinationally connected to any input, then o_l should also be combinationally connected to that input. So, in A'_1, this means that all the input buffers that o'_k depends on have been enqueued $n+1$ times. o'_k will be enqueued $n+1$ times as A'_1 obeys Dependency condition 2.3.1. Hence i'_j will be enqueued $n+1$ times. Now, in A', all the input buffers that o'_l depends on have been enqueued $n+1$ times. So, o'_l will be enqueued $n+1$ times eventually, again as A'_1 obeys Dependency condition 2.3.1.

□

Lemma 2.3.2.4. *A' obeys Cleanup condition 2.3.2*

Proof. By Lemma 2.3.2.2, the merged buffer $i'_j = o'_k$ has been enqueued n times. So, for A'_1, every input buffer has been enqueued n times, and every output buffer has been enqueued n times. So, every input buffer of A'_1 will be dequeued n times eventually, as A'_1 obeys Cleanup condition 2.3.2. This means that every input buffer of A' will be dequeued n times eventually. □

Lemma 2.3.2.5. *A' is cycle-accurate with respect to A.*

Proof. We prove the following. $\forall n \geq 0$, if the n values enqueued into all the input buffers of A' is exactly the same as the cycle by cycle sequence of n values in all the inputs of A, then

1. the n values enqueued or dequeued from the merged buffer $i'_j = o'_k$ of A' will be exactly the same as the cycle by cycle sequence of n values in the connection $i_j = o_k$ of A.

2. the n values dequeued from all the output buffers of A' will be exactly the same as the cycle by cycle sequence of n values in all the outputs of A.

We prove this by induction on n.

The hypothesis is trivially true for $n = 0$. Let us assume that the hypothesis is true for $n = m$. We prove that the hypothesis is true for $n = m + 1$.

By the definition of an iterative composition, output buffer o'_k in A'_1 can not depend on i'_j. So, in A_1, the $(m+1)^{\text{th}}$ value enqueued in o'_k is not influenced by the $(m+1)^{\text{th}}$ value enqueued in i'_j. Thus, if the rest of the $(m+1)^{\text{th}}$ values enqueued in the input buffers of A'_1 are the same as the values of all the corresponding inputs of A_1 during the $(m+1)^{\text{th}}$ cycle, then, since A'_1 is cycle-accurate with respect to A_1, the $(m+1)^{\text{th}}$ value enqueued in o'_k is the same as the value of o_k in A_1 during the $(m+1)^{\text{th}}$ cycle. This proves the first part of the hypothesis.

Now, since i'_j is the same as o'_k, the sequence of $m+1$ values enqueued in all the input buffers of A'_1 matches with the cycle by cycle sequence of $m+1$ values of all the inputs of A_1. Again since A'_1 is cycle-accurate with respect to A_1, the sequence of $m+1$ values dequeued from all the output buffers of A'_1 should match with the cycle by cycle sequence of $m+1$ values of all the outputs of A_1. In particular, the sequence of $m+1$ values dequeued from all the output buffers of A' should match with the cycle by cycle sequence of $m+1$ values of all the outputs of A. This proves the second part of the hypothesis. □

Lemmas 2.3.2.1, 2.3.2.3, 2.3.2.4 and 2.3.2.5 prove that A' is a latency-insensitive refinement of A. □

2.4 Modular composable latency-insensitive refinement of FSMs

In this section we describe the methodology to modularly convert an FSM into a composable latency-insensitive refinement.

Given an FSM A, Algorithm 1 gives the methodology to modularly convert A into a composable latency-insensitive refinement.

Algorithm 1 Modular conversion of an FSM A into a composable latency-insensitive refinement

1: Perform a cut on A to partition A into two FSMs A_1 and A_2.

2: Recursively apply Algorithm 1 on both the partitions A_1 and A_2.

3: Let A_1' and A_2' be the composable latency-insensitive refinements of A_1 and A_2, respectively. Apply parallel composition of A_1' and A_2' to obtain A'.

4: $\forall(j,k)$ such that i_j is an input to A_1 and o_k is an output to A_2, such that i_j was connected to o_k in A before performing the cut, apply (j,k) iterative composition on A'. Let A'' be the resulting refinement after applying all such iterative compositions.

5: $\forall(m,n)$ such that i_m is an input to A_2 and o_n is an output to A_1, such that i_m was connected to o_n in A before performing the cut, apply (m,n) iterative composition on A''. Let A''' be the resulting refinement after applying all such iterative compositions. A''' is the composable latency-insensitive refinement of A.

Figure 2-6 shows the application of one cut on FSM A and then transforming A into its composable latency-insensitive refinement.

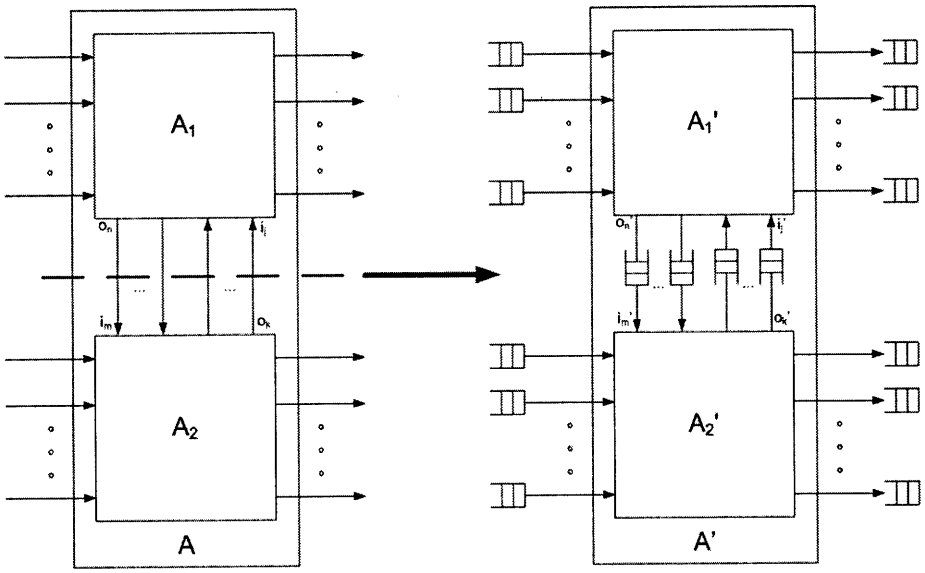


Figure 2-6: Modular composable latency-insensitive refinement of FSMs

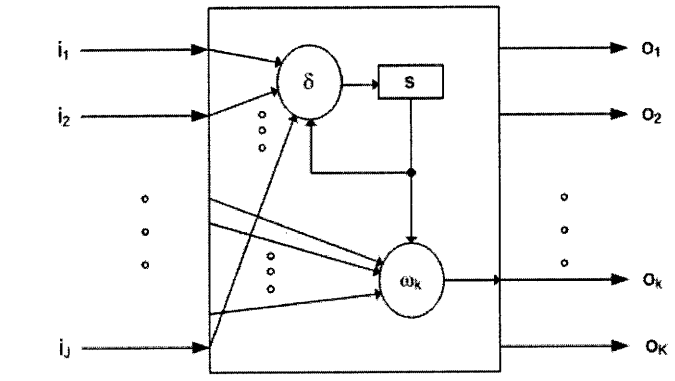
Chapter 3

The latency-insensitive transform

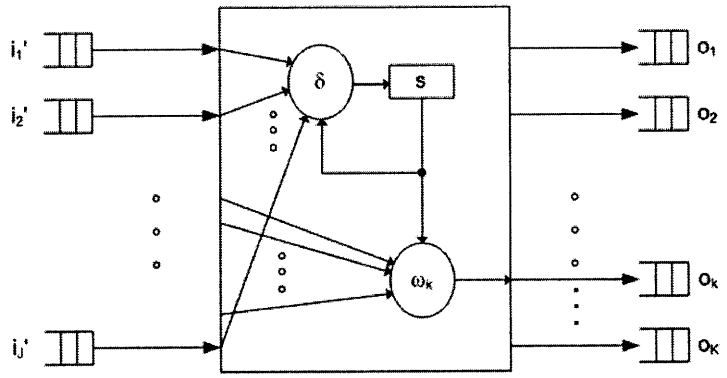
3.1 Latency-insensitive transform of an FSM

In this section, we define the *latency-insensitive transform* of an FSM. This transform is a particular composable latency-insensitive refinement of an FSM, which can be automatically generated from the FSM. We discuss the procedure to convert an FSM into its latency-insensitive transform and prove that it is indeed a composable latency-insensitive refinement. The transform serves as a concrete guideline for designing composable latency-insensitive refinements of FSMs, as opposed to an abstract set of conditions that a latency-insensitive refinement should obey in order to be composable as discussed in the previous chapter.

Consider an FSM A shown in Figure 3-1(a). Let $i_1, i_2, , \ldots, i_J$ be its inputs. For each $j \in \{1, 2, \ldots, J\}$, let I_j denote the set of values that input i_j can take. Let $o_1, o_2, \ldots, o_K$ be its outputs. For each $k \in \{1, 2, \ldots, K\}$, let O_k denote the set of values that output o_k can take. Let s be the state of FSM A, and S be the set of values that state s can take. The state transition function is given by $\delta : S \times [I_1 \times I_2 \times \ldots \times I_J] \rightarrow S$. Each output o_k is combinationally connected to state s and strictly combinationally connected to R_k inputs $i_{k_1}, i_{k_2}, \ldots, i_{k_{R_k}}$ (where $\{k_1, k_2, \ldots, k_{R_k}\} \subset \{1, 2, \ldots, J\}$). By o_k being strictly combinationally connected to i_j, we mean that if the n^{th} value of i_j changes, while the rest of the inputs remain the same for all cycles, then the n^{th} value of o_k changes. Thus, the value in o_k is given



(a) FSM A



(b) Latency-insensitive transform A'

Figure 3-1: Latency-insensitive transform of an FSM

by output function $\lambda_k : S \times [I_{k_1} \times I_{k_2} \times \ldots \times I_{k_{R_k}}] \rightarrow O_k$.

Algorithm 2 gives the behavior of FSM A during each cycle. $i_j(t)$ represents the value of input i_j during cycle t and $o_k(t)$, the value of output o_k during cycle t. $s(t)$ represents the state of A during cycle t. The clock cycles start from 1.

Algorithm 2 Behavior of FSM A during cycle t

1: **for all** $k \in \{1, 2, \ldots, K\}$ **do**
2: Calculate value of output o_k during cycle t using

$$o_k(t) = \lambda_k(s(t), [i_{k_1}(t), i_{k_2}(t), \ldots, i_{k_{R_k}}(t)]) \tag{3.1}$$

3: **end for**
4: Calculate the new state of A. This is the state of the FSM for the next cycle $t+1$. So,

$$s(t+1) = \delta(s(t), [i_1(t), i_2(t), \ldots, i_J(t)]) \tag{3.2}$$

Update state s with $s(t+1)$.

Consider the latency-insensitive transform A' of A shown in Figure 3-1(b). For every input i_j in A, there is a corresponding FIFO input buffer i'_j in A'; for every output o_k in A, there is a corresponding FIFO output buffer o'_k in A'. State s' of A' corresponds to state s of A. The set of values that can be enqueued into an input buffer i'_j of A' is I_j which is the same as the set of values the corresponding input i_j of A can take. Similarly, the set of values that can be dequeued from an output buffer o'_k of A' is O_k; and the set of values that state s' can take is S. The state s' of A' is initialized with the same value as the state s of A. One can enqueue into the input buffer i'_j only when it is not empty and similarly, one can dequeue from the output buffer o'_k only when it is not full.

Transform A' simulates each cycle of A. We denote this as *target cycle* of A'. Algorithm 3 gives the behavior of A' during each target cycle. Here vi'_j represents the value dequeued from input buffer i'_j and vo'_k represents the value enqueued into the output buffer o'_k. vs' represents the value read from state s' and ns' represents the new value of state s'.

Lemma 3.1.0.6. *Output buffer o'_k in the transform A' is dependent on input buffer i'_j according to dependency definition 2.2.1, if and only if o_k is strictly combinationally*

Algorithm 3 Behavior of latency-insensitive transform A' during a target cycle

1: **while** there exists $o'_k, k \in \{1, 2, \ldots, K\}$ such that

- o'_k is not full;
- in A, if o_k is strictly combinationally connected to $i_{k_1}, i_{k_2}, \ldots, i_{k_{R_k}}$, then $i'_{k_1}, i'_{k_2}, \ldots, i'_{k_{R_k}}$ are not empty; and
- o'_k has not been enqueued during this target cycle.

do

2: Calculate

$$vo'_k = \lambda_k(vs', [vi'_{k_1}, vi'_{k_2}, \ldots, vi'_{k_{R_k}}]) \tag{3.3}$$

3: Enqueue vo'_k in output buffer o'_k.

4: **end while**

5: Calculate

$$ns' = \delta(vs', [vi'_1, vi'_2, \ldots, vi'_J]) \tag{3.4}$$

Update state s' with ns'.

6: Dequeue all the input buffers.

connected to i_j in A.

Proof. According to Equation 3.3 in Algorithm 3, the n^{th} value enqueued into the output buffer o'_k can not change if none of the n^{th} values dequeued from $i'_{k_1}, i'_{k_2}, \ldots, i'_{k_{R_k}}$ changes, provided the rest of the values are the same. Thus o'_k can depend only on $i'_{k_1}, i'_{k_2}, \ldots, i'_{k_{R_k}}$ according to dependency definition 2.2.1. These input buffers exactly correspond to the inputs $i_{k_1}, i_{k_2}, \ldots, i_{k_{R_k}}$ of A which o_k is strictly combinationally connected to. In Equation 3.1, since o_k is strictly combinationally connected to $i_{k_1}, i_{k_2}, \ldots, i_{k_{R_k}}$, there exists some clock cycle for every input in that set for which the value returned by the function $\lambda_k()$ changes if that input changes, and every other input remains the same for all clock cycles. Since the same function is used in Equation 3.3, by dependency definition 2.2.1, o'_k depends on all of $i'_{k_1}, i'_{k_2}, \ldots, i'_{k_{R_k}}$. □

The while-loop in Algorithm 3 can be executed in parallel for all the output buffers whose conditions are met.

In the following, we prove that a latency-insensitive transform A' of an FSM A is a composable latency-insensitive refinement of the FSM.

Theorem 3.1.1. *A latency-insensitive transform of an FSM is cycle-accurate with respect to the FSM*

Proof. During the simulation of a target cycle by A', all output buffers are enqueued once, the state is updated and all the input buffers are dequeued once. Thus, $\forall n \geq 1$, if the n^{th} value is dequeued from every input buffer, then the n^{th} value is enqueued into every output buffer, and the state is updated for the n^{th} time.

$\forall t \geq 0$, let $i'_j(t), j \in \{1, 2, \ldots, J\}$ represent the t^{th} value dequeued from input buffer i'_j and $o'_k(t), k \in \{1, 2, \ldots, K\}$ represent the t^{th} value enqueued into output buffer o'_k. Let $s'(t), t \geq 1$ represent the value of s' immediately before the t^{th} update of s', i.e. the value between the $(t-1)^{\text{th}}$ and t^{th} update of s'. From Equation 3.3 and Equation 3.4 of Algorithm 3, we get

$$\begin{aligned} \forall t &\geq 0, \\ o'_k(t) &= \lambda_k(s'(t), [i'_{k_1}(t), i'_{k_2}(t), \ldots, i'_{k_{R_k}}(t)]) \qquad (3.5) \\ s'(t+1) &= \delta(s'(t), [i'_1(t), i'_2(t), \ldots, i'_J(t)]) \qquad (3.6) \end{aligned}$$

Given that the values in every input of A is the same as the values enqueued into every input buffer of A', we have

$$\begin{aligned} &\forall t \geq 0, \\ &\forall j \in \{1, 2, \ldots, J\}, \\ &\qquad i'_j(t) = i_j(t) \end{aligned} \qquad (3.7)$$

From equations 3.2, 3.6 and 3.7, we get (by induction)

$$\begin{aligned} &\forall t \geq 0, \\ &\qquad s'(t) = s(t) \end{aligned} \qquad (3.8)$$

From equations 3.1, 3.5, 3.7 and 3.8, we get

$$\begin{aligned} &\forall t \geq 0, \\ &\forall k \in \{1, 2, \ldots, K\}, \\ &o'_k(t) = o_k(t) \end{aligned} \tag{3.9}$$

Thus the t^{th} value enqueued into each output buffer of A' is the same as the value of the corresponding output of A during clock cycle t, whenever the t^{th} value dequeued from each input buffer of A' matches with the value of each input of A during clock cycle t. This satisfies the cycle-accuracy condition of Section 2.1. □

Lemma 3.1.1.1. *In a latency-insensitive transform of an FSM, $\forall n \geq 0$, if every output buffer of the transform has been enqueued n times, and if every input buffer is enqueued n times, then eventually every input buffer will be dequeued n times.*

Proof. From Algorithm 3, Step 6, every input buffer will be dequeued after every output buffer has been enqueued. This proves the lemma. □

Lemma 3.1.1.2. *In a latency-insensitive transform of an FSM, $\forall n \geq 0$, if every input buffer of the transform has been enqueued n times and every output buffer has been enqueued n times, now if all input buffers that an output buffer depends on is enqueued once more, and the output buffer is not full, then the output buffer will eventually be enqueued once more.*

Proof. Notice first that, according to Lemma 3.1.0.6, the while loop of Algorithm 3, waits only for all the input buffers that an output buffer depends on to become non-empty; it does not wait for any more input buffers to be non-empty. From Lemma 3.1.1.1, we get that every input buffer will eventually be dequeued n times. Thus if input buffers on which an output buffer depends on has been enqueued once more, the while-loop in Algorithm 3 guarantees that the output is enqueued. □

Theorem 3.1.2. *A latency-insensitive transform of an FSM is a composable latency-insensitive refinement of the FSM.*

Proof. Theorem 3.1.1 and lemmas 3.1.1.1 and 3.1.1.2 prove the statement. □

3.2 Refining the latency-insensitive transform

3.2.1 Multi-cycle combinational functions

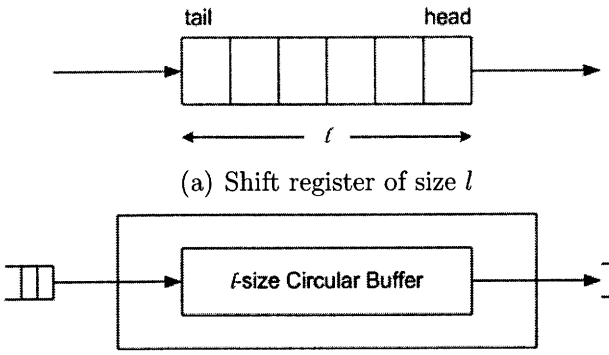
In Algorithm 3 for the latency-insensitive transform of an FSM, the value to enqueue into an output buffer is given by a combinational function (Equation 3.3); and the new value of the state of the transform is given by another combinational function (Equation 3.4). These combinational functions can be implemented to take several wall-clock cycles in order to obtain the result. Once the results are obtained, they are either enqueued into an output buffer, or used to update the state of the transform, as the case may be. The resulting system will still be a composable latency-insensitive refinement of the FSM; the proofs of theorems 3.1.1 and lemmas 3.1.1.1 and 3.1.1.2 do not change.

3.2.2 Dont-care optimization

The state of the FSM and the values of some of the inputs may be such that an output value can be calculated without using other inputs which are strictly combinationally connected to that output. That is, the inputs essentially become a *dont-care*. This can be taken advantage of to improve the performance of the naive latency-insensitive transform described in Section 3.1 as follows: Every input buffer is associated with a counter. Whenever an input is a dont-care in the FSM (for that cycle), and the corresponding input buffer in the transform is empty, then the counter can be incremented. When the input buffer is finally enqueued,if the counter is non-zero, the value can be dropped and the counter decremented. The values enqueued into all the output buffers remain the same as the naive transform, so this optimization is still cycle-accurate with respect to the original FSM. Dependency condition 2.3.1 is also satisfied, the proof being the same as Lemma 3.1.1.2. Cleanup condition 2.3.2 is satisfied, the proof for which is as follows.

Lemma 3.2.0.1. *Dont-care optimization does not violate Cleanup condition 2.3.2.*

Proof. If every output buffer has been enqueued n times, if every input buffer has



(a) Shift register of size l

(b) Optimized latency-insensitive refinement of the shift register

Figure 3-2: An example of an optimized latency-insensitive refinement

been enqueued n times, then by Algorithm 3, every input buffer will be dequeued n times. If instead, every output buffer has been enqueued n times, but some input buffer has been enqueued $k < n$ times, then the counter for that input buffer would be $n - k$, and hence it will drop $n - k$ values enqueued into the buffer. Thus, every input would be dequeued n times, if every input is enqueued n times and every output is enqueued n times. □

3.3 An example of an optimized latency-insensitive refinement

In typical hardware designs, modules communicate via pipelined links, characterized by the latency l. These links can be abstracted as a l-size shift register A with all the registers initialized as shown in Figure 3-2(a). Notice that the output is not combinationally connected to the input. The output is combinationally connected only to the head of the shift register. This property can be used to design the latency-insensitive refinement of the shift register.

Consider a system A' shown in Figure 3-2(b). It contains a circular buffer of size l. The circular buffer is initially full, and every entry is initialized with the value in the corresponding register of shift register A. The output buffer of the refinement is enqueued with the value removed from the circular buffer whenever the circular buffer

is not empty and the output buffer is not full. Note that according to the definition of dependency in Definition 2.2.1, the output buffer is not dependent on the input buffer. The input buffer is dequeued and inserted into the circular buffer whenever the circular buffer is not full and the input buffer is not empty.

Theorem 3.3.1. *A' is a latency-insensitive refinement of A.*

Proof.

Lemma 3.3.1.1. *A' is cycle-accurate with respect to A.*

Proof. The value enqueued in the output buffer is always equal to the value in the head of the circular buffer. Since the circular buffer of A' and the shift register A are initialized with the same values, if the same values are inserted into the circular buffer from the input buffer, as are inserted into the shift register, then the values in the head of the circular buffer and the shift register match. This proves the lemma. □

Lemma 3.3.1.2. *A' obeys Dependency condition 2.3.1.*

Proof. The output buffer is always enqueued as long as there is an element in the circular buffer. If k values are enqueued into the output buffer, then k values can be inserted into the circular buffer. So when k values are enqueued into the input buffer, they will be inserted into the circular buffer. The circular buffer will not be empty after k values are enqueued into the output buffer as it had l elements initially. So, one more value can be inserted into the output buffer. This proves the lemma. □

Lemma 3.3.1.3. *A' obeys Cleanup condition 2.3.2.*

Proof. If the output buffer is enqueued k times, then the circular buffer can be inserted k times. So, if k values are enqueued into the input buffer, they will all be dequeued and inserted into the circular buffer. This proves the lemma. □

Lemmas 3.3.1, 3.3.1.2 and 3.3.1.3 prove the theorem. □

Chapter 4

Related work and Conclusion

4.1 Related Work

The work done as part of this thesis is an extension of the Theory of latency-insensitive design [3], [5], [2], [4]. [3], [5], [4] and [2] give a methodology to automatically wrap modules in order to make them latency-insensitive. Their work focuses on solving the problem of wire delay between these modules. By wrapping the modules and making them latency-insensitive, any number of buffers can be inserted between the modules, thus mitigating wire delays. Each module in the system has a stringent structural restriction that all of its outputs should be registered, in order to be able to create the wrapper. Our work extends this in the following ways. First of all, we give an automatic transformation to convert any module into a latency-insensitive refinement. This transformation is very similar to the transformation discussed in the previous work, but it does not place any structural restriction on the modules – the outputs of the modules need not be registered, they can be combinationally connected to the inputs. Removing this restriction enables us to partition the system arbitrarily into modules, which is important from a design perspective. Secondly, we give the sufficient conditions that the refinements must obey in order for it to be composable. This gives the flexibility to create a refinement obeying these conditions from scratch, or to optimize an existing refinement, without losing the compositionality properties. For example, the designer can add buffers in a pure combinational module, and re-

timing the buffers will pipeline the module without affecting the functionality, thus improving performance.

The work on distributed simulation [7], [6] is similar to our work. A distributed system contains several communicating processes. The only means of communication between the processes is via messages. A process can take a finite time (delay) to produce its outputs. [7] proposes a scheme to simulate this distributed system of processes without deadlocks. Their scheme involves sending time-stamps along with every message, and sending null-messages through an output line, in case of no activity in that output line for the current set of inputs. Our solution is very similar to theirs when applied to the special case of simulating a distributed FSM. Each module of the FSM can be thought of as a process in [7], and the outputs are produced with 0-delay. This avoids the need for time-stamps in our latency-insensitive refinement. Also every output of the module is produced every cycle. This avoids the need for null-messages in our work.

Lately, there has been a surge of interest in the academic community in using FPGAs for micro-architectural simulation. This work was inspired by one such project, HASim [10], [9]. In this project, FPGA simulators for an in-order and an out-of-order pipeline for the Alpha ISA were developed. The modules of the simulator are connected via A-Ports [10], which model the communication latency between modules. An A-Port modeling an l-latency link, $l \geq 0$ is initialized with l values. The modules of the simulator have the following behavior: all the input A-Ports are read first and then, all the output A-Ports are written. This restricts the composability of the modules. For example, if a module has an input 0-latency A-Port and an output 0-latency A-Port, and in the overall design, if the two A-Ports are combinationally connected to each other, then the system will deadlock. The focus of our work is to enforce conditions on the behavior of the modules for composability. We state and prove the conditions that a module must obey in order for it to be composable, thus giving a theoretical foundation for FPGA-simulators to be correct and deadlock-free. The A-Ports themselves are exactly the same as the latency-insensitive refinements of the l-latency links which we discuss in the thesis. Other projects involved in creating

FPGA simulator infrastructure include FAST [8] and RAMP [1]. The modules of the simulators communicate using a model of a FIFO (Connectors in FAST and Channels in RAMP). These projects do not enforce restriction on the behavior of the modules, thus again restricting the composability of the modules.

4.2 Conclusion

In this thesis, we have formally defined *latency-insensitive refinements* and *composable latency-insensitive refinements*. We have stated and proved the conditions that a latency-insensitive refinement should satisfy in order for it to be composable. We also described an automatic procedure to convert any design into a latency-insensitive refinement, and the further optimizations that can be performed on it.

This technique enables us to easily write fine-grained parallel simulators for synchronous designs which can be implemented on FPGAs. It essentially allows the simulator to trade time for space, i.e., take several clock cycles but use lesser resources, while being cycle-accurate with respect to the original design. In the future, we plan to implement an FPGA simulator for multi-core, multi-threaded PowerPC processor along with the associated memory subsystem using this technique.

Bibliography

[1] Arvind, Krste Asanovic, Derek Chiou, Joel Emer, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, David Patterson, Jose Renau, and John Wawrzynek. RAMP: Research Accelerator for Multiple Processors - A Community Vision for a Shared Experimental Parallel HW/SW Platform. Technical report, University of California Berkeley.

[2] L.P. Carloni, K.L. McMillan, A. Saldanha, and A.L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*, pages 309–315, 1999.

[3] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(9):1059–1076, Sep 2001.

[4] L.P. Carloni and A.L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. *Design Automation Conference, 2000. Proceedings 2000. 37th*, pages 361–367, 2000.

[5] Luca Carloni, Ken McMillan, and Alberto Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and LNCS 1633 D. Peled, editors, *Proc. of the 11th Intl. Conf. on Computer-Aided Verification (CAV)*, page 12. UC Berkeley, Cadence Design Laboratories, July 1999.

[6] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.

[7] K.M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440–452, Sept. 1979.

[8] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil Patil, William H. Reinhart, D. Eric Johnson, and Zheng Xu. The FAST methodology for high-speed SoC/computer simulation. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 295–302, Piscataway, NJ, USA, 2007. IEEE Press.

[9] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs. *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 1–10, April 2008.

[10] Michael Pellauer, Muralidaran Vijayaraghavan, Michael Adler, Arvind, and Joel Emer. A-Ports: an efficient abstraction for cycle-accurate performance models on FPGAs. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 87–96, New York, NY, USA, 2008. ACM.