Hardware-based Temporal Logic Checkers for the Debugging of Digital Integrated Circuits

Jean-Samuel Chenard

Department of Electrical & Computer Engineering
McGill University
Montréal, Canada

October 2011

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Acknowledgements

Graduate studies for me were more than just a career change. The ability to take the time to reflect upon a problem and begin to see what others have done before me required time to acquire. Going from what I could do to what can be done requires a different mindset, but offers greater rewards and more elaborate opportunities. My industrial experience provided me with good practical skills before I decided to pursue graduate studies. My academic experience showed me countless ways to approach a problem and gave me appreciation for how much has been done before and how many very talented people have solved so many problems. It also showed me that by taking the risk to try fundamentally different approaches, even if they appeared to be very difficult, turned out to be the most profitable experiences.

I would like to express my gratitude towards my supervisor Professor Zeljko Zilic for his guidance, trust and patience through all the years I spent in the Integrated Microsystems Laboratory, first towards the completion of my Masters of Engineering and now for this doctoral thesis. His constant support and understanding made a great difference and by far exceeded all my expectations of what a supervisor can provide to his students.

I want to highlight the help of my good friend and colleague Stephan Bourduas for our collaboration on Network-on-Chip development, our many co-authored publications, and more recently for his insight on the verification methodology used at his work for the largest microprocessor company in the world. Marc Boule was also a key player in the work presented. His inspiring work on the MBAC hardware assertion compiler provided the foundation for many of the ideas pre-

sented in this thesis. Through our many co-authored publications I was able to appreciate his astute mathematical abilities and enjoyed discussing and debating with him about our proposed debug methodologies.

Thanks to my colleagues Bojan Mihajlovic, Nathaniel Azuelos, Jason Tong and Mohammadhossein Neishabouri for the help and feedback they have provided on this work and thanks to Amanda Greenman for her editorial assistance.

I wish to express my sincere thanks to my funding sources, notably NSERC and McGill. Without those, I don't believe it would have been financially possible to support these long studies. I am also very grateful towards CMC Microsystems for providing such high quality hardware tools, workstations and technical support over the years.

I wish to recognize the work of the many very talented open-source programmers who have helped realize the vision of the GNU/Linux operating system. I used this system throughout my studies on my workstations. I used the GNU/Linux resources as tools, reference material and as an experimental platform on many levels. I even used it to run my online business. It provided me a low-cost solution for selling electronic boards, contributing to paying the expenses associated with long-term studies. Following the open source philosophy was one of the best technical decisions I made. I have made a few contributions to this community and hope to make many more in the future.

A special thank to Edouard Dufresne, who, when I was only a kid, showed me the basics of Ohm's law, antenna design and electronic systems and provided me with my first paid job. His hope was that some day, I would do well in science and engineering. Hopefully, this work can demonstrate that I certainly did progress in that path.

I wish to thank my parents for their never-ending faith in my abilities and their approach to my education. Their unconventional approach to life and how to make your own way without worrying too much about what others think played a key role in the way I do my work, each and every day.

Finally, I wish to thank the love of my life and my dear wife Hsin Yun. She gave me the inspiration and support to ensure that this work came to an end. I am forever grateful.

I wish to dedicate this thesis to my two daughters: Eliane and Livia. May you realize that if you follow your passion and put in a lot of hard work, you can accomplish pretty much anything that you wish.

Abstract

Integrated circuit complexity is ever increasing and the debug process of modern devices pose important technical challenges and cause delays in production. A comprehensive Design-for-Debug methodology is therefore rapidly becoming a necessity.

This thesis presents a comprehensive system-level approach to debugging based on in-silicon hardware checkers. The proposed approach leverages existing assertion-based verification libraries by translating useful temporal logic statements into efficient hardware circuits. Those checker circuits are then integrated in the device as part of the memory map, so they can provide on-line monitoring and debug assistance in addition to accelerating the integration of performance monitoring counters. The thesis presents a set of enhancements to the translation process from temporal language to hardware targeted such that an eventual debug process is made more efficient. Automating the integration of the checker's output and control structures is covered along with a practical method that allow transparent access to the resulting registers within a modern (Linux) operating system. Finally, a method of integration of the hardware checkers in future Network-on-Chip systems is proposed. The use of a quality metric encompassing test, monitoring and debug considerations is defined along with the necessary tool flow required to support the process.

Abrégé

La complexité des circuits intégrés augmente sans cesse et à un tel point que le procéssus de déboggage pose de nombreux problèmes techniques et engendre des retards dans la production. Une approche d'ensemble de conception pour le déboggage (Design-for-Debug) devient donc rapidement une nécessité.

Cette thèse propose une approche détaillée de niveau système, intégrant des circuits de surveillance sur puce. L'approche proposée s'appuie sur la réutilisation de déclarations écrites en language de logique temporelle afin de les transformer en circuits digitaux efficaces. Ces derniers seront intégrés à la puce à travers son interface d'image mémoire afin qu'ils puissent servir au processus de déboggage ainsi qu'à une utilisation dans le système lorsque la puce est intégrée dans son environement. Cette thèse présente une série d'ajout au procéssus de transformation d'instructions de logique temporelle de manière à faciliter le procéssus de déboggage. Une méthode qui automatise l'intégration des sorties et du contrôle des circuits de surveillance est présentée ainsi que la manière dont une utilisation de ces circuits peut être accomplie dans le contexte d'un système d'exploitation moderne (Linux). Finalement, une méthode globale d'intégration des circuits de vérification dans le contexte de systèmes basés sur les réseaux-sur-puce est présentée, accompagnée de la chaine d'outils requise pour supporter ce nouveau processus de conception. Cette méthode propose l'utilisation de facteurs de qualité de test, de surveillance et de déboggage (Test, Monitoring and Debug) permettant une meilleure sélection des circuits ainsi qu'une intégration plus efficace au niveau des resources matérielles.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Ask any hardware engineer how they go about creating digital circuit designs and they will typically explain that based on a set of specifications, they write code that describes the logic of the circuit, or draw components that represent the structure of the design. Likely, they will be re-using pre-existing blocks and connect them together to make a major part of the system, thus rapidly and efficiently converging upon a final product.

After a thorough verification process, *Electronic Design Automation* (EDA) tools will help them transform their high-level description of the circuit and logic blocks into data structures that represent the primitive electronic gates. Those gates are then transformed into transistor circuits and finally into the geometric patterns that represent the layers' masks. Those are sent to a factory for fabrication. The device is powered up, works well and sells in large volumes.

This is the story that everyone in the integrated circuit world likes to hear.

A dark cloud usually floats above this pretty scenario, one that will never really go away: a bug lurking somewhere in the circuit. The incorrect implementation of a specification can throw an otherwise smoothly-running circuit into a behavior that one did not predict or validate. It could be a bug that stays invisible to the operation of the device and appears late in the product cycle, putting the entire company at risk. Even worse than the bug that one can see and examine is the one that seems to appear at random intervals, one that emerges and vanishes so

quickly that only a slight trace of data destruction remains in its path... too little, too late to help investigate.

It is that lack of visibility and the difficulty of tracing erroneous behavior in a silicon circuit that motivates this research. Our primary objective is to propose a method by which one can leave little circuits in the final device that act as small collectors of evidence. Evidence that one hopes will never be used in the final device, but if ever needed, would cut out weeks or months of forensic search to locate and remove the nastiest of bugs. In the quest to manage complexity, productivity and provide systems that will be bug-free, we propose a set of guiding principles, a design-for-debug methodology and the design tools to assist in the debug of future complex systems.

1.1 Semiconductor Manufacturing Process

One cannot really grasp the complexity of modern silicon devices without an overview of the manufacturing process and its implications on the final product's complexity.

From the conceptual design to the final circuit in the silicon, an impressive array of technological elements are involved. Highly accurate robots (controlled by computers) in an assembly line of impressive accuracy and repeatability, dope, etch, protect and polish a pure silicon wafer. Each step is carefully monitored. The silicon wafer evolves into a product whose worth will, by weight, surpass most of what can be produced by man. This wafer, containing hundreds of replicas of a miniature circuit, each containing up to a billion transistors is then separated and tested. The conceptual circuit is now a real object constrained by the laws of physics. Each individual circuit will undergo millions of test cycles to ensure that it meets specifications.

Each step in this amazing process relies on models, algorithms and empirical measurements that together have to converge to a working device. The end result is the production of an electronic device that, even for the most experienced, never ceases to amaze with its performance and integration.

So what makes the fabrication of modern, large-scale, integrated circuits possible?

A: Fast computers and massive amounts of advanced software.

How can those computers provide so much computing power to run this advanced software?

A: They use modern, large-scale, integrated circuits...

The idea that a machine could be programmed to calculate dates back to the 1800s, but it was only around 1950 when Turing-complete machines started to be used for generic computing. The use of the electronic transistor made the creation of much smaller and more power efficient circuits possible. In the 1970s the first commercial microprocessors came on the market. From then on, each new iteration of microprocessor design added complexity, but made each generation faster and more power efficient. Each computer generation assisted in the design and verification of their future replacements. Few industries can accelerate their own growth with the very products that they make. Some are now pondering how far this progress can continue and where this will lead us as a species [3].

Setting aside the philosophical question of human destiny and its links with computers, the fact remains that modern designs entirely depend on a massive number of computers in all steps of the design, verification and manufacturing process. From the business financial calculations to the individual layers of atoms deposited on the wafers, not a single step evades the computer program. It simply cannot be avoided since only the computer can handle the massive amount of data required to model and simulate the steps of such complex designs.

From the advances in the lithography equipment [4] and semiconductor processes to the improvement in EDA tools [5], each improvement in the design chain contributes to maintaining an impressive rate of progress known in the industry as Moore's Law [6]. The use of *Intellectual Property* (IP) blocks and computing cores keep the engineering productivity high enough to utilize the newly available logic resources available in each new generation of *Field Programmable Gate Array* (FPGA) and *application specific integrated circuit* (ASIC) processes.

Today's high logic integration density and advanced semiconductor processes allow ever more complex designs to be attempted, requiring tremendous engineering resources and capital expenses. Those designs also involve a significant amount of business risk, but the return on investment of a successful product is so substantial that many companies are willing to invest fortunes for the poten-

tial payback that a well designed product can bring to their shareholders. With each increase in complexity, new tools and methodologies must be devised to assist with the engineering of those newer systems. Unlike the computers that run the tools, engineering resources do not scale exponentially. As future devices will clearly not *lack* the technological means to support more logic resources, one has to find a way to better use the more limited engineering resources.

This thesis proposes to leverage a verification process called *assertion-based verification* that recently started to be successfully used in complex designs and brings many of its benefits all the way to the final silicon devices. This new verification methodology was found to be very efficient [7] at finding root causes of bugs. As logic bugs in silicon are ever more difficult to detect, analyze and eliminate, a methodology improvement in this area can make a big impact on the industry.

As this thesis will explain, some of the *formal* properties of a design described by *sequences* and *assertions* can be transformed into efficient hardware circuits that can be used to gather evidence of circuit malfunction. This thesis then proposes a few mechanisms to record and present the evidence such that the debugging process can rapidly converge to the source of the problem and how to integrate this information as part of a complete solution. This *design-for-debug* strategy is presented from the perspective of a set of *properties* applicable to a debuggable system and is tightly coupled with the operating system and firmware.

The use of in-silicon assertion checkers is studied in the context of future large-scale digital systems such as Network-on-Chips. The integration of checkers such that their output can be monitored and processed by advanced software libraries and algorithms is covered and methods are presented to integrate the checkers in a modern operating system.

1.2 Debugging Process

A moth found trapped between two contact points in an early relay-based computer in 1945 caused it to malfunction [1]. It became known as the first recorded computer "bug" (at least in the physical sense). However, the term *bug* in the con-

1. http://www.history.navy.mil/photos/images/h96000/h96566kc.htm

text of computer engineering sense had been used for some time.

The terms *bug* and *debugging* have become entrenched in all steps of building complex systems. For each new generation of computers designed, many bugs are discovered and resolved. Most of those bugs will end up recorded in log books and may haunt those who have to spend sleepless nights tracking them down. Some serious bugs have even "escaped" the scrutinous verification process such as the Pentium co-processor division problem experienced by Intel [2]. Such publicised bugs become famous mainly due to the financial impact they have on the company handling the recall of a flawed *Integrated Circuit* (IC). Those examples serve to show how many variables and conditions must be considered when making a large and complex system that one wants to be *bug free*. Those bugs that the public learn about in newspapers only represent the few that "made it out". Numerous high-profile projects are delayed by integration bugs, respins of large ASIC devices. Countless engineering man-hours are spent tracking complex and nasty integration bugs. Each one has the potential to cause massive loss of sales and delays in product delivery.

With designs currently exceeding one billion transistors and still predicted to increase in density and size for many years, one can clearly see that the verification and debugging of those extremely large circuits pose a significant challenge. Interestingly, verification is actually the most time and resource consuming part of a large digital design project. Debugging has always been challenging from the onset of complexity. It requires an in-depth understanding of the circuit, a mental model of the interaction between its parts and a fair amount of control and visibility to be efficient. In large systems, debugging is the part of the verification effort that consumes the most time. The ever increasing density of designs, coupled with the large amount of external IP involved in their conception requires a change in focus when tackling the debugging of complex integrated circuits.

Design methodologies cannot afford to simply react to problems once the design hits the proverbial laboratory bench, but must take a proactive approach to facilitate the diagnosis and location of problems by planning the upcoming debug phases early in the design process.

2. http://www.intel.com/support/processors/pentium/sb/CS-013007.htm

1.3 Debugging of future digital systems

The debug process can be seen from many perspectives. It spans a continuum from circuit-level hardware to the higher order application-level code execution. Future generations of devices will have complex, heterogeneous structures and the debugging process will have to consider real-time requirements that need to be met on top of functional considerations.

To understand the above statement, take, for example baseband processing in a portable wireless device such as a modern "smart" phone. Only a few years ago, the radio frequency part was provided as a complete, dedicated circuit that processed the radio signal and decoded it down to the packet-level digital communication. The baseband processing required many different ICs (analog and digital) to perform the task of recovering data from the radio signal. Modern solutions integrate all of those ICs into a single die. Many analog functions are now performed in the digital domain, increasing flexibility and reducing the need for expensive, accurately tuned analog components. The complex process of turning the radio signal into data packets has now turned into a parallel computing problem subjected to *hard real-time* requirements. By re-programming some software and firmware elements, the same hardware can now "tune in" to other frequencies like the global positioning system. This can transform the initial telephone into a navigation device. As the *Central Processing Unit* (CPU) incorporates more cores, more tasks that were once hardware devices will become software libraries and the electrical signals that used to carry information between devices on a board will be replaced by messages exchanged among the CPU cores.

This has profound implications on the debugging. Those future devices will have to perform parallel calculations within stringent time limits. The computations will have to be performed in a distributed system that operates like a small network of nodes, but one that offers practically no visibility of its internal activity on external pins. This transition from *system-on-chip* (SoC), where the various cores on a chip are dedicated to a given task, to a *Network-on-Chip* (NoC) where the cores are more general purpose and the software and routing strategy make it application specific, will thus require a sophisticated debugging infrastructure. NoC

solutions that aim to offer a flexible platform allowing designers to quickly deliver a range of working products will only reach their full potential if debugging is carefully considered at the core of the design process.

1.4 A Systematic Approach to Design for Debugging

An overview of computing trends shows that the complexity and design size tends to increase with time, no matter which computing paradigm one wishes to follow. What was previously considered a complex project taking many man-years to complete, for example a CPU core, can now be integrated on a SoC in a matter of hours by a design tool. The initial complexity of the re-used IP block remains, only hidden away by the level of abstraction that is gained from its re-use. When things go wrong as a result of a bug (in the core or in its integration), the complexity of the problem reappears compounded by the lack of a full understanding of each of the parts that are integrated in the design. Regardless of who is responsible for the bug: the IP vendor, the system integrator or an EDA tool, the problem has to be found, fixed and tested before the device can be released.

This puts a lot of pressure on engineering teams. A lot of time will be spent learning about the intricate details of the IP blocks used and trying to come up with scenarios to re-produce the failure in a controlled manner. Usually, those failures would not have showed up in simulation (otherwise the design would not have been released). Somewhere in the circuit, an erroneous condition exists, but only its end effect can be observed.

This is where the work presented in this thesis will attempt to assist. The main goal is to have silicon devices that not only perform their intended function well enough to please the customer, but also include hardware "intelligence" that can *assist the localization of the root-cause of bugs*, should they crop up during the latter phases of product design. Coupled with a database of formalized and structured information about the device's inner-workings, the powerful computing capabilities of the hardware will come to assist the debugging phases.

1.5 Properties of Debuggable Systems

The role of the debug engineer, when in charge of a large and complex project, is put in perspective by veterans of the semiconductor industry in the following quote:

> *"Such is the nature of silicon debug. To be successful, the debug engineer must be able to solve problems in areas where he has no technical expertise, drive design teams to make changes where he has no influence, and be able to predict the future." (Doug Josephson – Hewlet Packard ; Bob Gottlieb – Intel)* [8]

Even towards the end of the 1990s, engineers at the Philips Research Laboratories were aware that scan chain (a mechanism to serially shift bits in and out of the device registers via a bypass of the usual logic function) would not be enough to assist in the debugging of a large-scale, multiple clock domains IC [9]. In order to aim for the best debugging process possible, one can consider a series of properties that can augment the *debuggability* of a given system while easing the burden on debug engineers and design teams. As those properties are enumerated, the relevant sections of this thesis are highlighted.

1. **Increased Visibility**. One needs an increased visibility in the design, ideally as it is running and in a dynamic manner. The ability to "peek" at internal device states and monitor the various elements that affect the outcome will have a great effect on the efficiency of the debug process, since it will help build an understanding of the data flow. Often, in silicon ICs, one can relatively easily observe the inputs and outputs of the device (through the I/O pins). However, the internal data processing flow is a lot more difficult to observe, especially in real-time. In some cases, a combination of multiplexers and control circuits will allow a *snapshot* of the device state to be observed. This *scan-based* method is quite useful, but requires the complete operations of the device (or a significant portion of it) to be stopped while all the bits are shifted out (usually serially) from the device. *Shadow scan* registers can allow the system to continue its execution while a "snapshot" of its state is shifted out, but cannot accumulate more than one copy of the

running state. Someone debugging a large multi-core system or a NoC will want a more flexible solution. This thesis proposes a mechanism for the integration of *sequence checkers* and *assertion checkers* such that significant events are recorded and can be propagated within the system. They could then be automatically aggregated and stored in a larger memory as a trace of the detected failure. This allows better capture and better dynamic understanding of the operation. Chapter 5 details an approach to centralize the capture and traces through the re-use of the NoC transport mechanism. In current design tool flows, the visibility of an internal operation is very good in the simulation environment, but very poor in the silicon. Conversely, the speed of execution on the simulator is very low, but blazingly fast on the silicon. This thesis proposes a method by which key elements in the hardware execution, monitored at runtime by hardware checkers presented in Chapter 3 can be recorded in firmware-visible hardware registers (whose generation is covered in Chapter 4) to assist in re-creating a problem detected on-chip in a simulation environment to facilitate the bug localization process.

2. **Increased Controllability**. One needs the ability to control multiple heterogeneous flows of control. The device must allow the person debugging it to manipulate and alter the internal states in a way that can induce a predictable response from the system. This manipulation of internal states needs some hardware and firmware assistance such that one does not destroy the working state of the device under debug. Using a *scan-based* approach, the designer would be able to stop the design and modify a few bits before continuing. This method is well established as a way to insert specific *test* patterns inside a circuit to validate its operation (chip testing), but for system-level debugging, it falls short of providing an efficient and dynamic solution. In an ideal situation, it would be possible from within the system (i.e. not using scan) to force a device into a failure mode that has a similar signature to the system being debugged. Thus, by comparing symptoms from the buggy device and the manipulated version, one can aim at repeating rare bugs frequently. This is an important debugging rule [10]: be able to *repeatedly* reproduce a problem. At that point, the debug process can efficiently resolve

the issue and *confirm* that the bug has indeed been fixed completely. The work in this thesis addresses the concern that scan-injected *debug sequences or state modifications* (aimed at locating a bug) do not cause the device's internal circuits to go into states that would violate internal protocol requirements. Those violations would be flagged by the hardware checkers described in Chapter 3 and would indicate that the debugging strategy is flawed. The debug engineer could then modify his approach.

3. **Diagnostic Assistance**. The system should offer assistance in diagnosing the root cause of a bug. This is quite important when one considers how many registers and memories a future device will be able to host. A complex SoC can internally hold tens of thousands registers and memory addresses (excluding the billions of externally addressable memory locations). A database of registers coupled with firmware assistance and tools must be provided to the person debugging to help him understand the behavior of the circuit and extract meaningful interpretations from the register states. The person debugging a circuit is likely to only partially understand the internal operation and only from a high-level point of view. Only through abstraction and interpretation of the information by design tools will the person debugging be able to fully comprehend the underlying operation of modules and be able to pinpoint the source of an observed problem. Chapter 4 addresses those concerns by allowing system-level libraries within the device to leverage databases, graph manipulation libraries and rich I/O post-processing such that the device can assist with its own debugging process. With the proposed strategy and firmware assistance, the device, rather than simply stating that an error occured and give a bit location report, can perform internal lookup in a local database and report the cause of the assertion failure, the related IP module and the line number in the related formal specification document. In our proposed approach, since the information is now part of the application space of the system, advanced transmission mechanisms (e.g. wireless or wired networking, graphical display) can be leveraged to report the internal condition remotely. This can prove very useful for a future distributed system (sensor network, for example) as integration bugs become

even more difficult to tackle since the system may not be so easily attached to debugging hardware.

4. **Data Volume Reduction**. Efficient handling of exceedingly large amounts of debugging-related data. Dynamic tracing of memory access or instruction execution, especially in fast multi-processor or network-on-chip, hardware-assisted pattern processing is required. A basic example of that is the trigger logic for on-chip analyzers. The high internal bandwidth between on-chip elements can only be observed (traced) if some form of compression and pattern matching is used. Otherwise, the amount of data produced by the internal "tap" will so rapidly overflow the analysis unit that the captures will hold little to no meaning. This thesis proposes the re-use of verification assertion checkers as a way to extract higher-level patterns from the internal operation of the device. Those patterns can be used to trigger the input storage of trace buffers and reduce the acquisition storage requirements. Chapter 3 shows how debug-enhanced checkers can be used to fulfill this need. Furthermore, one can build more complex patterns by using temporal logic advanced semantics. Section 3.3 of explains how temporal multiplexing of checkers can be used to support on-line monitoring, yet reduce the hardware overhead. The same programmable logic structures used in this technique can also support complex hardware-based triggering mechanisms.
5. **Multi-Threaded Support**. Provide support for multi-threaded execution control of relatively fine granularity. This means that as hardware assisted threads of processing progress, one must be able to monitor and control the progress of those threads and be able to trace the blocking, dependencies and inter-thread communication. In a multi-threaded system, execution units operate independently. However, it is important to be able to trace through system transitions in the software execution and qualify a given set of event order, for example dealing with critical section locking and unlocking. By posting these signals as hardware events, assertion checkers can monitor and provide feedback the checker's process back into the operating system and trigger an exception if an event occurs that breaks the temporal specifications. Using the NoC transport mechanisms, one can also centralize the thread exe-

cution events to report system-wide status. Chapter 3 and Chapter 4 provide the foundations for the hardware structures to support this and Chapter 5 proposes an integration methodology for large and distributed systems. As threads are spawned across multiple cores (which in a network-on-chip may not necessarily share the same memory), the debugging process has to be made aware of the thread locations, while abstracting the underlying hardware architecture as much as possible. Although this thesis does not directly address this problem, a few hardware elements proposed in the design for debug infrastructure can be modified to interface with debuggers, providing more flexible breakpoints based on complex internal hardware states and coupled with software data structures. This proposed approach is explained for potential future work in Section 6.2.1.

6. **Multiple Levels of Abstraction**. Able to handle multiple levels of transactions, transparently, if possible. The hardware must be able to allow low-level monitoring of its structure, yet provide a simplified "view" of its transactions for higher order analysis. For example, one could want to see the detail on a bus-level transaction by observing each step on a hardware-based state monitor, but would also want to have only a counter on the full transaction completion for higher-level analysis such as performance review. This can be provided by the hardware checkers presented in Chapter 3. Furthermore, a technique proposed in Section 3.2.5 proposes a method to support highly pipelined circuits where many simultaneous streams of transactions are processed. In those instances, an assertion failure has to be correlated with a given entry in the pipeline which is difficult since, by definition, the pipeline is processing multiple data elements simultaneously.
7. **Operating System Integration**. Integrate well with OS services, outside the kernel space. Applications running on a system must be able to track low-level hardware "blocks" without relying on special CPU instructions or obscure hardware tricks. This will allow the end user (in this case the programmer or system level engineer in charge of debugging) to fine-tune his applications without the need to go beyond the use of an application programming interface. The interpretation of the hardware registers should be

done in user-space to gain access to the processing libraries available. Section 4.6 proposes such a mechanism that was prototyped in a high-end hardware platform using the Linux operating system as a case study.

8. **Remote Control and Visibility** Provide remote debug support by way of specialized hardware interfaces, allowing the complete device to be remotely controlled and with a deterministic way to execute the program cycle-by-cycle the program in its multiple cores. This supports the needs of debugging operating system integration, and low-level hardware problems. Chapter 2 of this thesis covers previous work from the literature that cover this aspect of the debugging problem and show the trends in the standardization of debug for those hardware interfaces.
9. **Support for Simulators and Emulators**. The debugging process must also allow transparent use of simulators and emulators, as well as in-circuit emulation with multiple targets. This debug process has to handle all the steps up to and including the physical design. From the system simulation, to the regression testing on hardware emulators, and finally in prototypes using in-circuit emulators or programmable logic to validate proper system integration. By leveraging assertion-based verification methodologies and carrying their properties at each step of the verification process all the way to silicon implementation where they can be used to correlate back to the simulations, assertion checkers offer a uniform representation of the critical properties. A methodology to select the assertions worthy of integration in the final silicon is proposed in Section 5.5 of Chapter 5 and explores how it can help unify the *test*, *monitoring* and *debug* of future devices.
10. **Measure of Dynamic Performance**. A good hardware debug infrastructure will also facilitate performance evaluation, in addition to plain functional evaluation, and can thus be used to solve critical real-time integration problems. At the same time, the debug infrastructure has to meet realistic cost (silicon area) constraints. Section 5.5.7 aims at optimizing the cost/benefits of including a hardware infrastructure by proposing a set of quality metrics that one can use to perform optimizations.

1.6 Thesis Contributions

The contributions presented in this thesis can be summarized in the following points:

- A set of temporal logic assertion checker transformations that assist in supporting an in-silicon design-for-debug methodology. Through a novel use of time multiplexing of debug-enhanced hardware checkers, sequence completion counters and control points, designers can benefit from a collection of in-silicon checkers and monitors that, by virtue of their closeness to the hardware and their parallel processing capability, can detect and report malfunctions in a timely manner. The hardware circuits can be directly derived from the existing assertion-based verification process, thus limiting the work required for their creation and can be temporally multiplexed to meet area constraints.

- An integration method for hardware-based checkers and monitors in the context of a modern operating system allowing firmware librairies to provide in-field assistance to the bug localization and tracking process. This automated integration method relieve the designers from the burden of integrating a large number of checkers manually and provide assistance in creating the supporting application interfaces to those registers. The proposed operating system integration method preserves fine granular control on memory access permissions through the device nodes to mitigate potential security breaches within the system.

- A methodology to accomodate a large number of assertion checkers and sequence monitors in a distributed system, notably in the context of a NoC. The approach considers the need for status aggregation in a central monitoring point. It also augments the traditional ASIC or large FPGA design flow to incorporate a Design-for-Debug methodology based on hardware checkers derived from assertion libraries and proposes a *quality* metric that can be leveraged to automate the selection of hardware checkers to meet area and

power constraints.

1.7 Self-Citations

The title and description of peer-reviewed publications and technical application notes that cover significant aspects of this thesis are listed below:

- *Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug [11]*: This paper presents techniques that enhance automatically generated hardware assertion checkers to facilitate debugging within an assertion-based verification tool flow. Starting with techniques based on dependency graphs, the algorithms for counting and monitoring the activity of checkers, monitoring assertion completion are presented. The concept of assertion threading is also covered. These debugging enhancements offer increased traceability and observability within assertion checkers, as well as the improved metrics relating to the coverage of assertion checkers. This paper served as the basis for the subsequent journal publication [12]. The contributions of JS Chenard are mainly in bringing the verification and debug perspective to the temporal logic to hardware translation such that the debug process based on assertion can be realized in-silicon like it was possible in a simulator. Those exact contributions are detailed in the third element of this list.
- *Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis [13]*: This paper presents the use of assertion checkers in post-fabrication silicon debugging. Tools that efficiently generate the checkers from assertions for their inclusion in the debug phase are described. The use of a checker generator that can be used as a means of circuit design for certain portions of self test circuits, and more generally the design of monitoring circuits is explained. Efficient subset partitioning of checkers for a dedicated fixed-size reprogrammable logic area is developed for efficient use of dedicated debug hardware. In this publication, the checker generator and associated description along with the redundancy and BIST concept were contributed by M. Boulé and Z. Zilic. The partition algorithm was developed with the co-

authors, the automation of synthesis data extraction (providing the metrics on which the partitioning algorithm relies) was developed and implemented by JS Chenard.

– *Debug enhancements in assertion-checker generation [12]*: A set of techniques for debugging with the assertions in either pre-silicon or post-silicon scenarios are discussed. Assertion threading, activity monitors, assertion and cover counters and completion mode assertions are explained. The common goal of these checker enhancements is to provide better and more diversified ways to achieve visibility within the assertion circuits, which, in turn, lead to more efficient circuit debugging. Experimental results show that such modifications can be done with modest checker hardware overhead. In this work, the debug enhancements of *completion monitoring*, *assertion counters* and *dependency tracing and logging* were brought forth by JS Chenard. JS Chenard also developed the CPU pipeline (derived from the DLX CPU and instruction set from Hennessy and Patterson) and completed the testbenches and sample debug session (through error injection in the pipeline) to produce the example of *hardware assertion threading*. Integration of those debug enhancements in the MBAC tool was performed by M. Boulé. The experimental results were produced by M. Boulé using MBAC while the automated synthesis and data extraction was done by JS Chenard. M. Boulé also provided a comparison of checker's generated area when compared to the FoCs tool by IBM as a way to highlight the performance and density of the assertion checkers. This work was done under the guidance of Z. Zilic.
– *Efficient memory mapping of hardware assertion and sequence checkers for on-line monitoring and debug [14]*: This publication (currently under submission) proposes an on-line monitoring infrastructure to incorporate hardware assertion and sequence checkers in complex CPU-based systems. An efficient heuristic to pack the bitfields is presented along with three different packing modes and their trade-offs, considered from a system-level integration perspective. The main elements of this publication are covered in the first part of Chapter 4. This work was performed by JS Chenard under the supervision of Z. Zilic.
– *A RTL analysis of a hierarchical ring interconnect for network-on-chip multi-pro-*

cessors [15]: The *register transfer level* (RTL) architecture of a hierarchical-ring interconnected network-on-chip is presented along with area and speed measures, favorably comparing this implementation to other NoC implementations in the literature. S. Bourduas provided the initial architecture and models of the hierarchical ring interconnect. JS Chenard's contributions were in architecturing the model to synthesizable RTL such that it can support asynchronous clock domains. The contributions also included many iterations of timing analysis and performance improvements (to reach the 250 MHz target), floorplanning on the Virtex-II FPGA and RTL-Level test bench implementation and data analysis. The integration of the Leon CPU cores was a collaboration between S. Bourduas and JS Chenard under the guidance of Z. Zilic.

– *Hardware Assertion Checkers in On-line Detection of Faults in a Hierarchical-Ring Network-On-Chip [16]*: This paper presents a methodology for using assertions in network-based designs to facilitate debugging and monitoring of system-on-chip. Relying on an internally developed assertion-checker generator to produce efficient RTL-level checkers from high-level temporal assertions, with optional debugging features. Tools to encapsulate the checkers into network-on-chip flits are discussed. The contributions of JS Chenard were related to the hardware architecture of the modified station, the concept of automated register integration and the proposed flow. N. Azuelos contributed the part on assertion timestamping and proposed the hp-flit concept. Implemenation of the hp-flit bypass mechanism was a collaboration between JS Chenard and N. Azuelos. M. Boulé MBAC tool was used to support the translation of the PSL statements to hardware. S. Bourduas provided the architecture of the NoC used in this publication. Z. Zilic provided the supervision and guidance along with material in the background section.
– *A Quality-Driven Design Approach for NoCs [17]*: This article advocates a systematic approach to improve NoC design quality by guiding architectural choices according to the difficulty of verification and test. Early quality metrics are proposed for added test, monitoring, and debug hardware. The concept of Quality metric was put forth by Z. Zilic. The SystemC modeling was mostly done by S. Bourduas from data gathered from the RTL model

provided by JS Chenard. The tracer assertion examples, the ASIC synthesis toolflow, memory cell generation for the TSMC process along with the performance and power extraction process were done by JS Chenard. The calculations to derive the quality scores were done as a collaboration between S. Bourduas and JS Chenard.

– *Canadian Microelectronics Corporation Application Note Series on the* Berkeley Emulation Engine Version 2 *(BEE2) rapid prototyping platform* [18, 19, 20]. This series of 3 application notes cover the details of generating the FPGA hardware, porting Linux 2.6 to the BEE2 and advanced techniques for hosting the user programs on this particular architecture. The first application note titled **Configuring, building and running Linux 2.6 on the BEE2 with the BusyBox user environment** details the steps to create the BEE2 control FPGA along with a customized Linux kernel and the creation of the root file system based on the Busybox [3] project. The second application note titled **Extending the Flexibility of BEE2 by Using U-Boot to Load the Linux Kernel via Ethernet** explains how the BEE2 reprogrammable system can be made fully controllable and re-programmable by hosting only the control FPGA bitstream and U-Boot (a bootloader, similar to a PC BIOS) on the physical system and having the Kernel, root file system remotely attached at boot time. This allows full remote access to the system and more rapid design space exploration. Finally the third application note titled **Using Linux Userspace I/O for Rapid Hardware Driver Development** covers the technical details of exporting the physical hardware registers to the user-space (application) such that the software can transparently access the hardware devices while keeping the system secure and physical memory access constrained to limit the potential of crashing the system if the application code contains bugs.
  All the work performed on the BEE2 system, including the port of the Linux operating system from kernel 2.4 to kernel 2.6, FPGA core integration root system file creation, debug and appnote creation was the work of JS Chenard. However, none of this would have been possible without the work of countless open source developpers around the world. A few indirect contributors

3. http://www.busybox.net/

that should be highlighted are: Dr. Hayden Kwok-Hay So for his work on the BORPH Linux platform (providing a basis for many drivers of the ported BEE2 drivers), Grant Likely (Secret Labs) for the CompactFlash Drivers and GIT tree aimed to support the Xilinx ML-300 platform (a close cousin of the BEE2 architecture), Hans J. Koch (Linuxtronix) and Greg Kroah-Hartman for their work on the UIO driver and assistance with the integration of the proposed UIO PowerPC MMU bugfix in the mainline kernel. Many other open source authors should be highlighted, but for conciseness, only their project are listed: DENX ELDK, DENX Das U-Boot, Crossdev, Busybox, Python, GNU GDB/DDD, kernel.org. Many of the methods used to port software to the BEE2 system derive from studying the Gentoo Linux ebuild structure and documentation provided by the Linux from Scratch project [4] and specialized books [21, 22, 23, 24].

1.7.1 Earlier Work on Debug and Systems

The following peer-reviewed publications cover some of the earlier work by JS Chenard linked to debug technologies, education on debugging methods. Those publications can provide insight and background material for some of the tradeoffs discussed in this thesis.

- *Architectures of Increased Availability Wireless Sensor Network Nodes [25].* This publication covers early work on remote debugging of wireless nodes through JTAG mechanisms, remote bootstrapping and redundancy support to increase availability. The contributions of JS Chenard in this publication relate to the hardware circuits (printed circuit board, firmware) and radio frequency elements (transceiver link and supporting circuitry). M.W. Chiang contributed the architectural study and the digital testing aspects to the publication. Prof. Zilic and Prof. Radecka supervised this publication.
- *Design Methodology for Wireless Nodes with Printed Antennas [26].* This publication details the method to design and build printed antennas for wireless nodes in a way that reduces the risk of having to debug complex interactions between the circuit board and antenna. In this work, the contributions of JS

4. `http://www.linuxfromscratch.org/`.

Chenard include the design and debug of the radio frequency transceiver circuit and radio frequency feed network along with the fabrication and test of the printed circuit board. C.Y. Chu did the 2.5D and 3D modeling of the radio frequency structures and contributed to the design of the loaded dipole antenna. Prof. M. Popovic and Prof. Z. Zilic supervised the work from a radio frequency and circuit perspective, respectively.

– *A Laboratory Setup and Teaching Methodology for Wireless and Mobile Embedded Systems [27].* This Transaction on Education publication summarizes the teaching kit that was designed and deployed by the IML Group at McGill University. It was used during more than five years as the platform for teaching the Microprocessor Systems course to undergraduate students. The design methods presented cover complex systems with a strong emphasis on debugging methods and techniques. It summarizes many years of teaching experience and offer insights on how to design laboratory kits for teaching of digital microcontroller-based systems. The methodology incorporates programmable logic along with the microcontroller such that students can develop and use advanced debug techniques on physical hardware. JS Chenard developed the McGumps teaching kit along with the accessories and supervised the fabrication and deployment of the kit. In this task, he was assisted by colleagues from the IML Laboratory, including Atanu Chattopadhyay Kahn Li Lim and Milos Prokic. Advice on teaching strategies and methods were provided by Genevieve Gauthier. In this publication, M. Prokic contributed the McZub teaching platform material and assisted in the preparation of the manuscript. Prof. Z. Zilic was supervising and teaching the course during most of the semesters and provided his expertise in the teaching methodologies and grading methodologies.

1.8 Thesis Organization

Chapter 2 starts by reviewing computing trends and why they translate into a considerable increase in complexity. The section also covers the terminology, the details of a modern verification process and topics related to semiconductor debug

and hardware units. Many elements from the literature are covered from industry experts and academics. Chapter 3 covers assertion terminology and the process used to convert assertions to hardware checkers. Hardware assertion threading and how this can assist in the debug of pipelined circuits, are also covered along with the partitioning mechanisms and an algorithm allowing time sharing of assertion circuits to reduce area overhead in circuits where many assertion checkers need to monitor a subset of signals. Chapter 4 focuses on connecting those circuits to the memory map of larger systems and provides an automation framework. This chapter also explains how the memory map can be made more transparent and directly accessible from the operating system while preserving memory protection mechanisms and permissions, enabling its use in large, multi-user systems. This chapter also outlines how this was prototyped on a high-end reprogrammable system running the Linux operating system. Chapter 5 discusses the integration of the debug methodology in a large Network on chip and associated problems when one considers hardware assertion and sequence checkers placement. This chapter attempts at offering a quantitative measure of *quality* when one considers the requirements of *test*, *monitoring* and *debug* capabilities.

Chapter 2

Background and Related Work

This chapter will first cover notable trends in debug support for modern digital computing systems. To clarify any ambiguity in the semantics, an overview of the terminology of IC design, verification and test will follow. Next, a modern verification methodology will be covered and the techniques will be put in context such that one can then understand why the verification process is so closely linked with the debugging phases in digital designs. New developments in verification are then presented, notably the use of temporal language and assertion statements in the verification process. Then, the debugging process of silicon devices is covered, highlighting previously accomplished work in that area. Notable trends from the literature in *Design for Debug* (DfD) are highlighted.

Finally, trends in complex SoC and migration to the NoC paradigm are discussed, along with how this will affect the complexity of debugging process and how other researchers have approached the problem.

2.1 Complexity Trends in Digital Systems

2.1.1 The "Simple" Hardware Systems

Simple digital designs, the ubiquitous kind that abound in everyday products from coffee makers to simple industrial controllers can afford to separate

the hardware-centric debugging from the application code and thus can provide a working device which the software can be developed on. The debugging of those simple designs can be approached methodically, but may not require so much effort in defining a verification and debug strategy. Most modern microcontrollers include at minimum a debug infrastructure to control the CPU execution, examine memory content and registers. In debugging those systems, a few captures of the device behavior and a succinct analysis is sufficient to extract enough information from the symptoms of the device to find the solution to its problems. Since the digital logic devices making those systems are thoroughly validated, the bugs that remain are mostly software based and require only a change in the code to be fixed (with the exception of the occasional silicon errata). This device segment offers a point of interest to researchers, who try to offer more dynamic visibility into a device's operation as it is running, instead of requiring the execution to stop (e.g. breakpoints) to examine the content. One has to note that those types of designs represent a large market share and consume the bulk of the semiconductor production. They are usually very cost sensitive, so the debugging support is usually minimal in order to limit the cost of the devices. For interested readers, a good overview of debug standard description (JTAG, breakpoints) used in those devices is provided by B. Vermeulen [28].

As future designs will be required to work on ever more complex data sets, their analysis requires more powerful tools, especially if the data is encoded in a non-trivial way. For example, if blocks of data are encoded to increase their robustness, segmented among multiple transfers or encrypted. A dump of states or signal trace becomes too complex for direct analysis. Some assistance from computer-aided tools, such as protocol analyzers, becomes beneficial.

More complex devices trickle down into the commodity product market at a very fast pace and consumers expect their latest appliances to be "smarter". This means that even low-end products will also see an increase in their internal complexity. For example, USB ports, at one time only available on personal computers are now part of many consumer products from digital cameras to telephones and even picture frames. This trend has to be addressed by providing engineers tools that will facilitate the debugging process and abstract away this new complexity. Nowadays, the lowest-cost microcontrollers that sell for a dollar already in-

clude the necessary logic to allow them to be debugged *in-system*. Recent updates by leading microcontrollers manufacturers now provide dynamic modification of memory content, hardware breakpoints and low-pin count debug interfaces in devices selling well below 2 dollars [1].

Such modern, but "simple" designs are not of prime interest in this research since their complexity is offset by the ability to control and monitor their internal structure. However we can observe that even those "simple" design examples would be considered considerable technological achievements only 10 years ago. Thanks to the basic debugging support built into modern microcontrollers and powerful logic analyzers, bugs in those systems are easier to locate and fix.

2.1.2 Programmable Logic and Reprogrammable Systems-on-Chip

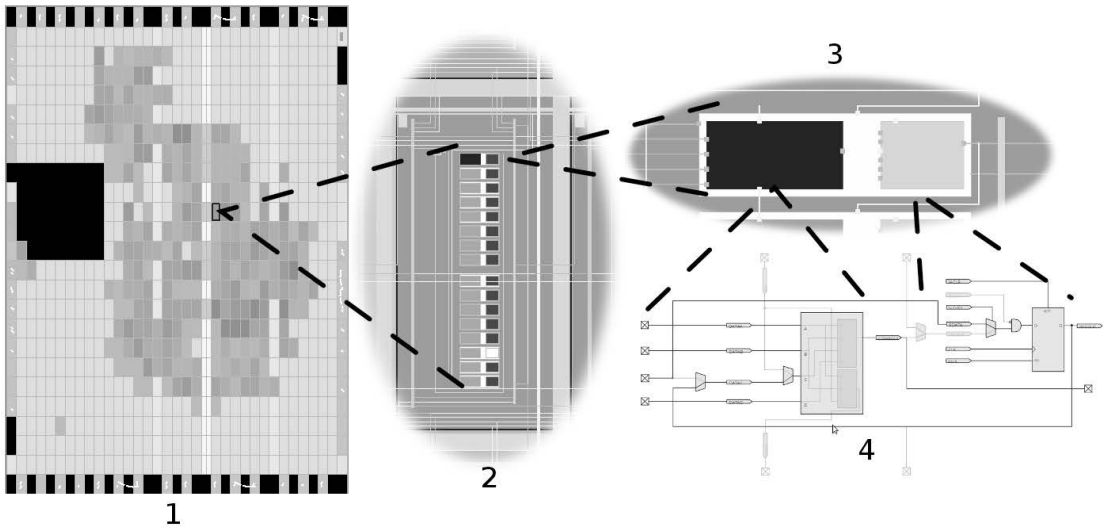


Figure 2.1: Small FPGA structure showing the die representation (1), a block containing many logical elements (2), a single logic block (3) and finally, the internal details of a logic block, highlighting the look up table and flip-flop (4)

In contrast to processor architectures that execute compiled code trough architectural microcode, an entirely different class of devices that differ fundamentally

1. Example include devices based on ARM Cortex-M3, such as NXP LPC1311 or ST Microelectronics STM32F100

in the way they process data exist: FPGAs.

FPGAs are based on a very large number of relatively simple logic primitives composed of look-up tables, single-bit registers (flip-flops), small and distributed RAM memories and sometimes dedicated hardware units for specific digital signal processing, communication functions and advanced Phase-Locked Loops. With enough FPGA logic primitives, one can essentially build any complex digital circuit. In their smallest offerings, FPGAs can be used to attach multiple circuits together or perform protocol adaptation. Their flexibility allows them to replace many different components on a circuit, often reducing the final bill-of-material. Their re-programmability is their main advantage, allowing the engineers to accept changes in the design and increase visibility and control when embedded in complex systems. Figure 2.1 shows the typical hierarchical structure found in modern FPGA in ascending level of detail as one can see when zooming in on the device representation [2].

FPGAs offer flexibility because they allow the designer to compose logic functions such as *OR* and *AND*, adders, multipliers, barrel shifters, memory and so on. By combining those primitives, one can architect an arithmetic and logical unit, an instruction decoder and a register file. Building up from those blocks allows the design of a custom processor optimised for a given application. The resulting device might be slower (frequency-wise) than its ASIC equivalent. However, the development cost of modern ASICs is constantly increasing, making FPGAs more and more economical in new applications. Such customized processors are typically very good at manipulating low-level data streams (bit manipulations) and can offer much lower latencies and faster response time than a software-based solution running on microcontrollers.

Advanced, state-of-the-art FPGAs can perform a lot more than *glue logic* or *protocol adaptation*. By combining logic elements and instantiating IP blocks, one can engineer an architecture that will do digital signal processing with a level of performance that outperforms any standard microprocessor, especially in fixed-point processing. Such examples can be found, for example, in large telecommunication

2. The FPGA illustrated is a low-cost Altera Cyclone III FPGA (EP3C10). The graphical representation of the logical elements were extracted from Altera Quartus II 10.0 using the Chip Planner utility.

(core broadband) switches, military radar digital processing and computer-aided tomography equipment. In some applications, the FPGA is the only type of device that will be an *economically* viable solution to handle the very large input/output data rates and keep up with the parallel processing requirements.

For low volume applications, the FPGA may also fully replace a custom ASIC. It will be faster to develop, have lower non-recurrent engineering costs and can tolerate a few bugs since it can be re-programmed. With the ASIC route, a bug will result in a lot of expenses to fix. Since the FPGA manufacturer can sell a given device to hundreds of different customers, the FPGA technology can use a very advanced lithographic process. The increased cost will be amortized on the larger production volumes. The FPGA is effectively slower than an equivalent ASIC for the same logic function, but it usually benefits from a generation or two of semiconductor process improvements and is often *fast enough* for the intended application. FPGAs are very successful products and are gaining ground on what used to be ASIC territory only a few years ago. FPGAs were once part of the *glue logic* of circuits, but are now a central element of many products. They can act as memory controller, switching matrix, protocol adaptation layer and often, are the actual computation unit of the product. The latest generation of FPGAs are astonishingly dense structures with over 2 million flip-flops. With those devices, the FPGA will definitively not be *glue logic*. They can integrate multiple CPU cores and advanced memory and communication controllers.

Because of their re-programmability and their ability to emulate any digital logic circuit, the FPGA is a very important tool in enabling rapid prototyping of ASIC circuits. Furthermore, newer generations of FPGA allow partial dynamic reconfiguration which allows sections of the device to be re-programmed while keeping other parts of the device active. This has particularly interesting applications to prototype the advanced debugging methods presented in this thesis. FPGAs will allow the emulation of complex circuits and their re-programmability can be a great aid to debugging. Sequence and assertion checkers can be instantiated inside the FPGA, next to the logic under debug. In doing so, dynamic observability of the device operation can be greatly enhanced, as will be covered in the next chapter.

Recent advances from leading FPGA vendors [1] allow heterogenous systems

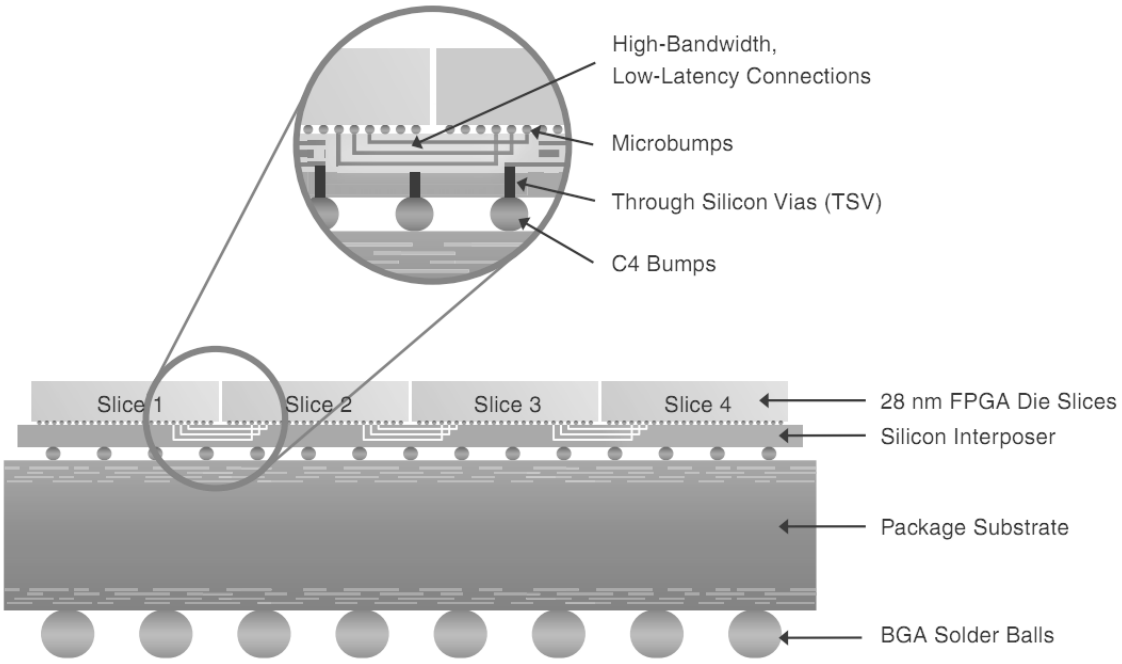


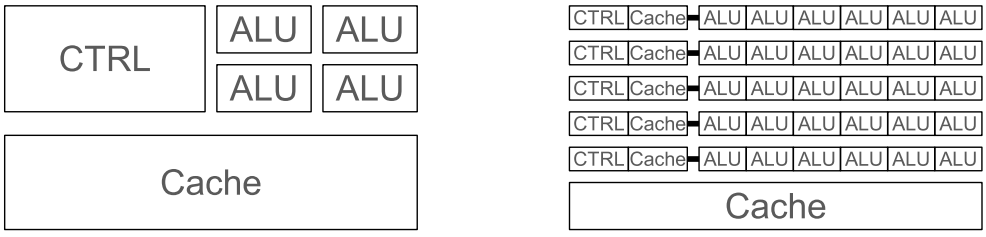
Figure 2.2: State-of-the-art Xilinx (1) FPGA interconnect using through-silicon vias to integrate multiple dies in a single package

(Processors, FPGAs, ASICs) to be reliably packaged in a single device, allowing even more logic density and functionality to stem from this novel interconnect and packaging technology. Figure 2.2 shows the process of combining multiple dies with through silicon vias and microbumps to achieve the highest integration density.

Currently, FPGA debugging tools mostly focus on integrating the equivalent of a small logic analyzer in the re-programmable fabric. By using some spare RAM memory blocks, samples can be accumulated inside the device and once captured, they can be sent out to a workstation for analysis. The triggering logic can include multiple levels or states, but generally, does not support complex temporal expressions. The universal support for logic circuit in the FPGA allows the use of a more advanced debugging strategy, but the code has to be provided at the register-transfer level to the FPGA tools. This thesis addresses the automatic generation of hardware register files to assist in this process in Chapter 4.

2.1.3 Graphic Processing Unit Programming

One of the earlier assistants to the processor, the video card graphic accelerator, is now surpassing the CPU (s) of most modern personal computers in its parallel floating point calculation ability. As CPU architects push more hardware resources into speculative execution, cache coherence mechanisms, virtualization and memory management, not much area on the silicon die is left for the actual floating point and fixed point arithmetic logic used to perform the actual computations. On the other hand, *Graphic Processing Unit* (GPU) architects always had to worry about the massive computational requirements of 3D image synthesis and manipulation. Modern games require real-time rendering of texture-mapped three dimensional scenes composed of millions of polygons. The algorithms used to render 3D scenes necessitate a large number of matrix operations. During its evolution, the graphic card, which started as an accelerator for matrix multiplication, slowly evolved into a massively parallel array of very specialized processing units capable of executing complex routines. A recent GPU video card can exceed a teraflop – one thousand billion floating-point operations per second. Yet, that "supercomputer" is affordable enough for anyone to purchase at any electronic retail store. This considerable parallel computing capability is reached using hundreds (sometimes thousands) of relatively simple processing units executing specialized instructions.



(a) Multicore CPU **(b)** Multicore GPU

Figure 2.3: Multicore CPU versus multicore GPU showing how much more area is dedicated for the control and cache memory in a CPU architecture when compared to the GPU architecture.

Figure 2.3 (adapted from NVidia CUDA Manual [29]) illustrates the fundamental architectural difference between a multi-core CPU architecture and the GPU ar-

chitecture. The multicore CPU has a significant area dedicated to cache memory and control while the GPU has a lot more area dedicated to the *Arithmetic and logical units* (ALUs) at the expense of much less sophistication in the control structures. This leads to a more restrictive and more complex programming style on the GPU, but outstanding performance when all the ALUs are actively used.

The typical modern personal computer thus embodies two very different computing paradigms running side-by-side. The main CPU has a few very versatile cores capable of executing a few billion complex instructions per second. The GPU has an array of simplified cores all running in parallel, capable of computing hundreds of billions multiplications, additions or logic manipulations per second.

This dual solution approach to the design of the personal computer leads to a few interesting attempts at utilizing the massively parallel computation power its GPUs can acheive to solve problems outside the realm of 3D image synthesis and rendering, such as molecular simulations and engineering finite-element analysis. Leveraging the GPU computing power is rapidly gaining in popularity. However, the complexity of writing (and especially debugging) the specialized GPU firmware has limited its growth. Very few programmers can write applications that make efficient use of the computing resources available in a GPU. New programming and debugging frameworks are helping to improve on that, but GPU programmers are also facing the limit of parallelism that can be extracted from a given algorithm, so both the CPU and GPU architectures are going to remain side-by-side for quite some time.

GPU architectures also require solid support for debugging. In most cases, a model of the GPU is used to assist the developer [30] to do the initial work of porting an algorithm to a GPU and debug it. Some researchers even go to the extent of full system simulation with hardware models to ensure proper integration of the drivers and GPU integration [31]. A lot of effort in programming for those architectures involve performance tuning, getting rid of subtle errors such as out-of-bound access and race conditions, so advanced methods to trace the execution flow were explored [32].

2.1.4 Computers and Virtualization

When CPUs started to integrate multiple cores, many observed that in typical systems those cores are not used all the time. In fact, CPU utilisation on a typical workstation is very far from its theoretical limit when averaged over a long period of time. Since computers use a significant amount of energy, multi-core systems become under-utilized if they are running a single operating system and selected user applications. In big corporate centers, this leads to wasted resources (electrical energy) and unnecessary costs (cooling, maintenance).

The disconnect between the hardware levels (registers, cache memory, interconnect) and operating system primitives (virtual memory, kernel task management) increases as programmers attempt to abstract the machine in a way to reuse an ever larger software code base without modification or re-compilation. The cost of developing and maintaining software systems has become so high that full machine virtualization is now an acceptable solution for many obsolescence problems. The cost of re-compiling (porting) the code base to the newer systems is often exhorbitant since programmer resources are limited, expensive and the port to newer operating system will introduce new, hard-to-fix bugs.

With virtualization, the processor microarchitecture is augmented with a few additional instructions and memory management modes such that the processor can effectively run multiple *Operating Systems* (OS) concurrently, with each OS being under the (selfish) impression that it runs alone on a dedicated machine. Information technology departments rejoice with the physical and power reduction of running multiple services on a single physical machine. The end user doesn't see any difference since network virtualization enables the virtual machine to appear exactly as a physical workstation on the local network. Virtualization technology helps isolate each application and OS interdependencies and allow for a more efficient use of computing resources.

Virtualization is also a very powerful mechanism to assist in the debugging of the internals of a running system. Since the operating system effectively runs in a sandbox, a crash at the core of the virtualized OS will not bring down the entire computer. The underlying *paravirtualization* layer and associated operating system will remain operational. This method, when viewed from a debug perspective

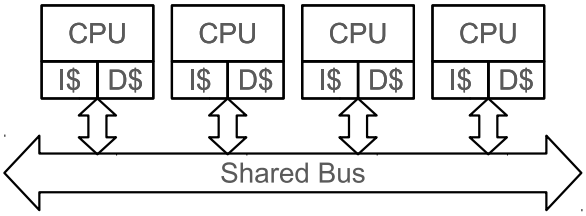
will allow a more efficient debugging process for distributed systems [33]. It also allows more advanced debugging and analysis of driver behavior [34]. While the debug of virtualized systems is an interesting topic and not the primary focus of this research, it shows that with proper hardware support (in this case specialized instructions), debugging abilities can be greatly enhanced.

2.1.5 Multi-core System-on-Chip and Network-on-Chip Evolution

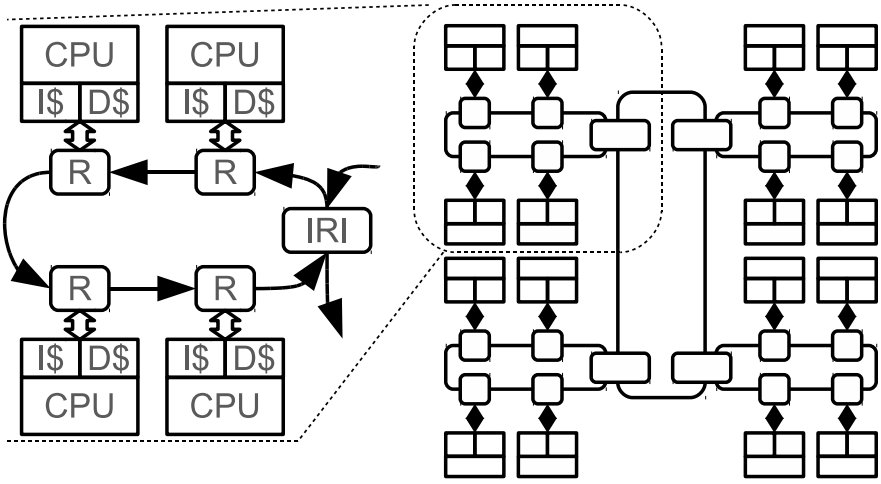
The modern microprocessor aims at processing a given stream of instructions as fast as possible. In a typical stream of instructions, parallel elements can be automatically extracted, leading microprocessors to implement many forms of speculative execution, which provide an overall gain in performance. Unfortunately, speculative execution greatly increases the processor's complexity. Performance becomes dependent on many different factors, including how the instructions are presented to the microprocessor and how the underlying instruction stream accesses the memory and computation resources.

Computing cores themselves have reached a level of complexity that stands to benefit from a parallel combination of the same units rather than improvements in the core themselves. This is shown by the relatively recent explosion of multi-core microprocessors. A multi-core machine (where the processor cores share a common level of cache memory) was an exotic system only a few years ago, usually spending most of its clock cycles amusing researchers in computer science and engineering or benefiting very large military or commercial endeavours. Nowadays, even a low-cost handheld music player can include multiple processing cores allowing the device to run a combination of processes in a power-efficient way. Individual core processing performance has significantly slowed from its rapid progression in the 80s and 90s. The new processors get their performance advantage from parallel computing on multiple cores enabled by the higher density of smaller geometry lithography. To keep software complexity under control, these cores share a level of memory. However, placing multiple cores on same memory structure creates resource contention and the performance doesn't scale linearly. Intel and AMD now offer chips with 6 cores, but experts agree that further "hor-

izontal" scaling of cores will not bring major benefits [35]. Multi-core processors thus suffer from a performance bottleneck stemming from sharing memory among the computing cores. As programmers prefer to work in a system where each processing element "sees" the same memory, the mechanisms to ensure cache consistency and coherency become more complex as cores are added. Furthermore, the physical distances and logic complexity increases the processing delays. This ultimately penalizes the performance of the system, negating the benefits of adding more processing cores.



(a) Four CPUs on a Shared Bus



(b) 16-cores NoC-based System

Figure 2.4: Multiple CPU cores sharing a single bus suffer from limited scalability. NoC-based systems address this problem through hierarchy, paralellism and locality of traffic. ***I$*** stands for instruction cache and ***D$*** stands for data cache.

Researchers have thus turned to the NoC paradigm [36, 37] as a way to handle the scaling up of the number of processing core. Figure 2.4 illustrates an example of this transition through the use of on-chip routers. Instead of having each comput-

ing core share their memory on a single bus, the cores are tiled and interconnected in a way that resembles a high-performance computing cluster. The interconnect is done using on-chip networking primitives. The obvious benefit is the lower interconnect distances and higher density, which leads to lower power usage. This new approach to multi-core computing can offer a path for future scaling of computing power. This field of research is quite young [36]. Which topology, interconnect medium or programming paradigm will prevail in the next years is far from obvious (and will depend significantly on the end application), but this approach to multi-core scaling shows great potential. One important aspect to consider is that as multi-core systems evolve, the software's ability to exploit these cores become increasingly more difficult. This novel interconnect strategy requires careful analysis of the traffic flow pattern since congestion and latency in the communication network will heavily penalize the performance of the final application. The debugging of this new computing paradigm will not be simple as both the logical structure and temporal relationship between computing elements have to be considered in parallel for the solution to work well.

As one considers the software as a dominant part of a digital logic solution, a successful product will more and more have to incorporate a holistic debugging strategy that leverages the visibility and control of the hardware structures with the abstraction level of firmware libraries. In future designs of multi-core or NoC systems, the low-level firmware that sets up the routing and data flow cannot be seen as an afterthought since it will become an integral part of the critical path the system, thus has to be debugged in parallel with the hardware device. As highlighted in the list of properties presented in Section 1.5, this work proposes a way to embed in hardware the means to dynamically observe sequences of events occuring inside the multi-core system. This way, the critical performance counters that programmers can use in multi-core sytems to help their integration of firmware can be derived from the system-level properties defined during verification.

In summary, many different hardware architectures and computing paradigms exist and more will evolve. In all cases, one important issue always remains: debugging the devices first and then the system. The logic gates, microcode, firmware and user code all have to be well integrated to obtain a stable and reliable

solution. From the early days of relay-based computers to modern day, billion-transistor multi-core systems, they all require significant debugging at every level of their design process and integration. The evolution of digital circuits does not converge to a single, universal and simple solution. A problem may be particularly well suited to be solved using GPU computing while another may be a lot more efficiently handled by an FPGA. Each solution has its own and unique debugging strategy. The integration of debug tools in heterogeneous architectures to form a comprehensive strategy will require a considerable abstraction effort. Even the most dedicated and brilliant engineers will not be able to fully grasp digital circuits of such scale.

2.2 Terminology

The semiconductor industry, being one of the major engines of modern progress, spans large industrial segments, each with its own set of terms and jargon. This thesis will be using the technical terms presented below in the context of large-scale digital design. Often, the general English meaning carries ambiguity, so a few examples will try to clarify their meaning in the context of this work.

Many terms are adapted from the *Property Specification Language* (PSL) [38] language reference manual, which defines a *verification* language. Therefore, the terms are well suited for a system-level approach as covered in this thesis since the verification process is situated between the conceptual design process (customer requirements, specifications) and the chip production phase (actual fabrication and automated testing of large quantity of IC devices).

– *Verification*: The process of confirming that, for a given design and a given set of constraints, a property that is required to hold in that design actually does hold under those constraints. In a more general context, the verification process aims at checking that the logic circuit actually behaves as intended in the design specification. *Verification* requires that the design specification be written such that each requirements is unambiguous and fully describes the intended output for a given set of inputs. Unfortunately, real-world specifications can only approximate this ideal. In most designs, natural language

(e.g. English) is used along with strong verbs aiming at emphasizing the requirements and objectives of the design (e.g. the unit *shall* acknowledge the transfer *within* 4 clock cycles). As designs get more complex, the verification process will require the use of more *formal language* to record their specifications.

– *Dynamic Verification*: A verification process in which a property is checked over individual, finite design behaviors that are typically obtained by dynamically exercising the design through a finite number of evaluation cycles. Generally, dynamic verification supports no inference about whether the property holds for a behavior over which the property has not yet been checked. This step is often referred to as *simulation* by tool vendors and the industry. Dynamic verification is simple to understand by using a small binary counter as an example. The *dynamic verification* of this counter would be to check that at each clock cycle, the output is incremented by one. If the counter has a reset input, then one must check that when this input is asserted, the counter outputs zero. If the counter has N bits, it is easy to visualize that after 2^N clock cycles, the counter would have gone through all its internal states and thus dynamic verification would have covered its behavior. However, to be correctly used, the dynamic verification environment has to use some form of *checker* to predict the correct counter value and at each clock cycle compare the actual logic state with the predicted value. As one can see from this simple example, even the simplest counter requires considerations from a dynamic verification perspective.

– *Formal Verification*: This verification process aims at proving or disproving the correctness of an implementation with respect to a *formally* defined specification [39] using mathematical techniques. To contrast this method with *dynamic verification*, the counter example can be re-used. In formal verification, we would state formally that the counter's next value is always the current value + 1 with the exception that when the counter is saturated, the next value is zero. The formal checker would then look at the counter logical implementation and expand the current-state and next-state logic to explore

all possible states. One can see that for large counter, this state expansion will create a very large structure. A formal checker can then traverse the generated graph and validate that at each possible state, the specifications are met. The end result is *proof* that the counter either meets the formally-defined specifications or fail them (in which case the tool would provide the counterexample). In this method, only the formalized specifications and the unit under test are required. A test bench is not needed. In advanced designs, only a small percentage [40] of a complete device can be *formally proven*. The most problematic part being that one needs a formal specification of the device, which in itself may be harder to define than the actual RTL implementation of it. However, many experts still agree that *formal verification* can add a lot of value to the process [39, 40]. However its main use is still focused on validating CPU microarchitecture, critical aerospace subcircuits or similar well-defined structures. Note that *formal verification* also carries another meaning when discussing logic circuits. Some tool vendors call *formal verification* the process of proving the equivalence of two logic circuits (flip-flops and their associated cone-of-logic) when it should be called *formal equivalence checking*. Often, that "flavor" of formal verification is used during logic synthesis to ensure that transformation (optimisation) of logic functions did not introduce bugs.

– *Simulation*: A type of dynamic verification, also called *logic simulation*. This process is typically done on a workstation using a special software that compiles the HDL language into an executable form. This executable program manages a very large number of tiny "threads" which emulates the parallel activity of the modeled hardware gates, flip-flop storage elements and similar primitives. On a physical IC, each hardware element operates independently and in parallel. On a simulator, this is abstracted by continuously computing state updates and advancing the time in discrete steps (delta cycles). It is easy to understand that as a system grows in complexity, the simulation performance can only degrade. Thus, large systems become notoriously difficult to simulate efficiently, which leads to the use of a hardware accelerated process called *hardware emulation* (see below). During simulation,

many checks can be done at every clock cycle to ensure that the design meets the specifications. Simulation is used extensively to validate designs. In *simulation*, the design is a pure software model, thus visibility and controllability of internal nodes is straightforward.

– *Property*: A collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behaviors. It is by defining a complete set of *properties* that one can build up the formal specification of a design.

– *Checker*: An auxiliary process (usually constructed as a finite state machine) that monitors the simulation of a design and reports errors when asserted properties do not hold. A checker may be represented in the same HDL code as the design (they are usually limited to non-temporal checking if the language is VHDL or Verilog, for example) or in some other form that can be linked with the simulation model of the design. For example, PSL or *System Verilog Assertions* (SVA) are languages that offer flexible and efficient means of writing checkers since they support temporally complex expression semantics. Some verification tools offer the use of languages more apt at abstracting complex behavioral statements (e.g. SystemC, "e", OpenVera). In this thesis, a *checker* can be a software process running on a simulator (in a simulation context), but may also be a digital circuit performing the same task such as a hardware checker in a silicon device. This thesis refers to the *checker* as *assertion checker* since it is usually associated with an *assert* statement. If the checker triggers, it means that an assertion has failed, indicating that the design fails a given *property*.

– *Coverage*: A measure of the occurrence of certain behavior during verification (typically dynamic) and, therefore, a measure of the completeness of the dynamic verification process. For an incremental counter, we can consider each counter value as a coverage point. We can also consider an asserted "reset" input as another coverage point. Coverage is an important measure of the progress of the verification. Without coverage information, it is difficult to judge how much of the design has been verified and how much remains be-

fore one sends the chip to fabrication. Note that *complete coverage* of a circuit can mean different things to different people. In formal verification, completeness means that all the possible states have been verified. This is only feasible for very small circuits due to the state space explosion problem discussed above. More generally, the coverage is complete when a given set of coverage measurements have been reached. In typical designs, the coverage will rapidly grow at the beginning of the verification phase. However, as more and more of the "low-hanging fruits" of the logic space are covered by the verification process, ever more complicated stimuli are required to reach certain states. For a simple counter, it is quick and easy to let it reach all its possible values by letting it run. However, one could want the reset of this counter to be tested for each possible counter value and the test would need some automation to let it run to a given value and then apply the reset. This example show that depending on the verification goals, coverage metrics have to be correctly defined and only then can the completeness of the verification process be measured. In contrast, *formal verification* in this example would only require an extra property stating that if the reset is asserted, the counter falls back to zero.

– *Design*: A model of a piece of hardware, described in some HDL. A design typically involves a collection of inputs, outputs, state elements, and combinational functions that compute next state and outputs from current state and inputs. Typical hardware design languages are VHDL and Verilog. Some designs can be described in more abstract languages such as SystemC or SystemVerilog. New developments in synthesizing those more abstract languages are starting to bear fruit enabling the description of algorithms to be transformed into a *design* in a flexible and efficient manner. It remains that most of the designs are still implemented at the RTL level, mostly in VHDL or Verilog. The *design* representation evolves during the various steps leading to the production of masks. It may start as a block of SystemC code, progress to VHDL code to refine its behavior, then translated into EDIF (interconnected ASIC primitives) and finally transformed into GDS II (IC Layout format defining metal layers and physical structure).

– *Hardware emulation*: The process of using a hardware circuit to replicate the behavior of the final circuit. A hardware emulation system often adds substantial debugging capability by allowing the state of the design to be monitored at clock cycle boundaries. Because of the high cost of hardware emulators, some users will use FPGAs in a prototype to test the functionality of an ASIC chip before it is sent to fabrication. Some hardware emulators internally use FPGAs to support the end circuit along with monitoring and control. The main advantage of hardware emulation is the higher execution speeds that can be obtained when compared to a software-only solution. Simulation runs that may take many hours on a simulator may be completed in a few seconds on a hardware emulator. A *hardware emulator* is a machine built from re-programmable logic elements that will be set up with a representation of the logic circuit, often adapted to the hardware emulator [5] architecture. Unlike the workstation that uses a general CPU to execute the hardware model in one fast thread, the hardware emulator will have inherent parallelism and can execute many of the hardware "threads" in parallel. However, hardware emulation typically falls short of providing the performance of the final device.

– *Testing*: The process of validating the correct *fabrication* of a given hardware device. Usually, the *testing* process will be aimed at providing a good level of confidence that a given production IC device from a manufactured lot has no defects. The *testing* process is tuned for large volumes and aim to operate in a cost-effective manner. Since each IC produced is a copy generated from the same masks, if the original masks are fault-free, each copy should be "perfect", in theory [3]. However, random events during the fabrication steps (dust particles, contaminants, process variations) can all introduce defects which can affect the manufactured circuit and leave the final product unfit for use. The aim of *testing* is thus to rapidly identify and reject the defective dies (or in some cases, work around the defect by re-programming a section

3. If the masks used to fabricate a device have a bug (design error), then the bug will be present on *every* manufactured device. From a *testing* process point of view the manufactured devices would all be acceptable. The bug being due to a design error and not a physical problem during manufacturing the device.

of the device).

2.3 Modern Digital Verification Methodology

A simple custom logic solution ASIC below a few tens of thousands of logic gates can be modeled and realistically simulated using only the RTL source as a basis for its verification. Small FPGAs involving a low risk and similar simple designs can be "verified" simply by looking at the waveforms resulting from a simulation. The advantage of this approach is that only RTL development experience is needed. The designer can look at the waveforms, study the results of the simulation and interpret them as either *valid* or *invalid*. This technique works but suffers obvious drawbacks. The most obvious one is that every change in the design requires the review of all the test cases and an interpretation of the pass/fail criteria. A slightly more scalable approach would be to save the *golden reference* trace and use it to compare the updated design. However, simple changes in the design may result in a new waveform that renders the *golden reference* waveform unsuitable for comparison, requiring a new visual analysis. The ad-hoc verification technique using golden reference waveforms (or saved traces) falls apart very quickly for complex systems. Many simple changes or bug fixes transform the waveforms so significantly that the process quickly uses too many engineering resources. A higher level of verification abstraction is needed to make the verification process tolerant to small changes in the logic circuit (e.g. changes that are made to improve timing of logic paths, but which do not affect the functionality).

As a design grows in complexity, size and feature set, the number of test cases required to validate its logical functionality becomes increasingly large and the resulting waveforms become undecipherable at the RTL and bit level (think of time-multiplexed systems or encrypted data). Software is then required to assist in the interpretation of the simulation data sets and as they propagates in the memories, allow the designer to track the progress of abstract data units, computation pipelines and logic gates of the circuit. The verification problem is further constrained by the fact that larger designs become increasingly computationally intensive to simulate on a workstation. The combination of both complexity and

size requires a change in abstraction level to allow larger systems to be successfully verified.

To utilise the massive logic resources of newer FPGA or ASIC processes, a large increase in abstraction has to be accomplished in the verification process. This is becoming critical and will be a necessity to accomplish the verification and integration of large SoCs and even more for the NoCs of the future. Not only does the verification environment have to change, but new languages have to be integrated in the verification flow to improve the designer's productivity.

A well-known approach to enhance the productivity is the use of pre-built modules, commonly known as *IP Cores*. When used, they greatly accelerate the design flow, but they require a different approach to verification.

2.3.1 Black Box and White Box Verification

The verification of modern designs typically employs a multi-level approach to the problem. It can be approached in two very different manners, namely *black box verification* and *white box verification* based on the internal controllability and observability and intimate knowledge of the circuit under verification. Often, for a same circuit, both approaches may be simultaneously used with the name *gray box verification*.

The black box verification of a design does not assume any knowledge of the internal circuit, thus the task is based only on the input/output behavior and compares this behavior against a *model* or its specifications. This approach is required, for example, to verify an IP core that is bought from an external entity in the context of design re-use [41]. The source code would not be available for IP protection reasons or because the design is the actual circuit representation and not available in high-level code, for example when using hardware macros built from design primitives such as LUTs in FPGAs or primitive cells in an ASIC design flow. In the black box mode of circuit verification, the core of the verification effort will be located at the boundary of the design unit (interface ports) such that the input and outputs to the modules can be accurately monitored, recorded and predicted. Often, the IP interface protocol is well specified (e.g. Advanced Microcontroller Bus Architecture bus [42], Wishbone [43]). Verification of the IP block interfaces is

required even if the block comes from a reputable source (and is presumably thoroughly verified) since one has to prove that it is correctly *integrated* in a new design. For example, the IP block may be used in a way that is beyond its specifications and may cause a protocol error on its interfaces even if it was designed and verified by the provider. The IP provider could not envision their block to be driven in such a corner case, thus did not verify it under those conditions. The use of an assertion-based methodology can greatly assist in the integration of black-box IP since the assertion checkers can be left as part of the IP block as a way to ensure that the module is used as intended. Section 2.4 and Chapter 3 provide more details and usage scenarios since assertions are a key element of the presented work.

An example of white box verification would be for design blocks produced as part of the design implementation process and usually built from RTL code (e.g. VHDL, Verilog) or higher-level design methods such as synthesized RTL from C or SystemC [44, 45, 46] or using automated architecture generators such as GAUT [47]. In this case, the designer has a good visibility on the internals of the design block and may add monitor and special logic checkers to the logic to ensure that the internal circuit operation respects certain fundamental assumptions. In the case of high-level synthesis, however, the internal structures of the generated RTL is relatively cryptic as they are the result of complex optimizations, automatic register scheduling and state-machine generation. Therefore, pure white-box verification becomes too difficult and a compromise has to be reached where visibility and controllability of the internal nodes is traded off. In such scenarios, a gray box verification approach is more suitable as the internal details of the circuits are too complex to directly address, but the data flow, internal structure and overall circuit behavior is nevertheless well understood.

2.3.2 Structure of a Verification Environment

Although most verification environments have to be adapted to the particular type of circuit under verification, they tend to follow a common structure with the details varying depending on the complexity of the circuit being verified.

A prototypical verification environment is presented in Figure 2.5 and illustrates the *Device Under Verification* (DUV) being driven by two *Bus Functional Mod-*

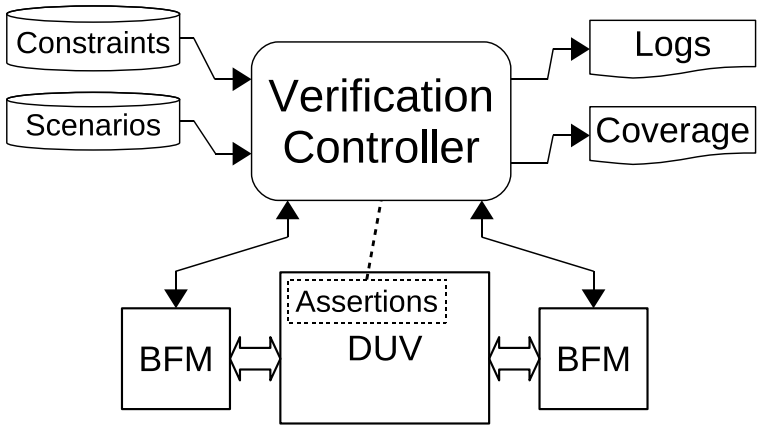


Figure 2.5: Prototypical Verification Environment

els (BFMs) under the control of the *Verification Controller*. The DUV may represent a sub-module of the entire design or the complete circuit (including the printed circuit traces and multiple ICs), depending on the level of verification. Typically, the top-level simulations are only performed as a sanity test to verify the main interconnections and glue logic rather than as a way to fully verify the design compliance. This is mainly because of the very long simulation times (many days) required to run the full simulations using the complete device model. Once a certain level of confidence is achieved with the block-level simulations and top-level simulations look sane, the designers will move on to emulation or FPGA prototypes in order to put more simulation clock cycles on the design and attempt to uncover the more obscure bugs that can only be triggered when a much larger set of scenarios are performed.

The DUV is connected to a few BFMs. These address the translation from abstract structures (e.g. packets, flits, memory transactions) into their time domain, clock-accurate representations. This translation process is often a complex set of routines, but the increase in abstraction facilitate the maintenance and creation of verification scenarios. Most of the generation logic benefits from working at much higher levels of abstraction and can be made independent of the exact timing details of the RTL. For example, the verification of “smart” networking switches requires a detailed generator of Internet Protocol packet streams. A “smart” switch has to keep track of connection streams starting and stopping, and those span mul-

tiple packets. An Internet Protocol stream generator can be developed once and re-used in multiple simulation environments without any changes at a high level of abstraction. However, it will require a new BFM for each interface to the various internal hardware blocks. For example in one part of the chip the packets may be present on a wide (e.g. 128-bits) bus and in the next part of the chip, the same packet may be serialized on a single differential pair at the pins. Often, the high level generators will be written in a programming language suitable for easy incorporation of existing libraries (e.g. C/C++) and then integrated in a verification environment that uses libraries to extend the representation of parallelism and typical hardware concepts. A good example of such language is SystemC [44, 48].

Finally, the verification environment will include a few standardized simulation control points that can initiate the DUV reset, setup the proper clock inputs and perform the typical configuration of the DUV registers. This places it in a mode where the DUV can execute the verification scenarios through the BFM.

2.3.3 Verification Classes

Keating et al. [41] separate the verification types in general categories. These authors use the word "testing" (e.g. real code testing) as part of the names for each verification class. However, to reduce confusion with the post-production chip *testing* process, this thesis will use the word "verification" (e.g. real code verification) for each type.

Real code verification is the core of the verification process and aims at proving the correct operation of the DUV in *normal* operation and with real application scenarios.

Compliance verification is typically done with a test suite that focuses on a design meeting some standardized specification. Sometimes, these test suites can be acquired through the purchase of the verification IP from specialized companies.

Corner-case verification focus on the limits of the design, in contrast to the previously mentioned *real code verification*. In corner-case verification the scenarios will push the design into error conditions, for example by filling up FIFOs until they overflow or by observing the recovery of transient error conditions by introducing invalid conditions. Some conditions may not be realizable from the

external DUV inputs, so some form of *gray box* approach has to be used to inject the condition directly inside the logic.

2.3.4 Constrained Random-Based Verification

Recently, the use of **random-based verification** has increased. This can be noted by the dedicated tools sold by Cadence and Synopsys and the long-term goals of SystemVerilog to provide a comprehensive language that enables both hardware description and hardware verification.

This approach to verification does not require the engineer to anticipate each scenario, but rather to set up the verification environment such that typical parameters (packet length, burst count, response latencies) are selected randomly at the start or during of the verification scenario execution. The parameters may even be made adaptive to the simulation in progress. The entire simulation derives from a single random *seed* to allow the exact re-creation of the scenario, should a bug be discovered. This verification approach basically allows the computer to *create* its own verification scenarios within the bounds given by a set of constraints. In this context, the simulations would be run in batches and can uncover very obscure bugs in the design since the unpredictability of the input scenarios can force the device in a state unanticipated by the designers. Dedicated verification languages such as *e* or *Open Vera* have built-in support for constrained (biased) random-based generation. This approach to verification is very productive, but requires large computation resources. For large ICs, this mean *compute farms* that can host up to ten thousand compute nodes.

One of the reasons that *constrained random-based verification* has gained ground in large projects [49] is that when verifying complex circuits it can be used for both *real code verification* and *corner case verification*, by applying the proper constraints to the generation unit. For example, in the verification of a network switch, a random packet generator could be designed to produce packets containing any kind of source/destination and protocol information along with length ranging from 0 bytes (headers only) to thousands of bytes. If that generator was used without any constraints on its generation, it would produce what amounts to digital "white noise" at the input of the DUV and would not be that helpful early in the

verification effort. To address this problem, a set of constraints is applied to this generator such that it will produce valid packets with proper headers and a valid body size. The generator code remains the same, only the *set of overlaid constraints* has changed. If the generator produces valid packets, then the verification environment is operating in *real code verification* mode.

Assuming that the engineers want to test the limits of the design by sending a sequence of large packets, it becomes possible to constrain the generator such that it will start producing the biggest packet that the design can support and sustain this generation mode for a given number of cycles. The same generator is now operating in the *corner case verification* mode. If the engineer wants to test an error condition, he could temporarily force his generator to issue a packet bigger that the maximum size that is allowed and thus enter the realm of error recovery of his circuit. Since the same generator can now be used to produce both typical, valid and invalid data streams, the productivity gains are significant.

A lot of research is ongoing in this area, with teams trying to generate the random scenarios in some "intelligent" manner by tuning the probabilities of the random generators [50]. Other researchers [51], including our group at McGill [52], aim at directly generating the test vectors from formal temporal specifications.

2.3.5 Golden Reference Model and Predictor

The constrained, random-based generator explained in the previous section will generate the scenarios. However, this method alone does not *validate* the DUV. Two more elements must be present in the verification environment: *predictors* and *coverage monitors*. Together they form the *validation* layer of the verification environment.

One of the most difficult challenges of advanced verification is the creation of that *predictor* (also called *golden reference*). Usually, the predictor is coded in a high level language that will attempt to abstract out low-level clock level timing and model the *correct* input-output behavior of the design based on the specifications. As an extra safety net, the *predictor* is usually coded by a different team in large projects to ensure an independent interpretation of the specifications. The verification task becomes a comparison between the DUV output and the reference model

output. The challenge usually lies in handling subtle differences in the interpretation of the system-level specification from which both the hardware design and the reference model are derived. The specifications are usually written in English, which inevitably creates ambiguities that lead to differences in interpretation. Assertion checkers derived from temporal logic are among the tools that can be used to code the *validation layer* of the verification environment. Unlike *predictors* however, they only handle the low-level temporal checks, so *predictors* are still required. Temporal languages such as PSL thus assist in the creation of predictors since they allow parts of the predictor to be built up from smaller temporal sequences. In most complex verification projects, there is no single solution to the verification problem and the approach will be a mix of reference models, temporal statements and even post-processing of log files, when simulation results have to be processed using advanced algorithms.

Often, multiple reference models are "connected" together to emulate larger, more complex systems. For example, Lin et Al. show how they can structure the verification environment [53] to address the complexity of verifying networked systems (applied in their verification of an IEEE 1394a PHY Core). They use the design natural network topology as a way to increase the complexity of generated verification scenarios while reducing the burden of writing their own scenarios. Vitullo et Al. [54] offers good examples of the challenges faced when verifying large-scale NoCs.

In industrial applications, the reference models represent a very valuable form of IP. Some companies even specialize in selling high-quality reference models of hardware units [4].

2.3.6 Measuring Coverage of the Verification

The last element to *close the loop* on constrained random-based verification environments is the process of *coverage monitoring*. Once a testbench structure has been created, BFMs have been written to connect to the DUV interfaces and a reference model for the particular feature is ready, the system can then be simulated

4. Verification vendors such as Cadence, nSys or Synopsys directly sell verification models for most of the modern complex interfaces used in large designs such as double data rate memory controllers, PCI-Express and Ethernet.

and error conditions or inconsistencies in the DUV will be noted by the test environment. Using randomly generated scenarios, the testbench will exercise many different logic paths and will thus uncover quite complex bugs. However, the very large state space defined by the design possible states make it a tremendous challenge to run in a simulation. As an example, suppose that in a NoC, one wishes to verify a simple on-chip router unit. Assuming that the *flit* (transfer) size can range from 1 to 32 clock cycles, the destination address can vary from 0 to 3, we get 32*4=128 different combinations of values which would test those two parameters and their combined effect. More complexity stems from the temporal variations that also need to be covered, such as a flit going to port 0, followed by one going to port 1. This increases the number of scenarios required to cover the temporal variations in destination addresses.

Repeating the same analysis would lead to an interest in verifying that the router can handle one large flit followed by one small flit. The definition for a large flit could be 20 to 32 clock cycles and the small flits could be defined as 1 to 4 clock cycles. Then a scenario would be constrained to alternate large and small flits. However, another approach exists: instead of forcing exact scenarios via constraints on the generator, leaving the flit size to a random number is bound to eventually produce this scenario. However, in order to *guarantee* that that particular condition has occured, one needs to add *coverage* to the test environment. Monitoring the current flit size entering the DUV and previous flit size, along with performing an analysis using the last 2 sizes that entered the same unit will allow the verification engineer to know that the particular scenario has indeed occured at some instant in the simulation through the random generation process. The *coverage* information is usually kept in a database such that one can answer concerns about a particular scenario by querying the database for a match. Thorough coverage of the parameters along with no detected assertion failures and no discrepancies between the DUV and the *predictor* increases the confidence in the design robustness and is a *measurable* indicator of the verification progress.

Some coverage points may be difficult or near impossible to reach by having the scenario generator randomly try various input combinations (even with a lot of directed constraints). Some researchers thus focus on tuning the generators to reach complex states in the circuit by analyzing its structure [50]. They can then

obtain the required coverage a lot more rapidly.

Coverage information being the key to gauge the verification progress plays a key role in automated verification environments [55]. As such, *a lot* of coverage information has to be gathered during the verification process. As will be shown later, coverage can be collected during simulation on workstations, but can also be obtained by running a partially synthesized version of the DUV on emulators or hardware prototyping machines (based on FPGAs, for example). This thesis shows that some of the coverage logic can be efficiently written in temporal logic by using *sequences*. Those can then transformed into hardware primitives and integrated in emulators or even in the final silicon. This way, coverage can be collected much more rapidly and even when the device is released. Furthermore, coverage or sequence triggers can be very beneficial in the debugging process as will be shown in Section 3.1.

2.4 Assertions and Temporal Logic in Verification

In recent years, a new design and methodology has made great improvements in the resulting quality of verification that offers assistance to debugging. Assertion-based design [7] proposes using assertions in RTL code as part of the design methodology. Those assertions effectively provide layers of formalism to the specification and at the same time can be re-used during the verification to flag instances of a given design error detected through deviations from the specification. Temporal sequences and assertions are derived from the mathematical branch of temporal logic which was explored in the 1960's [56]. *Assertion-Based Verification* (ABV) has found recent supporters since it allows designers to formalize their specification in layers. As the device specifications are translated in *sequences* and assertions, the ambiguities are progressively reduced and the specification thus becomes a lot less subject to human interpretation.

In building the set of assertions defining a given circuit, the designers are effectively creating a lot of intellectual property in the process. This information is extremely valuable and many benefits can be drawn if it is re-used in various steps in the design process.

Using sound design principles and applying them to assertions, the teams that verified the SUN SPARC CMT [57] cited the following as their first recommendation on using assertion-based verification:

> The philosophy of assertions has to be "write once, use always". Ideally they should be usable across all levels of simulation, formal verification and hardware emulation.

Our goal of using assertions in the final silicon is shared with other researchers including notably the work of Gharehbaghi, Hessabi and his group [58, 59] at Sharif University and the commercial endeavour of Abramovici at DAFCA [60].

This work aims to extend this idea to the final silicon prototypes (and even released devices) to keep the benefits that they bring when it comes to bug tracking. Recent research tools have allowed temporal expressions to be transformed into hardware circuits that perform the checks in a device [61, 62, 63].

If, ultimately, one is interested in obtaining a correct *physical* device, then the verification process cannot only be considered in isolation. Re-using the verification IP development effort and leveraging the protocol analyzers and other instrumentations present in the test environment such that they can be re-used in the final silicon is a noble goal. Furthermore, future designs that will lean towards the use of high-level synthesis, which ultimately produces code, has to be considered as a black box, making the verification of internal operation difficult. However, the set of properties that the design must meet are still applicable at the boundaries and thus can be attached to the hardware block. As typical assertion languages such as PSL or SystemVerilog *attach* to hardware modules under verification, they would be applicable to validate the output of high-level synthesis tools.

As the saying goes : "trust, but verify.". One has to trust the algorithms that will produce the RTL code from a high-level description or algorithm, however, the high stakes involved in delivering a functional circuit require that a fair amount of effort also be spent in controlling the output of the high-level synthesis tools. After all, even well-built high-level synthesis tools are not themselves immune to bugs in their internal logic.

2.4.1 Design for Debugging

Since debugging in the process of verification consumes a significant amount of effort, researchers and industry groups have highlighted the need for a DfD methodology that can be followed through up to the silicon. B. Vermeulen at NXP (a large semiconductor manufacturer) has highlighted in multiple publications the need to integrate and plan debug as part of the design process [64, 65, 66, 67]. Their group emphasizes the necessity to have a to have a Debug Methodology, Debugger software and Design for debug Tools for a proper silicon debug program. As design get denser and faster, the internal data rates require silicon assistance for the debug.

In other cases, researchers want to detect specific race conditions in complex systems such as a NoC [68] since those are typically *very* difficult to debug. Their DfD method can classify eventual errors as timing, FIFO protocol violations or timing errors by analyzing the distributed monitor's outputs. Goel and Al. cover [69] the need to have special hardware when dealing with multiple clock domains which is very common in large systems. In order to obtain consistent results, the hardware clocks have to be stopped in a certain order. They explain well the data invalidation problem and how to work around it in hardware.

Pyron et al presented an early paper [70] documenting the use of schmoo plots and providing a good coverage of possible silicon defects. LeBlanc's publication [71] clearly illustrates the problems in debugging parallel systems. Some type of log/event order needs to be kept. However, with the new systems integrating very complex functions purely in hardware, and the very high throughput of those systems, one cannot accept the speed penalty of using software to log hardware event orders. Hardware assertion checkers can monitor conditions and with a hardware timestamp mechanisms can report the assertion firing order which will assist in determining the root cause of the problem. Peishl et al. discuss [72] the problem of fault localization in hardware design by reducing the model complexity based on error traces which ultimately help the designer find the root cause of the problem.

Conventional RTL debugging is based on overlaying simulation results on structural connectivity information of the HDL source. This process is helpful in lo-

cating errors but does little to help designers reason about the how and why of errors. Hsu et al. show [73] how automatic tracing schemes can shorten debugging time by orders of magnitude for unfamiliar designs and how advanced debug techniques reduce the number of regressions by emphasizing a methodical approach to extract, analyze and query a design's multi-cycle temporal behavior. The same team explored techniques [74] that also cover the register selection analysis for DfD and multi-level design abstraction correlation for viewing values in the RTL. Experimental results show that visibility enhancement techniques can leverage a small amount of extracted data to provide a high amount of computed combinational signal data. Visibility enhancement provides the needed connection between data obtained from the DfD logic and HDL simulation-related debug systems.

Hyunbeam et al. [75] presents DfD methods for the reuse of NoC as a debug data path in an NoC-based SoC. They propose on-chip core debug which can support transaction-based debug. An interface unit is also presented to enable debug data transfer through an NoC between an external debugger and a core-under-debug. However, they make no mention of using assertion checkers in their work. As will be explained later in Section 4.4 and Section 5.4, our work follows a similar structure, but integrates assertion checkers in the hardware.

Abramovici et al. present a Design-for-Debug (DFD) reconfigurable infrastructure [60] for SoCs to support at-speed in-system functional debug. This is a distributed reconfigurable fabric inserted in the RTL that provides a debug platform that can be configured and operated post-silicon via the JTAG port. The platform can be repeatedly reused to configure many debug structures such as assertions checkers, transaction identifiers, triggers, and event counters. This work is the closest to the dynamic partitioning algorithm that is presented in Section 3.3. However, that publication does not detail the algorithms used to optimize the partitioning.

Gharehbaghi et al. covered a similar approach [76] for SoC to the one presented in this thesis with some important differences. They use their own temporal logic expressions. In contrast, our work focus on PSL and SVA re-use to leverage the design effort expended in the verification process. They rely on trace capture/compression to extract the data outside the IC. We propose that the embedded process-

ing elements in the IC will take care of the processing. The larger bandwidth and redirection to the system memory will provide more accurate and deeper traces without needing additional hardware resources.

Quinton et al. proposed programmable logic core enhancements [77] for SoC designs that enable DfD and correction of design errors after fabrication. Their work demonstrate that the programmable logic fabric can support direct interfacing with the fixed-function circuit. Our methodology does not explicitly require programmable logic to support hardware checker since they can be implemented as *fixed* functions in the logic. However, the result presented in Quinton's work are important as our proposed DfD is greatly enhanced by the availability of high-performance programmable logic structures for the more advanced techniques presented. For example, they allow the support for our proposed time-multiplexed assertion checkers.

Nicolici et al. [78] highlight the need for DfD hardware and offer a good overview of the research in that area. Ko and Nicolici [79] further explain how trace buffers can be used in combination with scan to capture internal states of the device. Some commercial techniques proposed by Veridae / Tektronix attempt to simplify this type of interfacing and on-device probing. Our work complement quite well those approaches. The sequence checker outputs are themselves amendable to be integrated in trace buffers. They effectively pre-compress the data by extracting meaningful higher-level events from the low-level and boolean logic layers. By pre-analyzing the low-level signals and extracting interesting sequence information, the automata of the checker effectively reduces the size of the data set that needs to be processed outside the device.

In a panel Vermeulen had the following recommendations [67] for future complex systems:

> Next-generation SoCs will contain (even) more programmable processors, a scalable communication infrastructure (e.g. a network-on-chip), and a multitude of dedicated, hard-wired functions. At run-time many software and hardware execution threads will be active simultaneously. As such, solutions are required for multithreading, multiprocessor, and communication debug that go beyond the capabilities

offered by DfD today. The existing DfD infrastructure will be extended towards the system level as an integrated hardware/software debug approach is required to debug, yet unknown, system issues from user application software to silicon.

2.4.2 Follow-up work on Time-multiplexing of Assertion Checkers

Our proposed approach of time-multiplexing assertion signals [13] proposed the integration methodology and algorithms to perform the integration, but left some unanswered questions as to how much benefit the method would bring to a real circuit. Gao and al. used the term *Time-Multiplexed Assertion Checking (TMAC)* [80] to qualify our proposed technique and applied it to an H.264 decoder to answer the following questions (adapted from their publication):

1. Given a number of assertions, how much can a silicon debugging process benefit from on-chip assertion checkers in terms of detecting more bugs and isolating bugs quicker?
2. What would be the area overhead of employing TMAC, especially considering the interface and interconnection between TMAC checkers and the design-under-checking? What is the growth trend of area overhead with respect to the population of assertions implemented in the TMAC manner?
3. How do we utilize the internal information captured by TMAC checkers to speed up the bug isolation process?
4. If the entire population of assertions is not affordable in silicon, even in the TMAC manner, what would be the assertion selection criteria be?

A summary of their results is presented here:

"The results of a case study demonstrated that, among those hard-to-detect bugs which cannot be detected by numerous standard H.264 video test cases, a TMAC of eighty assertion checkers detected 17.9% of the nonpropagated bugs and 39.4% of the non-detected bugs with only 1.3% area overhead. In the bug isolation phase, TMAC reduces the average bug detection latency by 87 times which leads to more than 500

> times a reduction in debugging data volume. Besides, with sufficient checkers in each module, the first assertion violation helps directly localize the faulty module. The benefits of TMAC on both bug detection improvement and bug root cause speedup could be more significant if more assertion checkers are implemented."

Those results are in line with our work with sample assertions showing how hardware checkers result in significant compression ratios in produced data volumes due to the temporal checking that is being performed by the automata in real time.

2.4.3 Design-for-Debug in Network-On-Chip

The complexity of designing efficient and scalable on-chip communication interconnects will continue to grow as increasing numbers of cores are integrated onto a single chip. A major challenge in chip design will be to provide a scalable and reliable communication mechanism that will ensure *correct* system behavior [36, 37]. Traditional SoC designs have used shared-medium, or bus-based, architectures whose limitations have now become apparent [81, 82]. In fact, for systems consisting of more than 20 cores, a bus interconnect quickly becomes the system bottleneck [83], degrading the performance such that it is no longer a feasible solution to the communication requirements. The key problem with bus-based approaches is in their limited *scalability*. To address the shortcomings of the shared-medium architecture, concepts from the domains of networking and parallel processing were adapted for on-chip use, giving rise to the idea of an on-chip communication network, or network-on-chip (NoC) [36, 37, 84].

As discussed previously, the increase in design complexity requires a change in the abstraction level in order to compensate for the limited engineering resources available to complete a given project. With new devices appearing on the market incorporating more than a billion transistors, the shift in design method becomes evident. At this point in time, it is possible to buy a processor for home use that integrates six processing cores sharing a common level of cache memory. A few years ago, this was limited to the realm of server and high-end systems and required multiple IC packages.

While the NoC concept addresses the shortcomings of shared-medium architectures, the vast NoC design space adds complexity to the design flow. Specifically, since the network topology will have a large impact on performance and device cost, the topology selection must now be included as part of the design space exploration and high-level prototyping stages. Typically, designers will start with high-level functional or *transaction level models* (TLM) for rapid prototyping and design space exploration. The high-level models are then refined until they can be synthesized to hardware. The traditional verification effort is usually left until the final stages of development, and is usually used to verify correct system behavior for a certain range of possible system inputs (functional coverage). Further, design for testability, another significant quality factor, is not integrated into the architectural exploration, where there is now a possibility to reuse the NoC as a test access mechanism [75, 85].

A few years ago, the microprocessor race was focused on internal clock rate. High-end microprocessor companies' marketing departments were hammering down the clock rate of their CPUs; the higher numbers usually winning major system deployments. Nowadays, the competition is much more subtle. The clock rate keeps increasing, but at a much lower pace. In 2000, Agarwal et al. [86] predicted the tapering off of the clock rate improvement and suggested the need for a change in the CPU microarchitectures that was observed in commercial microprocessors at around 2005. Always pushing a single core faster and faster resulted in CPUs using more and more power to meet their timing constraints. The power usage per instruction and the thermal dissipation envelope became a major problem in server environments and the shift towards slower clock rates and more power efficiency became the new industry direction. Maintaining the effective computing capability improvement for each generation, yet keeping the clock rate bounded, requires many computation cores on the same die. Instead of having one core pushed to its limits of thermal and process capabilities, modern scaling uses the newly available transistors to add more computing units on the same chip and process the data in parallel threads.

The addition of cores to a chip has the benefit that multiple parallel threads of execution can progress simultaneously. On the other hand, one has to keep in mind the complexity of the software. Therefore, hardware mechanisms to keep

the cache memories consistent among the multiple cores had to be implemented. They offer a unique memory view to the software, making their task a lot easier and keeping compatibility with legacy code. Legacy support is hugely important as enormous amounts of capital investment are placed in the software and it must be re-usable in newer computers.

However, this idea that one can scale the number of cores on a chip using shared cache memory cannot be sustained beyond a certain point. In a lot of cases, beyond 10 cores, additional processing elements will only bring marginal increases in computing benefits [87]. This is mainly due to the very large amount of cache coherency message passing that must be maintained on the bus to keep the memory consistent in all the core caches coupled with the increased physical distances and electrical load that appear due to the large interconnect.

A trend has thus emerged where instead of scaling the cores on a single bus, cores would be working with their local memories and a on-chip networking infrastructure would be created to exchange the information. Moving away from a shared bus to a set of routers effectively removes the effects of bus saturation and will keep the industry pace of computing capability increases going forward. NoCs effectively pipeline the communication transactions, thus increase the throughput of the communication channels. Since routing decisions are made without giving any core "ownership" of the bus, more transactions can process in parallel. Furthermore, the solution is very scalable.

However, this new NoC paradigm [35] is not without its share of problems. The network topology plays a key role in the final performance of the NoC. The optimal NoC topology will depend on the problem that must be solved. Several NoC topologies were studied [35, 88], but the lack of an accepted universal NoC benchmarking and the complex task required to appropriately model the NoC make this a challenging task to attempt. However, it can be shown that some topologies offer clear advantages in the context of either routing efficiency, redundancy in the communication paths (allowing the chip to be sold at a lower price, even with defective "sections" as opposed to be considered scrap).

Recent developments in benchmark standardization for NoC have started to converge [89] helping the community compare the various architectures in a uniform manner with real-world computational loading scenarios and standardized

workflows.

The OCP-IP group [90] discusses the standardization of on-chip debug interfaces such that integration of cores from many vendors can provide a uniform debug methodology for on-chip networks.

2.5 Chronological Work Overview

This section covers the chronological work performed throughout the years and explains the philosophy behind the proposed assertion-based DfD methodology presented in the next chapters.

2.5.1 NoC Research Work

The author's collaboration with Dr. S. Bourduas [91] on various topologies analyzed under various workloads allowed a better understanding of the need for high-level NoC characterisation and modeling. It is this collaboration and work on a low-level RTL implementation of a hierarchical-ring NoC interconnect [15, 17] that led to the observations that protocol checkers and performance monitoring circuits would have to be integrated with the assistance of EDA tools in order to maximize the productivity. A successful initial attempt at implementing a two-level hierarchical-ring NoC in VHDL using the Leon SPARC RISC core as a processing element leaded to a few key observations that later were used to direct this research.

The most important observations from this early research are covered in the next pages. The early conclusions from this initial work lead to the acquisition and use of hardware prototyping equipment and later the work on proposing the integration methods for temporal logic checkers in complex digital systems such as NoCs.

2.5.2 NoC Topology Consideration for Physical Implementation

The physical topology of the NoC has to consider the reality of an eventual physical implementation if one wishes to maintain efficient clock rates and good hardware utilisation.

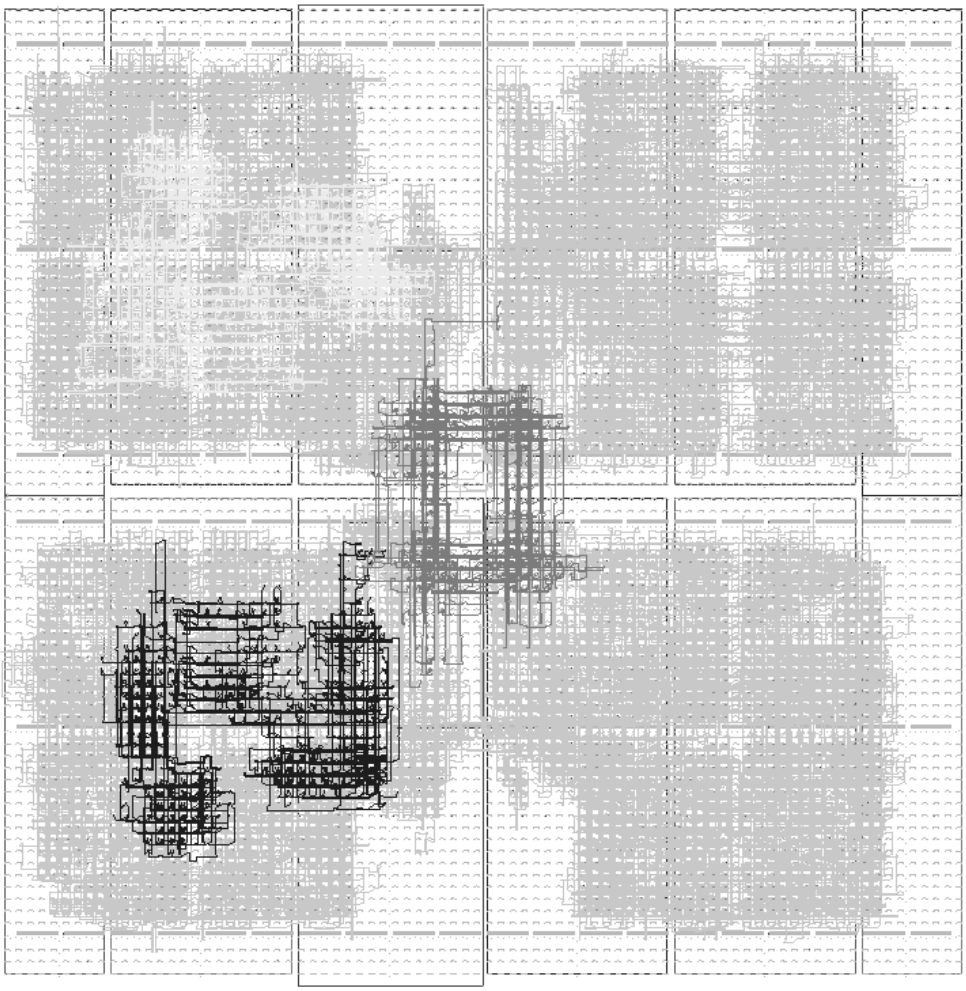


Figure 2.6: FPGA-based Network on chip and its routing localization and efficiency

Figure 2.6 shows the clean separation of quadrants in a hierarchical ring NoC (that has been slightly constrained in terms of floorplan) that was produced in an early RTL implementation of a two-level hierarchical NoC [15]. The result-

ing automated place and route highlight the localization of routing and the use of shorter routing segments, allowing faster clock rates to be achieved. Early in the NoC implementation project there was concern about some of the topologies and their suitability for prototyping and implementation. Any topology that requires some form of three dimensional structure creates a routing problem on a physical chip. The outside edges of the die have to be connected (e.g. in a Torus NoC) such that the data can flow from left to right directly. On the physical level, this means that some wires have to span the entire die (on higher levels of metals) to connect the outside edges. This poses problems in the routing strategy that will ultimately lead to lower overall performance because of the longer delays in those long lines. Adding metal layers in a device also leads to higher cost. In our attempt at implementing the NoC, we focused on a 2-level hierarchical ring topology (16 computing cores in total in 4 groups of 4 cores). This had the initial advantage of a clean floorplan on a large FPGA as seen in Figure 2.6. Furthermore, for an eventual prototype on high-end hardware [92], one has to consider the actual physical layout of programmable devices and communication channels.

Figure 2.7 shows the potential mapping of 16 cores (each having one of the DRAM module in the user FPGA) on the BEE2 [2] rapid prototyping architecture. The hierarchical ring topology would allow the global ring to span the interconnect between the four user FPGAs while each of the 4 local rings would map directly inside each of the user FPGAs.

2.5.3 The Need for Hardware-Based Monitoring Points

One of our early observations was that the dynamic behavior of a NoC is very important to its performance tuning and to locate congestion points. For example, the usefulness of dynamically monitoring the use of FIFOs at the system-level was a necessity to uncover bugs and improve the NoC system-level performance. To do this, we would need global access to the FIFO performance counters in a dynamic and transparent manner within the system. In our RTL simulator, the example graphs were generated from plotting the analog representation of a given set of hardware bits (e.g. FIFO utilization). Figure 2.8 shows the typical dynamic behavior the buffer usage and other hardware signals in our NoC RTL implementation.

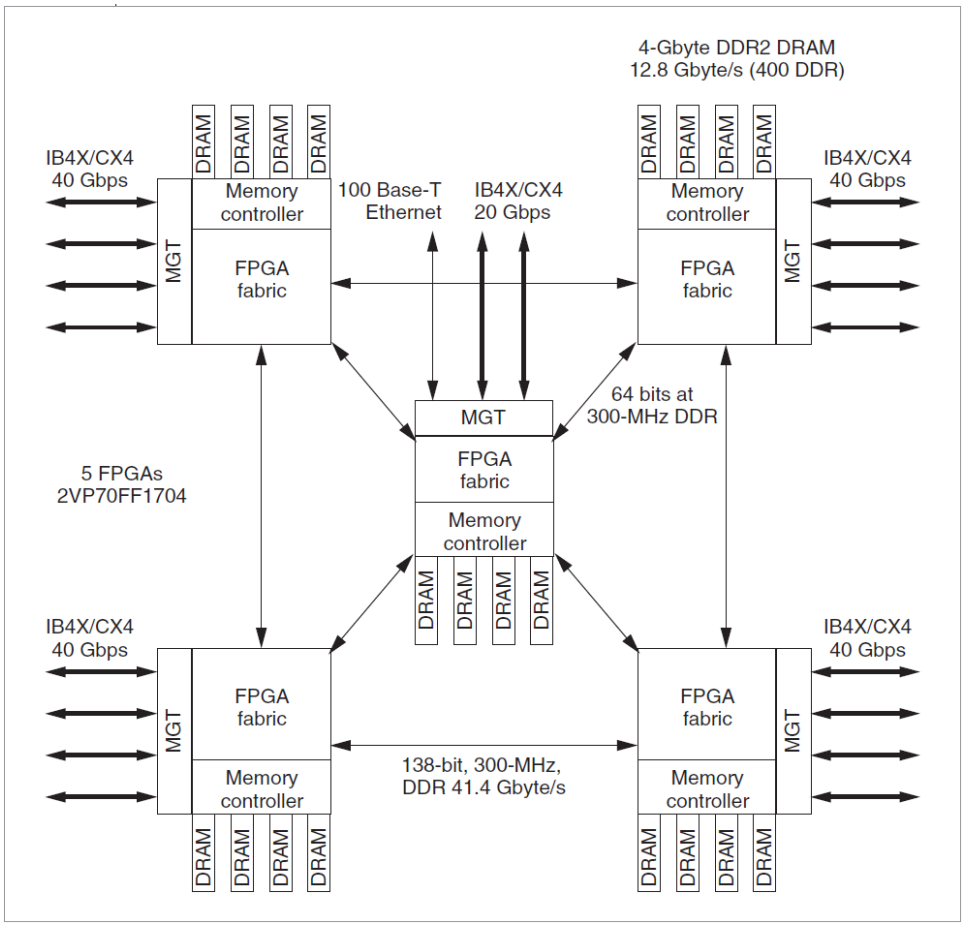


Figure 2.7: BEE2 System-level block diagram from Chang et al. (2)

At this point, it was noted that a lot of the bus transaction signals and performance counters that were so easily accessed in a simulation tool (ModelSim from Mentor Graphics) would disappear from view in the final hardware implementation. The actual registers (flip-flops) would still be present on the chip, but their visibility would be gone from within the system. This led to a search for a way to add those debug and monitoring points such that those registers can be mapped as part of the coverage and performance monitoring counters and reported to the software via an internal interface covered in Chapter 4, thus making live monitoring of the system possible.

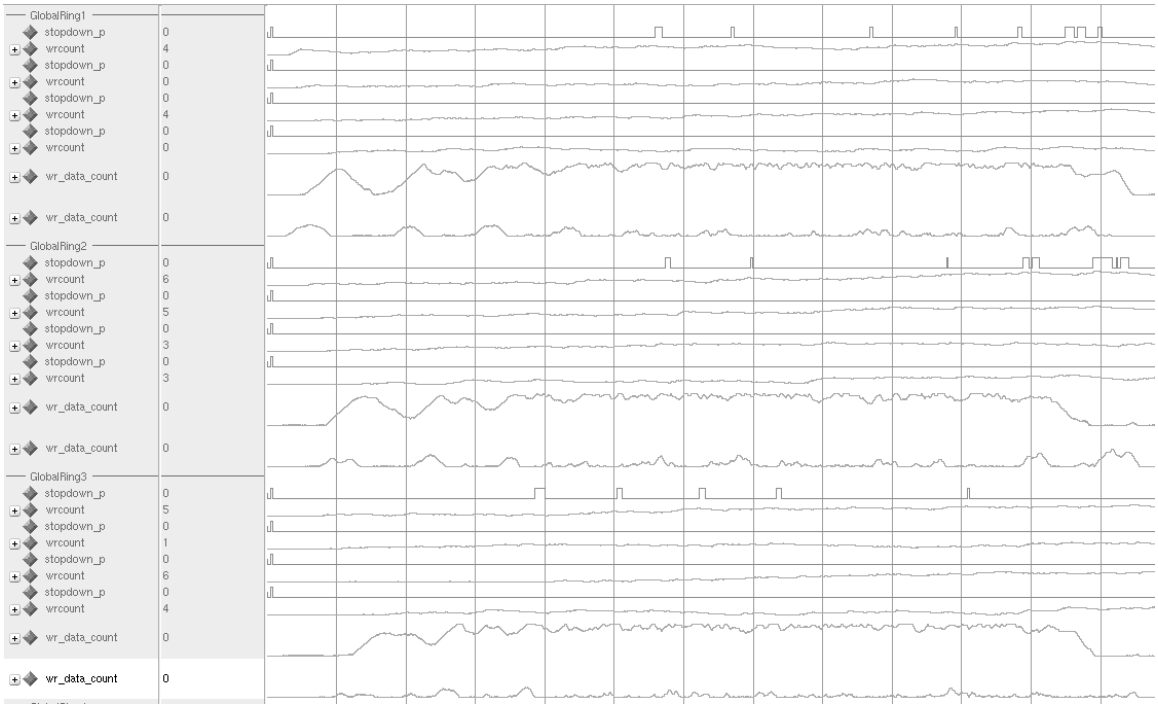


Figure 2.8: Modelsim simulation of FIFO occupation during heavy NoC traffic

2.5.4 The Difficulty of Integrating Large Systems

Once the RTL of the NoC routing units were debugged and ready for hardware implementation and the simulation models showed that we had properly integrated the CPU cores (Leon II from Gaisler Research), the research took a turn in a different direction as a result of the following issues:

A: The core interconnect was extensively simulated, but the simulations became extremely long when integrating more than a few CPU cores. Our 16-core model could only execute a few thousand instructions per second due to the very heavy load of simulating the CPU model. This slow execution of the full NoC model was unamendable to extract proper system-level performance as so few transactions per second were generated by the CPU models. This led to the research by S. Bourduas [91] to re-model the interconnect in SystemC and use instruction set simulator models instead of the RTL level model of the CPU. The use of data from the RTL model made the SystemC model very accurate and

representative of the hardware performance.

B: We investigated the use of the BEE2 hardware platform as an accelerator for simulation performance and to host the NoC system. However, logistical problems due to the complexity of the task quickly came up. One of them was the very long placement and routing time to realize the NoC on the very large user FPGAs present in the BEE2 (tens of hours of placement and routing for each FPGA). Adding the time to debug each FPGA made it a prohibitively time-consuming task. The second problem was to map the physical links to the hardware pins of the FPGA which would have been a considerable endeavour. Large parallel buses were needed to carry the NoC traffic and 8 links had to be defined with hundreds of pins in each direction. High-speed serial links were considered, but IP licensing constraints and added latency on the interconnect were too complex to rapidly resolve. While the realization of the physical NoC would have been a very interesting proof-of-concept technically, it would have consumed a lot of time and would not have provided so much research benefits. On the other hand, the attempt highlighted a dire need for a way to rapidly export internal registers to a centralized location for ease of debugging.

C: The BEE2 came with its own operating system aimed at performing digital signal processing (BORPH) [93]. However, this operating system was based on the Linux 2.4 kernel and did not include any of the advanced features of the Linux 2.6 kernel, notably the UIO driver (whose usefulness in this work is detailed in Section 4.6). As part of this research a lot of effort was dedicated to re-creating the Control FPGA of the BEE2 for use with open source IP cores (UART, memory controller), porting the U-Boot [5], the hardware drivers, the Linux Kernel 2.6 and creating a root file system image for the BEE2 hardware such that the newer user space IO (UIO) drivers could be tried as a proof-of-concept for userspace hardware assisted debugging (covered in Section 4.6). A set of application notes [18, 19, 20] and software / IP core libraries were

5. http://www.denx.de/wiki/U-Boot

documented and provided to the Canadian Microelectronics Corporation as part of this effort.

D: A large problem remained: how could we get the protocol analyzers and performance counters in the hardware without hand-coding each and every one? The solution came from extensive collaboration with Marc Boulé. The concept that temporal logic would allow the verification effort to be re-used in hardware meant that protocol checkers and sequence monitors could be generated automatically. Performance monitors required some modifications to the tool's algorithms, but this meant that statements that were used in verification could be translated to hardware. Combining this with automatic register generation and operating system integration meant that a tool flow was taking care of the tedious hand coding. This motivated most of the work presented in Chapter 4.

Chapter 3

Checkers as Dynamic Assistants to Silicon Debug

As explained earlier, assertion checkers have recently been added to the verification process toolbox that contribute significantly to enhanced productivity. Using assertions increases the design effort since it forces the designers to formalise their specifications, but as this thesis will show, they can bring major benefits in the silicon verification and debugging processes if they are carried along with the design as hardware checkers.

As *assertions* and *sequence* statements provide a very consise and formal way of specifiying design behavior and interface timing requirements, it makes sense to use them to express hardware checkers rather than coding those checkers from scratch in RTL. In doing so and using automated generation of hardware, the designer's effort to carry the benefits of the checkers into the silicon will be greatly reduced.

The ABV methodology defines a device (or module) behavior as a set of properties. Those properties capture the essence of the temporal and logic behavior without the ambiguity of natural languages.

As can be seen in Figure 3.1, the proposed hardware checkers with debug enhancements can be used in multiple scenarios, ranging from hardware emulation to in-field deployment of the final device. This chapter will highlight key contributions to hardware assertion checkers and their applications in the debug of silicon devices. Then a mechanism to partition a large set of assertions into smaller

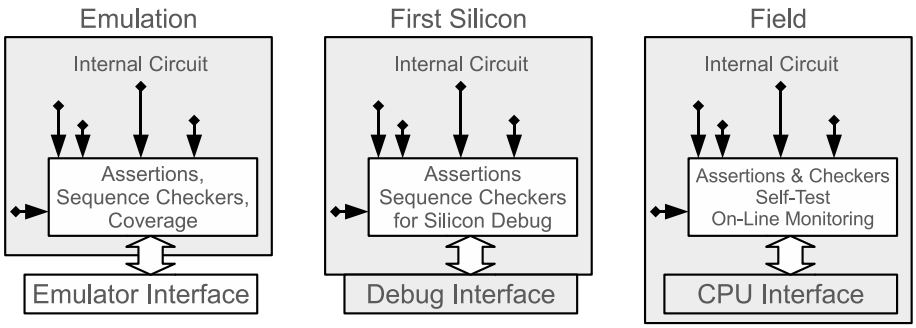


Figure 3.1: Usage scenarios for hardware assertion checkers.

sub-groups that can be dynamically re-programmed in a section of an ASIC is presented. The partitioning algorithm is detailed along with the proposed method of integration. Finally, the experimental results of adding debug enhancements to assertion checkers are covered.

3.1 Benefits to Designers

As mentioned earlier in the background chapter, the designers reap the benefits of assertions when they have to tackle a complex debugging scenario. Aided with the simulator support from the various assertion checkers, fired assertions will pinpoint the moment in the simulation that a problem started to show its symptoms. Since assertions are layered and monitor very low-level hardware timing, they will typically *fire* very quickly after the circuit deviates from the specifications. By analyzing the assertion failure(s), designers may use elimination or root-fault localization tools [94] to locate the source of the bug.

Assertions can also be used in hardware emulation or simulation accelerators to obtain similar benefits; however, until recently, descriptions in high-level property specification languages were not easily converted into efficient RTL descriptions, suitable for hardware emulation platforms. To exploit the power of assertions in a hardware context, our research group and most notably the work of M. Boulé [95] contributed to this research effort to offer efficient conversion from temporal logic expressions to hardware checker circuits. These checkers monitor the

physical DUV for violations of assertions and raise an output signal each time a violation is observed. Circuit-level assertion checkers can then be used, not only for pre-fabrication verification, but also for post-fabrication silicon debugging if one leaves the checkers in the final silicon.

Assertions can clearly benefit the designer when a bug is found in the simulation. If the assertion density is sufficient, the erroneous condition will be detected and an assertion will *fire*. Internal *virtual* signals can be used to monitor the assertion and allow the engineer to quickly pinpoint the failure and trace it back to the design [7]

As assertions are written and used by the designers to help them debug their simulations [1], they build up to form a substantial body of knowledge. As designers continuously add to this library of checkers and complex sequences, it becomes valuable reference since it contains most of the assumptions, limitations and temporal behavior of the design's hardware blocks. Naturally, one would want to re-use this implicit design knowledge in the final silicon, ideally without expending too much effort. Low-level sequences present in the assertion library represent very valuable abstracted states of the design and sometimes are directly applicable as performance counters or status monitors. For example, every time the following sequences is seen in a NoC: {**worm_start**; **data**; **worm_end**}; the meaning of that sequence is clear from a hardware point of view. It represents a worm (a packet in the context of a NoC) traversing that particular bus. Counting the occurences of this sequence turns it into a valuable performance counter. That performance counter would typically need to be coded *manually*. With our proposed approach, engineers can dig into their assertion library and pick the ones that can effortlessly provide system-level performance monitors, as required by the application.

The MBAC tool, developed at McGill University by our group, attempts to address the problem of specifically converting PSL or SystemVerilog assertions into *compact* and *efficient* hardware checkers. The book published by M. Boulé and Z. Zilic [96] cover in details the transformation of PSL statements into a hardware circuit. The work contributions presented in this chapter aim at expanding this tool to support a wider range of uses such as the automatic creation of performance

1. From information gathered from colleagues and alumni working at AMD and Intel, the assertion librairies represent up to tens of thousands of statements

monitors that provide aids to debugging. That way, the powerful hardware-based temporal checkers do not stay confined to support only accelerated verification.

In this emerging DfD space, several companies were quick to see the opportunity and are promoting a range of solutions to address the debug problem. Tools from companies such as Novas now support advanced debugging methods to help find the root cause(s) of errors by back-tracing assertion failures in the RTL code [97]. Temento's DiaLite product accepts assertions (to a limited extent) and provides in-circuit debugging features. DAFCA also offers this possibility, and provides support for assertion checker synthesis and use. However, as these tools are from commercial ventures, very few papers cover their inner-workings. In most cases one has to gather information from those companies' marketing material. Futhermore, the MBAC tool outperforms other available industrial solutions available [11, 62] and our group benefits from the fact that it was developed internally, therefore amenable to modifications.

Increasing and enhancing the visibility into the design's temporal state transitions is an important aspect in silicon debugging and DFD [98]. Increasing visibility using ABV techniques were explored [11]. From this exploration this thesis proposes the following enhancements: **Antecedent Monitoring**, **Assertion Dependency Graph**, **Assertion Activity and Coverage**, **Assertion Completion Mode**, and **Assertion Threading**. These specific enhancements improve the debugging capabilities of the resulting checkers in many scenarios of silicon debugging. All debugging enhancements are implemented as a part of the checker generator, where they can be optionally activated by the use of command-line switches when the tool is run. The added visibility into parts of the assertion checker circuits, along with additional ways to track assertion results, are the principal means by which this thesis proposes to improve assertion-based debugging of physical devices.

The debug enhancements will prove more valuable as IP cores evolve in complexity. Even if the blocks are sold as *black boxes* by the IP vendors, they still require their interface to be driven according to specifications. Thus, they are very likely to be sold with an accompanying *assertion library* readily usable in simulation. Upon integration in, say, a large FPGA, some of those assertions can be selected and transformed in hardware, effectively providing a continuous check on the proper integration of the block and an assistance to debug when things go wrong.

3.2 Assertion Checkers Enhancements for In-Silicon Debugging

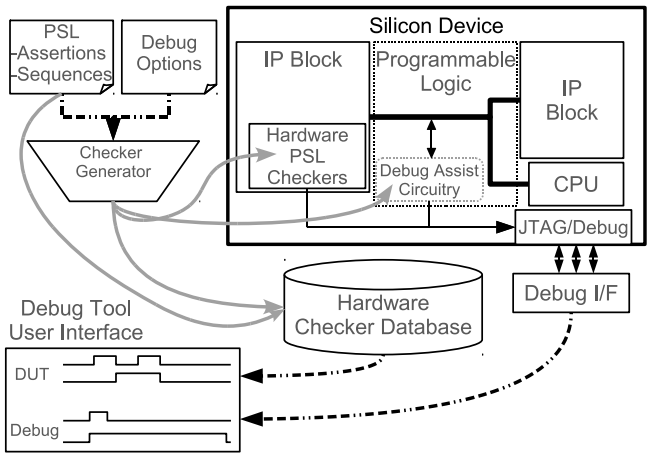


Figure 3.2: Hardware PSL checker within a JTAG-based debugging enhancements

This section presents the enhancements that can be optionally added to the assertion checkers produced by the checker generator. These enhancements increase the visibility within assertion circuits, and also enhances the coverage information provided by the checkers. Figure 3.2 illustrates how the checker additions intervene in the methodology. The checker generator produces monitoring circuits from PSL statements augmented with various debug-assist circuitry. Other forms of debug information, such as signal dependencies, can also be sent to the front-end applications. Since the techniques are implemented at the RTL level within the checkers, they can be used in concert with any other circuit debugging tools including scan chain access such as JTAG, trace buffers with timestamp mechanisms or with the proposed register-based mechanisms presented in the next chapter.

3.2.1 Antecedent and Activity Monitoring

The debugging capabilities introduced are in the assertion domain, in which the assertion's behaviour itself is further explored to locate the source of a problem. The approach involves augmenting the checker generation algorithms to in-

strument the checkers in ways that help to locate the root causes of assertion and circuit failures.

Example 1 illustrate a simple assertion that may require a fair amount of investigation to deduce the cause of failure if it ever fires. By enhancing the assertion checker in hardware, one can break the sequence complexity and export valuable debug information. It states that whenever the arbiter is ready and receives a bus request, then the grant signal should be low in that cycle (the temporal implication |–> is overlapping). The grant should then be given within at most 5 cycles; furthermore, the arbiter's busy signal must be true until the grant is given, when it must then be low.

Example 1 A Typical Bus Arbitration Assertion Statement.

assert always { REQ & READY } |–> {∼GNT ; {BUSY & ∼GNT}[∗0:4] ; GNT & ∼BUSY};

Knowing only that this assertion failed will not reveal the exact cause, or even the sequence of events responsible for the assertion failure. For example, if REQ, READY and GNT are all asserted simultaneously, this will be a failure as much as if the GNT was never asserted. If a tool can provide the explicit knowledge about the antecedent's status (in this case "REQ & READY"), this avoids having to manually create new signals in the debug environment or chase two possible sources for a given assertion failure in silicon. In our simple example, the antecedent signal can be easily created manually; however since PSL allows having an arbitrarily complex sequence as an antecedent (one which can also be a suffix implication), some statements would be difficult to re-create in the debug environment and even more difficult to observe in physical hardware. On a physical device, by tapping into the hardware signals that trigger the automata (antecedent completion triggering the consequent), a monitoring point can be exported that will help the debug engineer directly "see" that event in the assertion checker. This information can then be exported to a register bank for in-system monitoring as covered in Chapter 4.

As one can observe, the simple REQ & READY antecedent can be quite useful to detect the *start* of the sequence check. Simulators such as ModelSim [2] provide this

2. A well-known commercial hardware language simulator sold by Mentor Graphics

kind of information by annotating the waveforms during the simulation. Using the proposed method of exporting sequence activity, this valuable debug information can now be carried to the silicon. The example presented is simplified to ease its presentation. However, one can build very complex and intricate temporal expressions using PSL operators, yet the sequence activity will still be as simple to observe with a minimal effort on the part of the designer.

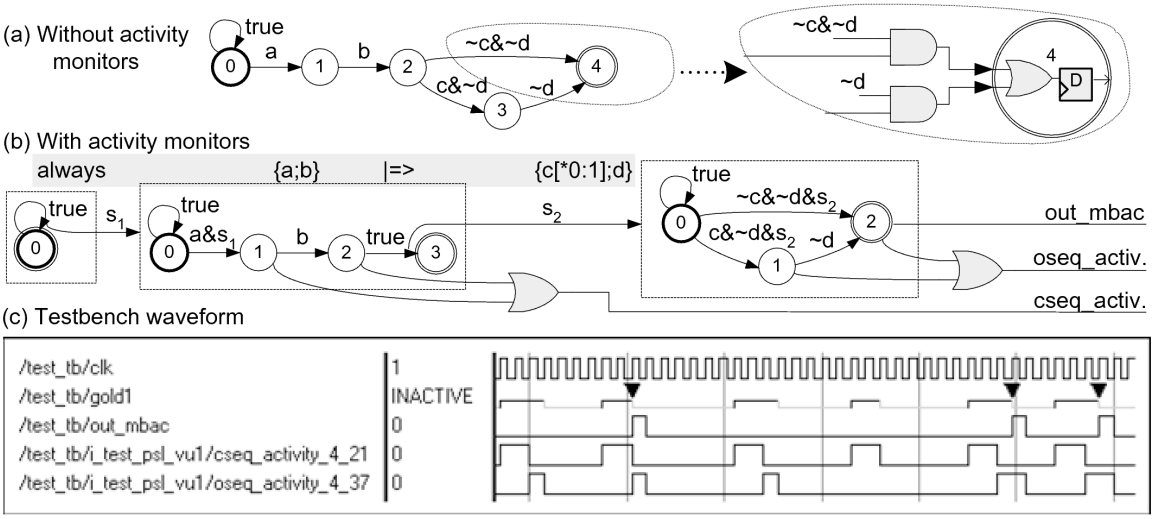


Figure 3.3: Activity signals for property: ***always*** ({a;b} |=> {c(*0:1);d}). oseq corresponds to the right-side sequence, cseq to the left-side sequence.

Figure 3.3 shows the mechanisms by which the presence of *tokens* in the checker's automata is exported to the circuit as a way to indicate activity in the checker. Activity monitoring would mainly be used in conjuction with an internal logic analyzer allowing the visibility in the hardware where the sequence is being actively checked. It can be used, for example, to trigger the storage of data in the logic analyzer, capturing only interesting activity sequences and to make better use of the limited buffer memory.

3.2.2 Assertion Dependency Graphs

When debugging failed assertions, it is useful to quickly determine which signals and parameters can influence the assertion output. In the toolset, all of the

signal and parameter dependencies are listed as annotations for each assertion circuit in the database.

Taking Example 1 as a reference, the direct dependencies are REQ, READY, BUSY and GNT. In a given system, the READY may not be directly an input signal to the hardware checker, but could be derived at the *Boolean layer* from a combination of signals or a decoded status bus. In such cases, the dependency graph would trace back to the actual signals composing the READY part of the assertion statement, simplifying the task of defining probes or waveforms to log as part of the debug process.

The dependency graph is constructed by the tool to help pinpoint the cause of an error, or for automatic wave script generation in an emulation environment. When an assertion fails, the signals that are referenced in an assertion can then be searched and automatically added to a wave window and/or extracted from an emulator, in order to provide the necessary visibility for debugging. Dependency graphs are particularly useful when complex assertions fail, especially when an assertion references *other sequences* and/or properties, as allowed by PSL [99]. In such cases, assertion signal dependencies will help ensure that all the signals on which a checker depends are included in the analysis. Tracing back from the assertion output through its related cone of logic can directly provide cues as the causes of the assertion failure without requiring the person doing the debugging to *know* the circuit beforehand.

This feature prevents the problem of incomplete probe set in an emulation or FPGA system during debug. In a typical debug scenario, a set of probes are added to the FPGA or emulator to add visibility for the signals influencing the checker failure. Those signals are needed to find the cause of the problem. One important issue is the *time* needed to re-compile the FPGA with the added probes. For large FPGAs, this can mean hours of waiting while the probes get added to the design (requiring re-routing of parts of the FPGA), the preparation of the bitstream and the setup of the test case. This is why an error-proof method of adding checker dependencies is required to avoid having to repeat this long and tedious process. Recent work in the field [100] tends to focus notably on software solutions addressing the problem of debugging the assertions along with the circuit. Many of the presented techniques could be used to enhance our more *hardware-oriented*

approach.

3.2.3 Assertion Completion Mode

For a verification scenario to be meaningful, assertions must be exercised: assertions that do not trigger because the test vectors did not fully exercise them are not very useful for verification or debug. In such cases, when the assertions are trivially true the designers could be led to believe that the property has been validated, and thus overlook the true cause of a non-failure. On the contrary, assertions that are extensively exercised but never trigger offer more assurance that the design is operating as specified. The dependency graphs from the previous section efficiently determine which signals must be stimulated to properly exercise an assertion that is found to be trivially true.

The hardware checker generator was thus enhanced such that assertions can be alternatively compiled in a *completion mode*, to indicate when assertions complete successfully and are not trivially true. The completion mode affects assertions that place obligations on certain sub-expressions, such as the consequent of temporal implications for example. In temporal implications, for each observed antecedent, the consequent must occur or else the assertion will fail. As opposed to indicating the first failure in each consequent, as is usually done, the completion mode assertion indicates the first success in the consequent, for each activation coming from the antecedent.

The completion mode has no effect on assertions such as "***assert never*** *seq*", given that no obligations are placed on any Boolean expressions. This assertion states that the sequence argument *seq* should not be matched in any cycle. Thus, every time the sequence is matched (i.e., is detected as occurring), a violation occurs and the assertion output triggers. The actual PSL syntactical elements which are affected by the completion mode are sequences and Boolean expressions when used directly in properties, with the following exceptions: the argument of the ***never*** operator, and the antecedent of implications (suffix implication and property implication).

Using terminology in [101], our technique identifies interesting witnesses, i.e., examples of where the property was exercised. Antecedent non-occurrence – com-

monly referred to as vacuity – is only one possible cause for trivial validity. The knowledge that an assertion completes successfully can be useful when evaluating the coverage quality of a regression suite. Completion mode creates behaviour analogous to the ***cover*** operator for sequences, except it is at the property level. Completion mode can also be referred to as "pass checking", "success checking" and "property coverage".

The completion algorithm, illustrated with an example follows. The completion mode transformation algorithm first determinizes the automaton such that each activation is represented by only one active state. From any given state, a deterministic automaton transitions into at most one successor state. Determinizing automata with Boolean expressions on transitions is more involved than in conventional automata [102]. The determinization step is required so that when the first completion is identified, no other subsequent completions will be reported for the same activation.

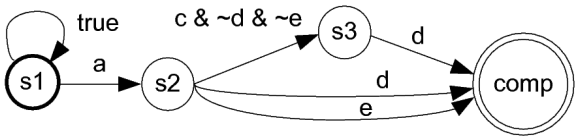


Figure 3.4: Completion automaton for ***always*** ({a} |=> {{c(*0:1);d}|{e}}).

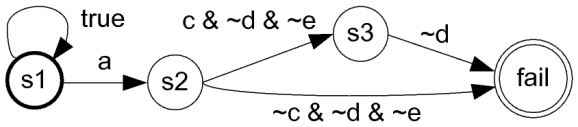


Figure 3.5: Normal automaton for ***always*** ({a} |=> {{c(*0:1);d}|{e}}).

The second step in the completion mode algorithm is to remove all outgoing edges of the final states, when applicable (in Figure 3.4, there were no such edges to remove). Any unconnected states resulting from this step are removed during automata minimization. Both the failure and completion transformation algorithms take as input the detection automaton that corresponds to the sequence being handled. The completion-mode algorithm can also be used to implement the SVA operator **first_match()**, and is a dual of the *FirstFail* algorithm [103]. The *FirstFail* algorithm is normally applied to Booleans and sequences that are used

directly as properties. In completion mode in the checker generator, all calls to the *FirstFail* algorithm are replaced by calls to the *FirstMatch* algorithm (also called the completion mode algorithm). Assertion completion is best visualized using an example.

Example 2 Test Assertion for Assertion Completion.

assert always ({a} |=> {{c[*0:1];d}|{e}});

The assertion above is normally compiled as the automaton in Figure 3.5, where the final state is triggered when the assertion fails. The completion mode automaton for this example is shown in Figure 3.4. The sequence of events $a;c;d$, for example, will make the automaton in Figure 3.4 trigger (completion); however, the failure automaton will not reach a final state given that the sequence conforms to the specification indicated by the assertion. In the automata graphs, the highlighted state s1 indicates the initial state, which is the only active state when reset is released. The PSL ***abort*** operator has the effect of resetting a portion of the checker circuitry [104], and thus applies equally to the normal mode or completion mode.

3.2.4 Assertion Activity and Coverage

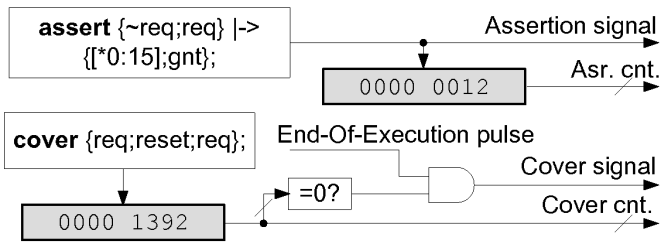


Figure 3.6: Counting assertions and cover statements.

Automatically creating counters on ***assert*** and ***cover*** statements offers a rapid way to transpose coverage information to hardware devices. Counting assertion failures is straightforward, as observed in the top half of Figure 3.6; however, counting the cover directive requires some modifications. In dynamic verification, cover is a liveness property which triggers only at the end of execution. In order to count occurrences for coverage metrics, a plain matching (detection) automaton

is built for the sequence argument [3], and a counter is used to count the number of times the sequence is matched. The cover signal only triggers at the end of execution if the counter is at zero, as shown in the lower half of Figure 3.6. If no counters are desired, a one-bit counter is implicitly used. The counters are width parameterized, and by saturation arithmetic, do not roll-over when the maximal count is reached. The counters are also initialized by a reset of the assertion checker circuit or by a control register as explained in Section 4.4.

As will be covered in Chapter 4, a more granular control on the counters is provided by the hardware interface generator such that each counter in a group can be individually controlled. This thus allows to keep the device state (i.e. not force a device-wide reset), yet restart specific coverage counters which will become particularly useful during the debug process as a way to isolate a problem.

Counters can be used with completion mode (Section 3.2.3) to construct more detailed coverage metrics for a given scenario. Knowing how many times an assertion completed successfully can be just as useful as knowing how many times an assertion failed during debugging. For example, if a predetermined number of bus transactions is initiated, the related assertion should see itself complete successfully the same number of times. In general, by signaling successful witnesses, completion mode provides an indication that if an assertion never failed, it was not because of a lack of proper stimulus.

3.2.5 Hardware Assertion Threading

Assertion threading is a technique by which the checker generator instantiates multiple copies of a sequence checking circuit, and alternately activates these circuits. This allows a violation condition to be separated from the other concurrent activations in the assertion circuit. This helps visualize the exact start condition that caused a failure. In general, by using a single automaton-based recognizer, all temporal checks become intertwined in the automaton during processing. The advantage is that a single automaton can catch all failures and it reduces the hardware overhead of the checker. However the disadvantage is that it becomes difficult to correlate a given failure with its input conditions. During the debugging phases

3. In PSL, a property is not a valid argument for the ***cover*** operator.

of a design, one may want a more direct localization of the failure. The assertion threading in effect separates the concurrent activity to help identify the root cause of the sequence of events leading to an assertion failure. Threading applies to PSL sequences, which are the typical means for specifying complex temporal chains of events.

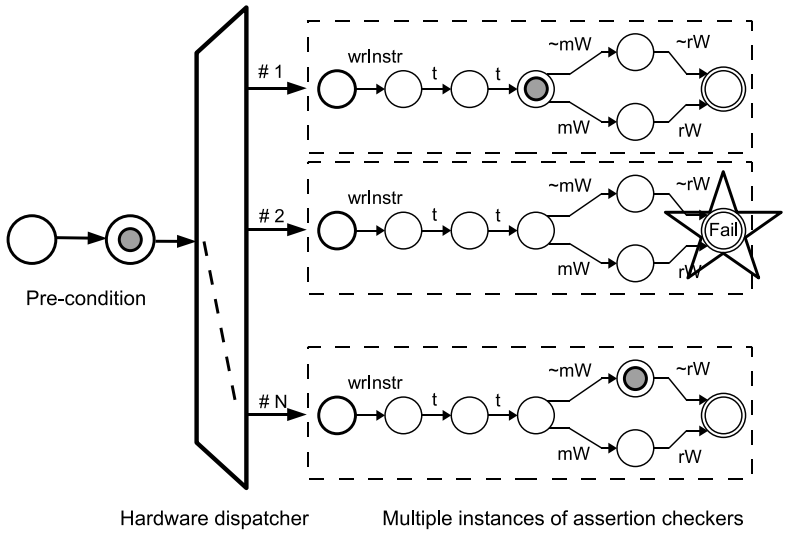


Figure 3.7: Hardware assertion threading

Figure 3.7 illustrates the mechanisms used to implement assertion threading. The hardware dispatcher redirects the activation signal to the multiple sequence-checker units in a round robin sequence. The tokens indicate the progress through the sequence automata. In the figure, hardware thread #2 has identified a failure. With this information, one can trace back to the antecedent expression that initiated the sequence checking in thread #2. An example will follow illustrating how this method can be used in tracing back an execution error in a CPU.

In assertion threading, entire failure-matching sequence-automata are replicated in hardware. Since a single automaton can detect all sequence failures, replicating the automaton and sending tokens into different copies still ensures that no failure is missed *even if the number of threads is below the concurrency level (depth) of the monitored sequence*. The dispatcher rotates a one-hot encoded register such that each activation is sent to one of the hardware threads. If a token enters a thread for which a previous token is still being processed, identifying the precise

cause of a failure becomes more difficult, but still feasible. In such cases, increasing the number of hardware threads can assist in the process of properly isolating the faulty sequence. Evidently, increasing the hardware resources to assist in the debugging is only possible in systems where programmable logic is present such as FPGA-based accelerators or in devices that integrate re-programmable logic fabric as well as fixed silicon, application-specific resources.

Threading also applies to the plain matching sequence automata (as opposed to the failure matching automaton discussed above). In such cases, the plain occurrence matching automaton is threaded for increased causality visualization. The nuance between plain matching and failure-matching modes (called *conditional* and *obligation* modes [103]) can be observed by comparing the automata for both sequences appearing in the assertion in Figure 3.3 (b). In this example, the occurrence and failure modes correspond to the left and right sides of the |=> operator, respectively.

To complete the threading, the sequence output is defined as the disjunction of the threaded automata outputs. Seen from the sub-circuit boundary, a multi-threaded sub-circuit's behaviour is identical to that of a non-threaded sub-circuit. Threading applies to any PSL property in which one or more sequences appear. Threading of a simple Boolean expression used at the property level is obviously not performed. A tradeoff is required between an accurate location of the source of the failure and hardware resources usage, as will be shown in Section 3.4.

An example scenario where assertion threading is useful is in the verification of highly pipelined circuits such as a CPU pipeline or a packet processor, where temporally complex sequences are modeled by assertions. In such cases, it is desirable to partition sequences into different threads in order to separate a failure sequence from other sequences. Once the sequence processing is temporally isolated, the exact cause of the failure can be more easily identified. The following case study shows how assertion threading can be used to quickly identify incorrect instruction executions in a CPU.

3.2.5.1 Assertion Threading – CPU Execution Pipeline Debug Scenario

A simplified CPU execution pipeline, similar to the DLX [105] RISC CPU with 5 levels of pipeline, was coded in RTL and two classes of instructions were considered, namely memory writes and register writes. This CPU is used to execute instructions that contain memory and register manipulation. An error injection mechanism is also incorporated into the instruction decoder, such that errors can be inserted in the execution pipeline. Memory writes are committed at the 4th level (MEM Stage) in the pipeline, and register writes are committed at the 5th level (WB Stage) in the pipeline. For a given WRITE instruction only a single destination is allowed by the architecture (Memory or Register).

Example 3 shows the PSL code used to create an assertion checker circuit for monitoring the memory write or register write instructions. The sequences *Smemwr* and *Sregwr* are built to ensure that a write is either to the register file or to external memory. In this CPU pipeline, those temporal expressions represent the same "store" instruction at two different pipeline stages. The sequence *Swr_instr* models the instruction decoder detecting the presence of a write instruction. The property *Pcorrect_wr* ensures that this write instruction will either result in a memory write or a register update (but never both for the same pipelined instruction execution cycle).

Example 3 Assertion checker built from sequences to illustrate a pipelined CPU write instruction.

```
default clock          =  ( posedge Clk );
// Memory write sequence
sequence Smemwr        =  {[*2] ; MemWr ; ~RegWr};
// Register write sequence
sequence Sregwr        =  {[*2] ; ~MemWr ; RegWr};
// Write Instruction
sequence Swr_instr     =  {InstrValid && Instruction[31]==1'b1 &&
                           (Instruction[30:29]==2'b10 || Instruction[30:29]==2'b01) };
// Write works ?
property Pcorrect_wr   =  always { Swr_instr } |=> { {Smemwr} | {Sregwr} };
assert Pcorrect_wr;
```

The above PSL code is given to the assertion and sequence compiler along with the CPU RTL code in Verilog. The resulting checker is instantiated in the CPU architecture. The CPU along with its checker are exercised by a testbench running

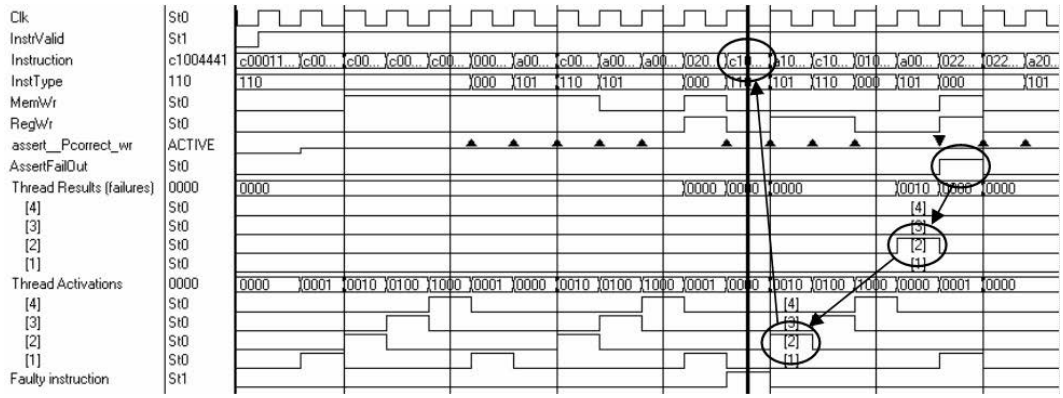


Figure 3.8: Using the assertion threading method to efficiently locate the cause of an instruction execution error in the CPU pipeline example.

various verification scenarios.

Figure 3.8 shows the resulting simulation trace. The dependency graph is used to determine the list of signals that relate to the assertion being debugged, and by extension the signals that need to be logged in the wave window. The AssertFailOut signal is asserted at a given time point, indicating a violation in the correctness of the write instruction behaviour. In this example, the instruction was committed to both the memory and the register file, which is impossible in this architecture. Tracing back through the Thread-Results vector, it is found that thread #2 has detected the failure. Working back through the activations of this thread, it can be observed that the instruction causing the error is highlighted by the cursor in the Figure. The assertion threading helps to isolate the source of the faulty sequence, and allows us to quickly determine which specific instruction was responsible for the assertion failure. In our example, for the sake of simplicity, the CPU executes one instruction per clock. In more complex problems, some instructions could take a variable number of cycles to execute; assertion threading would become an even more important asset to help debug these types of circuits.

3.3 Temporal Multiplexing of Checkers

Hardware assertion checkers can be incorporated as a part of the final silicon as dedicated checkers that continuously monitor the circuit for abnormal condi-

tions. In a typical ASIC, some of the IP cores are known to be quite robust from previous use or because they are provided by a third party vendor with multiple previous successful tape-outs. Therefore, a balance between risk mitigation and on-chip assertion capabilities has to be calculated. Programmable-logic fabric or reconfigurable elements are increasingly inserted into ASICs to allow corrections of silicon bugs or to bypass faulty modules. Since this reconfigurable fabric should be unused at the initial tape-out, it represents an excellent opportunity to include assertion monitors with no cost impact.

As a way to increase the number of assertions that can be monitored under area constraints, this thesis proposes to time-multiplex groups of assertion checkers. By continuously re-programming the reconfigurable fabric, it is thus possible to support a much larger assertion coverage.

Figure 3.9 shows different levels of core confidence, as could be encountered in a typical System-on-Chip (SoC) design, for example. Core1 could have been used in a previous design, thus the confidence is high and a limited number of connectivity points are shared with the programmable fabric. Core3 could be a new design and thus being more risky, more re-programmable resources are dedicated to potential bug fixing, while additional checkers can be built into the silicon as extra precaution and to assist post-silicon debug.

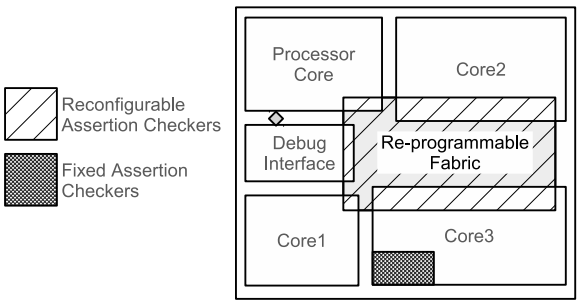


Figure 3.9: Typical SoC floorplan implementing fixed and reprogrammable assertion checkers.

The assertion checkers benefit from observability on the main system buses for protocol checking and assertion-based debugging enhancements. Before tape-out, an analysis of each checker circuit is performed and the routing overhead is esti-

mated based on each assertion's input dependencies. Once the design is locked with a specified list of available monitoring points, the tool can provide the designer with all the assertion checkers that will be supported by the future ASIC. New assertion checkers can be generated after tape-out as long as they respect the silicon area constraints and primary input requirements. In this scenario, the microprocessor can coordinate the instantiation of the proper checkers for each test sequence in the reprogrammable fabric. Checker groups (also called partitions, or subsets) are instantiated one after the other in the reconfigurable area, to correspond with the set of test sequences being run.

This process of in-system re-programming of logic cells is called Partial Dynamic Reconfiguration or Run-Time Reconfiguration and is well understood. Platzner et al. have a great number of publications related to this area [106]. Most of the research is done on FPGAs since modern devices support the methodology. An good example of the method and applications outside the scope of assertion checkers can be seen in the work of Abel et al [107].

As a final interesting use of re-programmable checkers is their use as a flexible *trigger mechanism* for storage buffers. Since they implicitly support complex temporal expressions, the output of a sequence checker can be used to enable the capture of a given bus transaction in a buffer. A complex triggering problem may be put forward like in this example: "Capture the data bus values during cycles 3,4,5 of a delayed-grant burst transfer initiated by device A". In this case, the completion mode logic of a sequence checker can be used with a precondition given by the detection of a delayed-grant burst transfer from device A. In PSL, the delayed grant sequence would be a temporal property. The source of the transfer along with the grant sequence temporal property would form subsequences that are used to trigger the consequent in the completion mode automata. The checker would then assert its output when cycles 3,4,5 are in progress. Those hardware signals, when asserted enable the storage buffers and thus only the requested data would end up in the captured trace. In such system one would simply have to temporarily remove the on-line assertion checkers and re-program the logic for the trigger logic of the storage buffer. The mechanisms to create the trigger logic is the same as the ones used to make the on-line assertion checkers.

In contrast, most modern logic capture systems offer trigger mechanisms that

are expressed through a simple boolean layer and with a few programmable state machines. Those usually involve learning a new interface for the particular instrument and offer far less flexibility and expressiveness than the use of a full *temporal* language such as PSL.

3.3.1 Assertion Checker Partitioning Algorithm

This tool builds a database of each of the checker modules by automating their synthesis and extracting all the relevant metrics. In the current proof-of-concept implementation, the Xilinx XST synthesis tools for VirtexII FPGA devices are used.

Once the checkers have been individually synthesized and their sizing metrics are obtained, Algorithm 1 is used to create subsets of checkers suitable for multiple reconfigurations in the reprogrammable logic area. This algorithm is based on solving the subset-sum problem by dynamic programming [108]. However, because the circuit metrics comprise two variables, namely # of flip-flops (FF) and # of lookup tables (LUT), the typical subset-sum procedure can not be employed directly on its own. This thesis therefore presents a two-phase algorithm, which returns a near-optimal partition, given the circuits' metrics and the size of the reprogrammable area (also specified as # of flip-flops and # of lookup tables).

Phase 1 in the algorithm (lines 3-8) uses flip-flops as the dominant metric and performs subset-sum on this metric (line 5). The subset-sum algorithm requires that the circuits be sorted in increasing order according to the dominant metric (line 4). A search is then performed for the best subset according to the size limit of this dominant metric which also respects the maximum size for the secondary metric (line 6). Once the best subset has been determined, it is logged and removed from the set (lines 7 and 8). This procedure continues until the set of checkers is empty (line 3).

The dominant / secondary metrics are interchanged and the same procedure is repeated (lines 9 to 14). A comparison is then made between both phases (lines 15 to 18), and the solution with the fewest subsets is logged. When both phases have the same number of subsets, it was empirically observed that the more balanced partition is the one for which the dominant metric corresponds to the metric which is the most constrained by the area limits (smallest freedom).

It can be shown by counterexample that the algorithm is not guaranteed to create an optimal partition; however, our experiments show that it drastically outperforms the brute force approach in computation time. Furthermore, when one of the metrics has a large amount of freedom with respect to its constraint, the problem tends toward a single variable subset sum for which our algorithm is optimal.

Algorithm 1 Assertion circuit partitioning algorithm.

FUNCTION: SUBSET-CIRCUIT(set C of circuit metrics (FF, LUT), $area_{FF}$, $area_{LUT}$)
$D \leftarrow C$
while there are circuits left in C **do** {phase 1 (dominant metric is #FFs)}
    sort circuits C according to $\#FFs$
    build dynamic programming table T for subset-sum on $\#FFs$
    search T for best subset S such that $\sum_{s_i \in S} \#LUTs(s_i) < area_{LUT}$
    log subset circuits in S as a group in phase 1 results
    remove circuits S from C
end while
while there are circuits left in D **do** {phase 2 (dominant metric is #LUTs)}
    sort circuits D according to $\#LUTs$
    build dynamic programming table T for subset-sum on $\#LUTs$
    search T for best subset S such that $\sum_{s_i \in S} \#FFs(s_i) < area_{FF}$
    log subset circuits in S as a group in phase 2 results
    remove circuits S from D
end while
if number of subsets in both phases differs **then** {analysis}
    return results of phase which has the fewest subsets (groups)
else
    return results of phase for which the subset-sum was performed on metric with smallest freedom
end if

3.4 Experimental Results

The effects of assertion threading, assertion completion and activity monitors were explored by synthesizing the assertion circuits produced by the checker generator using ISE 8.1.03i from Xilinx, for a XC2V1500–6 FPGA. The checker generator used is MBAC version 1.71 and the synthesis is optimized for speed (as

opposed to area). The dependency graphs from Section 3.2.2 do not influence the circuits generated by the checker generator, while the assertion and coverage counters from Section 3.2.4 contribute a hardware overhead that is easily determined *a priori*. The number of flip-flops (FF) and four-input lookup tables (LUT) required by a circuit is of primary interest, given that assertion circuits are targeted towards hardware emulation and silicon debug. Since speed may also be an issue, the maximum operating frequency for the worst clk-to-clk path is reported.

This section demonstrates the use of the algorithms. The checker generator is used to produce assertion checkers for two suites of assertions. Some of the assertions are used to verify an AMBA slave device and AMBA AHB interface compliance, and were taken from Chapter 8 in the PSL book [109] by Cohen et al., while others were taken from the book [7] by Foster et al. Finally some come from examples in the text or were created during the development to exercise the checker generator. The AMBA, PCI and CPU example assertions appearing in the tables are derived from our publication [12]. Because of the temporal nature of the assertions and simple storage mechanism (single bit for each state), the assertion checkers utilize more combinational cells than flip-flops. This effect is even more pronounced when large comparators are used in the boolean layer. However, the proposed partitioning algorithm can operate on any type of circuits whether they are balanced or biased towards either flip-flops or combinational logic.

The checkers only monitor the internal circuit signals. This imposes a small extra loading that can at worst add small delays, which can be kept low by following standard synthesis techniques. For instance, for the signals in the critical path that are monitored by an assertion, isolation buffers can be inserted to minimize the loading of the circuit under debug.

3.4.1 Signaling Assertion Completion

The effect of changing an assertion checker into completion mode show that in the checker's complexity remain about the same and logic usage sometimes decreases a bit. Table 3.1 shows the result of using this debug monde on a selection of assertion statements.

Assertion	Normal			Assertion Completion		
	FF	LUT	MHz	FF	LUT	MHz
assert always {a&b} |–> {∼c; {d&∼c}[*0:4]; c&∼d};	6	8	433	6	7	444
assert always ({a} |=> {{c[*0:1];d}|{e}});	3	3	610	3	2	610
assert always ({a;b} |=> {c[*0:1];d});	4	3	611	4	3	611
assert always {a} |=> {{[*2];b;∼c} | {[*2];∼b;c}}; /	6	3	564	6	3	564
assert always {a} |=> {b;c;d;e}; (AMBA asr. [109]) /	5	5	514	5	4	611
assert always {a;∼a} |=> {(∼a)[*0:15];a} abort b; (AMBA asr. [109]) /	18	17	611	18	23	312
assert always {a;b} |=> {c;{{d[*];e}[+];f}&&{g[*]}} abort h; (PCI asr. [7]) /	5	10	468	5	7	470
assert always {a} |=> {e;d;{b;e}[*2:4];c;d};	15	21	329	15	15	430
assert always {a} |=> {b; {c[*0:2]} | {d[*0:2]} ; e};	7	12	333	7	9	414
assert always {{{b;c[*1:2];d}[+]} : {b;{e[–>]};d}} |=> next a;	8	7	473	8	7	473
assert always {a} |=> {{{c[*1:2];d}[+]} && {{e[–>2:3]};d}};	16	38	304	16	31	386
assert always {a} |=> {{{b;c[*1:2];d}[+]} & {b;{e[–>2:3]};d}};	44	141	260	44	139	250
assert always {a} |=> {{{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d}};	35	118	251	35	100	281

Table 3.1: Assertion-circuit resource usage in two compilation modes. The assertion signal definitions use simplified booleans (e.g. A and B and C can be viewed as a new variable D) and the names of the signals are condensed into a single letter (e.g. READY&GNT become a&b). They are identified by the / symbol.

3.4.2 Activity Monitoring

Table 3.2 show that adding activity monitoring only slightly increases the complexity of the hardware checker by a few extra LUTs. This table also shows that the final maximum operating frequency of the checker *decreases slightly*. Fortunately, in FPGAs of that family, designs are not typically aiming at frequency targets in the 300-400 MHz range. Typical Virtex-II designs are targeted to run between 100-200 MHz (at least for most of the logic), which shows that the hardware checker's logic is not likely to be part of the *critical timing path*. In the event a given checker end up loading a critical timing path, they could be isolated by adding a set of flip-flops at their inputs which would only delay the assertion results by a clock cycle, which would not affect the final outcome.

Assertion (assert x)	**Baseline**			**Baseline+Act.Mon.**		
	FF	LUT	MHz	FF	LUT	MHz
always {a&b} |–> {~c; {d&~c}[*0:4]; c&~d};	6	8	433	6	11	429
always ({a} |=> {{c[*0:1];d}|{e}});	3	3	610	3	4	610
always ({a;b} |=> {c[*0:1];d});	4	3	611	4	5	564
always {a} |=> {{[*2];b;~c} | {[*2];~b;c}}; /	6	3	564	6	5	559
always {a} |=> {b;c;d;e}; (AMBA asr. [109]) /	5	5	514	5	6	509
always {a;~a} |=> {(~a)[*0:15];a} abort b; (AMBA [109]) /	18	17	611	18	23	564
always {a;b} |=> {c;{{d[*];e}[+];f}&&{g[*]}} abort h; [7] /	5	10	468	5	12	411
never {a;d;{b;a}[*2:4];c;d};	12	11	564	12	15	559
always {a} |=> {e;d;{b;e}[*2:4];c;d};	15	21	329	15	26	312
always {a} |=> {b; {c[*0:2]} | {d[*0:2]} ; e};	7	12	333	7	14	331
never { {{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d} };	16	19	395	16	24	381
always {a} |=> {{{c[*1:2];d}[+]} && {{e[–>2:3]};d}};	16	38	304	17	40	293
always {a} |=> {{{b;c[*1:2];d}[+]} & {b;{e[–>2:3]};d}};	44	141	260	44	150	259
always {a} |=> {{{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d}};	35	118	251	35	128	243

Table 3.2: Resource usage of assertion circuits and activity monitors. (/ = Simplified Booleans.)

3.4.3 Hardware Assertion Threading

Table 3.3 show the overhead cost of using assertion threading as a hardware assistance to debug. As expected from the replication of the checker circuit, the area scales proportionally to the number of parallel "threads" the hardware is using with the slight overhead introduced by the dispatcher. In the case of A7 and A8, 8-way threading is not required since the assertion automata does not have enough "depth" to be useful and the extra hardware threads would serve no purpose.

Assertion	None			2-way			4-way			8-way		
	FF	LUT	MHz	FF	LUT	MHz	FF	LUT	MHz	FF	LUT	MHz
A1	6	8	433	15	18	386	29	33	306	57	62	241
A2	12	11	564	25	23	564	49	46	433	97	91	408
A3	15	21	329	33	44	298	65	83	252	129	164	235
A4	16	19	395	33	39	395	65	77	395	129	160	318
A5	26	80	246	57	165	252	113	297	205	225	570	177
A6	35	118	251	73	235	239	145	430	213	289	881	186
A7	6	3	564	15	11	442	29	20	362	not required		
A8	5	5	514	13	16	323	25	24	326	not required		
A9	18	17	611	39	38	442	77	75	364	153	144	297
A10	5	10	468	13	23	311	25	39	278	49	67	235
A1: assert always {a&&b} |–> {~c; {d&~c}[*0:4]; c&~d}; (Example 1)												
A2: assert never {a;d;{b;a}[*2:4];c;d};												
A3: assert always {a} |=> {e;d;{b;e}[*2:4];c;d};												
A4: assert never { {{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d} };												
A5: assert always {a} |=> {{{b;c[*1:2];d}[+]} : {b;{e[–>]};d}};												
A6: assert always {a} |=> { {{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d} };												
A7: assert always {a} |=> {{[*2];b;~c} | {[*2];~b;c}}; (Example 3) /												
A8: assert always {a} |=> {b;c;d;e}; (AMBA asr. [109]) /												
A9: assert always {a;~a} |=> {(~a)[*0:15];a} abort b; (AMBA asr. [109]) /												
A10: assert always {a;b} |=> {c;{{d[*];e}[+];f}&&{g[*]}} abort h; (PCI asr. [7]) /												

Table 3.3: Area tradeoff metrics for assertion threading. (/ = Simplified Booleans.)

3.4.4 Checkers Partitioning

Table 3.4 shows the individual resource usage of checkers for the assertions in the AHB and mem_slave examples. In the table, N.A. means Not Applicable, and occurs for circuits containing only one FF with no feedback path (the MHz performance being a clk-to-clk estimate). Table 3.5 shows how the synthesized assertion circuits from Table 3.4 are partitioned into a minimal number of sets by the subset-circuit algorithm, for a target area of 50 FFs and 50 four-input LUTs. In both cases, phase two results were logged (dominant LUTs). The right-most column lists the sums of the circuit metrics in each group.

Table 3.6 shows how the actual resource usage can be slightly diminished when the circuits that form a subset are actually synthesized together. As a general result, it can be expected that, as the number of circuits per subset increases, the resource sharing (a side effect of the synthesis process) becomes more important, and the overall metrics for a given subset become smaller. For comparison purposes, Table 3.6 also lists the full-set metrics, which are obtained by synthesizing all checkers as a single module.

AHB example:

Assertion	FFs	LUTs	MHz	Assertion	FFs	LUTs	MHz	Assertion	FFs	LUTs	MHz
ahb_A1	2	2	667	ahb_A10	1	6	N.A.	ahb_A19	3	3	667
ahb_A2	2	3	611	ahb_A11	2	30	667	ahb_A20	3	2	667
ahb_A3	2	3	667	ahb_A12	2	18	667	ahb_A21	1	23	N.A.
ahb_A4	2	2	611	ahb_A13	2	18	611	ahb_A22	1	21	N.A.
ahb_A5	2	2	667	ahb_A14	1	12	N.A.	ahb_A23	1	21	N.A.
ahb_A6	2	2	667	ahb_A15	1	36	N.A.	ahb_A24	1	19	N.A.
ahb_A7	2	2	667	ahb_A16	2	20	667	ahb_A25	1	6	N.A.
ahb_A8	2	2	667	ahb_A17	2	2	611	ahb_A26	18	17	611
ahb_A9	2	2	667	ahb_A18	3	2	667				

mem_slave example:

Assertion	FFs	LUTs	MHz	Assertion	FFs	LUTs	MHz	Assertion	FFs	LUTs	MHz
mem_slave_A1	1	4	N.A.	mem_slave_A10	5	16	456	mem_slave_A19	1	5	N.A.
mem_slave_A2	2	4	667	mem_slave_A11	5	22	469	mem_slave_A20	1	6	N.A.
mem_slave_A3	2	2	667	mem_slave_A12	2	3	667	mem_slave_A21	1	6	N.A.
mem_slave_A4	2	2	667	mem_slave_A13	2	3	667	mem_slave_A22	1	1	N.A.
mem_slave_A5	1	2	N.A.	mem_slave_A14	2	3	667	mem_slave_A23	2	5	667
mem_slave_A6	1	7	N.A.	mem_slave_A15	2	3	667	mem_slave_A24	1	4	N.A.
mem_slave_A7	1	2	N.A.	mem_slave_A16	2	7	667	mem_slave_A25	1	3	N.A.
mem_slave_A8	1	7	N.A.	mem_slave_A17	1	2	N.A.	mem_slave_A26	1	18	N.A.
mem_slave_A9	4	9	417	mem_slave_A18	4	12	442				

Table 3.4: Resource usage of assertion checkers.

AHB example:

Subset	Assertion circuits in partition	ΣFF, ΣLUT
#1	{A9, A14, A15}	4, 50
#2	{A8, A22, A23, A25}	5, 50
#3	{A7, A10, A21, A24}	5, 50
#4	{A6, A11, A13}	6, 50
#5	{A1, A2, A3, A4, A5, A12, A16}	14, 50
#6	{A17, A18, A19, A20, A26}	29, 26
	Total:	63, 276

mem_slave example:

Subset	Assertion circuits in partition	ΣFF, ΣLUT
#1	{A6, A8, A19, A20, A21, A22, A26}	7, 50
#2	{A1, A11, A15, A18, A23, A24}	15, 50
#3	{A2, A3, A5, A7, A9, A10, A14, A16, A17, A25}	21, 50
#4	{A4, A12, A13}	6, 8
	Total:	49, 158

Table 3.5: Checker partitions for reprogrammable area.

The end result is an efficient partition of checkers which minimizes the number of times the reprogrammable logic area must be reconfigured. A test procedure can then run a batch of test sequences with a given subset of checkers, then instantiate a new set of checkers, re-run the test sequences, and so forth. Once the verification with checkers is finished, the reprogrammable fabric can be re-used for the functionality of the intended design.

3.5 Chapter Summary

This chapter described a set of specific enhancements to temporal logic to hardware translation tools that are increasing the usefulness of the generated hardware when used in the context of debug. Some of the proposed debug enhancements incur little hardware overhead, yet provide very important information that is intrinsically close to the logic structures and can thus record a very accurate signature of a hardware bug. Using this signature and external data analysis, designers can create accurately targeted verification scenarios in their simulation environment to

AHB example:

Subset	FFs, LUTs
#1	4, 50
#2	5, 50
#3	5, 49
#4	6, 34
#5	13, 48
#6	29, 26
Total:	62, 257

Full-Set (FFs, LUTs)
60, 250

mem_slave example:

Subset	FFs, LUTs
#1	7, 43
#2	15, 47
#3	21, 47
#4	6, 8
Total:	49, 145

Full-Set (FFs, LUTs)
48, 129

Table 3.6: Subset and full-set synthesis of a sample of hardware checkers.

address the problems in a faster, more direct manner.

Chapter 4

Memory Mapping of Hardware Checkers

The previous chapter showed a series of enhancements that can be added to hardware assertion and sequence checkers to increase their usefulness during debug. Hardware checkers augmented with debug enhancements alone are useful, but only up to a point. Their benefit to debug are greatly enhanced when many assertion checkers, sequence monitors and coverage counters are considered together as a more complete "health report" of the device under debug.

In this chapter, the assertion monitors, coverage counters and sequence monitors will be referred to as *checkers* or *hardware checkers*. This chapter will explain how hardware checkers can be integrated into larger systems by grouping them via the use of CPU-accessible addressing space. In doing so, the control points, coverage counters and debug enhancement outputs from the checkers will be accessible in the same manner as other hardware peripherals. The addressing space, which is an abundant resource in modern systems, especially in newer 64-bit architectures, allows the *virtual* concatenation of checkers from different physical areas of the system to appear contiguous when observed from the point of view of a given CPU. The proposed toolset allows direct use of the on-chip resources to monitor the checker's output via a register file that the local CPU resources can access. An algorithm is proposed to automatically *pack* registers to maximise the use of the memory addresses and its runtime is demonstrated by packing up to 1000 checkers and their respective controls.

Later in this chapter, a mechanism is proposed to integrate the generated assertion checker groups in the context of an operating system. The proposed mechanism leverages the device file abstraction offered by the operating system to carry the virtual to physical memory mapping process. This method allows applications to directly interface with the hardware checkers and is demonstrated to be scalable, secure and easy to adapt to operating system updates, while saving considerable development effort.

4.1 Need for Automation

It is possible to benefit from the hardware checker's debug enhancements presented in Chapter 3 in a CPU-based system by *manually connecting* the output of the checker to a hand-coded register file. One would then validate the resulting system by issuing CPU reads through a BFM performing the low-level steps involved in accessing the embedded register. This would be done in a testbench executed by a hardware simulator. The final qualification of the register access would be performed on a hardware prototype. However, the tediousness of connecting the output of many assertion checkers (hundreds or thousands) along with the error-prone process of defining and mapping the resulting address location and bit position will use a lot of expensive hardware engineering time. To improve efficiency and acceptance of the proposed methodology, automating the connection and generation of hardware interfaces to the memory map is our primary objective.

The hardware engineer only has to provide the PSL file to the register generator tools and the proposed process will reliably connect the checkers to the appropriate control structures and maintain a database of the resulting connections. Since this automated mapping process can be thoroughly validated through unit testing, post-synthesis simulations and regression testing, it will be able to rapidly and reliably connect a large number of checkers to the memory map without increasing the engineering workload. As a side benefit, the internal database can also be used to output C-based data structures or output a database that can then be used in-system to interpret the memory bit mappings.

4.2 Memory Mapping Concepts

This section proposes a quick overview of the memory mapping concept and the supporting hardware required. It will help illustrate the system-level considerations needed to integrate hardware checkers as slave peripherals in a memory-mapped system.

4.2.1 General Overview

In modern IC architecture that is comprised of processors and peripherals, there is one (sometimes more) well-defined bus architecture that allows the CPUs to communicate with the peripherals. Generally, a given CPU has direct access to a high-speed cache memory and then, one level below, to a local bus interconnect. The cache memory grants repetitive accesses to the same memory region to be performed with a comparatively lower delay on a copy of the data than directly manipulating the main memory content. This advantage of locality of information brought about by cache memory offers great speed in normal use and explains why most performance-oriented embedded processors make use of this technique.

A transaction that is not handled by the cache memory will propagate to the high-speed peripherals that are directly connected to the bus and possibly through a bridge unit to lower-speed secondary buses. What determines the destination of the transaction is its *address range*. In most systems, when the CPU issues a transaction, the intended recipient will perform the operation (read or write) and provide an acknowledgement. On the other hand, if the transaction is sent to an address where no decoding is performed and no peripheral is present to process it, the transaction typically terminates with a bus error and an exception mechanism is triggered in the software.

A bus-based architecture thus allows the addition of peripherals by allocating a new address range and providing hardware resources to decode and respond to requests targeted to this new area of memory space.

4.2.1.1 Volatile Registers

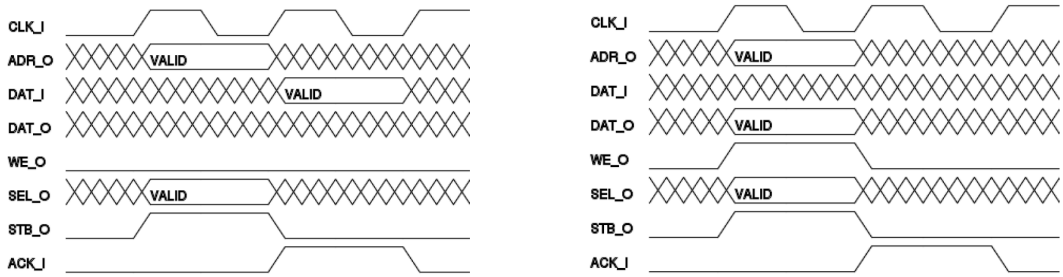
The term *Volatile Register* is often used to describe a memory location (a specific address) that is mapped to a hardware-based structure, subject to change without any CPU intervention. This is in contrast to *Random Access Memory* (RAM) where the content is always under the control of the CPU. The RAM regions benefit from being *cacheable*, which improve performance. On the other hand, the hardware based resources are not cached [1] since they may change independently of any CPU activity. The operating system must know about the hardware-based registers or other *volatile* areas of the memory and enable the processor-specific modes to perform the hardware accesses with the caching mechanisms disabled. The exact mechanism used to bypass the cache memory varies from one architecture to another. Generally, if the system supports *virtual memory* then the caching behavior can be defined for each page and controlled by the OS kernel. In other systems, the cache memory behavior is defined per block or per *chip select* line (hardware signal used to access the peripheral).

4.2.2 Wishbone Interconnect

Some form of standardization of the high-speed and low-speed bus internal buses benefits SoC architectures as they allow the re-use of components and rapid integration. A candidate of interest when building custom SoC in a research environment is the Wishbone [43] interface. The specification is *open source* and anyone can implement Wishbone-compatible hardware blocks without seeking licensing permissions. Furthermore, a lot of available *open source* hardware cores have already been implemented using this specification, providing numerous examples and useful hardware modules.

Figure 4.1 illustrates two of the fundamental Wishbone bus cycles, namely the pipelined read and pipelined write cycles. With only those two bus cycle types, one can build a memory-mapped block. The cycles are synchronous and use a

1. They could support caching, but some invalidation mechanism must be put in place to support hardware-based invalidation messages to be triggered when registers are updated. If those registers are subject to constant changes (e.g. activity counters), then they could generate many invalidation messages and affect the system's performance.



(a) Pipelined Read **(b)** Pipelined Write

Figure 4.1: Example Wishbone Bus Cycle Timing

system-wide clock illustrated as ***CLK_I***. The bus master (typically the CPU unit) drives ***ADDR_O*** and asserts ***STB_O*** to indicate the transfer. The slave peripheral responds with ***ACK_I*** and in the case of a read, drives the ***DAT_I***.

The Wishbone specification also proposes a non-pipelined version of read and write that is a bit less efficient when scaled to larger systems due to the loading on the bus and the longer time between transfers. The Wishbone specification also details more advanced bus cycles [43] (block transfers, atomic modifications) and optional signals to improve transfer efficiency and to support multiple masters.

The Wishbone interconnect has been used in a similar research project at the TIMA laboratory in France, also based on an assertion-based methodology and on-line checkers [110]. Their research project proves that there is the possibility for continuous on-line monitoring of the memory mapping hardware layer itself. This could be used in addition to the monitoring of components in the system.

4.2.3 Other Interconnects

Wishbone is not the only interconnect available to build up complex SoC. CoreConnect from IBM, used in this research to connect the PowerPC processor of the FPGAs in the BEE2 prototyping machine, is one example with a set of features similar to the Wishbone interconnect.

The AMBA AHB interconnect [42] is also very prevalent due to the large number of ARM-based systems in the marketplace. Some companies offer AMBA compatible cores, notably Aeroflex Gaisler, who makes the Leon II processor (SPARC instruction set) that was used in early development of our NoC-based RTL-level

prototypes.

4.3 Register File Structure

Once the interconnect has been selected, one can focus on the actual register file's internal format. A properly implemented register file interface can considerably reduce the resulting firmware and software complexity. The term *firmware register* is used in this chapter to describe a *group of bits accessible from the local CPU interconnect*. This terminology differs from the one used when describing logic hardware, where a *hardware register* simply describes a collection of bits present in the circuit. A *Firmware register* is therefore a *hardware register* from the point of view of the logic circuit.

A *hardware register* can't be seen by the firmware and is not considered in the algorithms presented below. For example, the sequence checking automata state information is not made visible to the firmware. Some automata state information is visible by the firmware only when explicitly requested as part of a debug enhancement, for example when performing *Antecedent Monitoring* (Section 3.2.1). In that case, a firmware-visible register is created.

From experience (which one can gain by reading many datasheets from different IC vendors), firmware registers are usually defined using a single bit for control or status information that is Boolean. An example of this would be *assertion check failed* or *coverage counter enabled*. Multi-bit registers are used to control parameters requiring encoding (e.g. multiplexing different sources of information) or to report multi-bit values such as in the case of coverage counters in this application.

Registers have different possible *access types*:

- **Read Only (RO)**: A register that reports information. Writing to this register has no effect.
- **Read Write (RW)**: A register that reports an internal control value state and which can be manipulated by the firmware.
- **Clear on Write 1 (CW1)**: A register that typically reports an event recorded state and that can be cleared by writing a 1 to the location.
- **Clear on Read (CR)**: This type of register has a value that can only be read

once for each event recorded. Once read, the value will be affected. It can be used to report counts and perform an atomic clear when the count is read. Generally, it is not a recommended practice to have this kind of register type as they are difficult to debug. If the memory range is *watched* by a software debugger, the value will constantly be read and effectively becomes useless [2]. This type of register offers a performance advantage which is why they are considered as an option in the RTL generation of the hardware checker register file.

- **Set on Write 1 (SW1)**: A register that can be written by the CPU and that can only be set. The hardware usually clears the register after an operation completes. It is usually used to start processing sequences and can be monitored by the CPU to know the end of activity.
- **Write Only (WO)**: A seldom used type of register which can accept write commands and will typically initiate an operation in the hardware. Normally, this type of register is not appreciated by firmware programmers since it is quite difficult to know if it was accessed correctly or not due to lack of feedback from the hardware. An advantage would be to hide a certain set of proprietary control or test registers or to save a few logic gates.

For detailed discussion and guiding principles for making efficient firmware interfaces to the hardware, one can refer to Gary Stringham's book [111]. The rules and principles presented in this reference were acquired over many years of engineering practice and, when followed, will lead to higher firmware performance and easier integration.

In this work, the automatic generation of hardware registers interfacing to the assertion checkers has to produce efficient register partitioning and bit ordering by adhering to as many of the book's proposed rules and guidelines as possible. Those registers may end up in the final silicon (those most useful for post-silicon debugging). Ensuring that they can be used *efficiently* by firmware to assist in debugging, gathering coverage and as in-system monitoring points is an important concern.

2. Unless the debugger has information about this register type and handles it properly

4.4 Tool Flow

This section details the required data structures needed by the checker *Aggregator Tool* to properly generate in-system registers that can be read by the device processing units as part of the system memory map.

The Python programming language was chosen for the implementation of the tool. Python is an open source software project, supports object oriented programming and offers many advanced text manipulation libraries and parsers on top of a very good integration with the operating system.

Python was already used to automatically launch syntax check, synthesis and output log parsing to produce the results presented in Chapter 3. It was also used to launch the Xilinx tools used to automate the process of extracting synthesis data from hardware checkers to obtain post-fitting timing information.

Python is an *interpreted* language, thus allowing rapid prototyping of ideas since code changes can immediately be tested. Python's powerful list manipulation is used to organize and extend the data structures during the register generation process. The slower execution speed of Python when compared to a *compiled* language such as C/C++ is not an issue as it takes less than a second to generate the RTL for a typical set of assertions. The synthesis process to translate this generated RTL to gates is much slower (a few minutes) ; FPGA place-and-route is even more time consuming (up to a few hours for large FPGAs, for example those present in the BEE2 system when they were configured to include a CPU and peripherals).

4.4.1 Phase 1: Source File Processing

The first part of the process is to associate the source file and assertion line number with the resulting hardware sequence monitor or assertion checker. This is a process that starts by parsing the PSL sources and the associated generated Verilog sources.

The problem can then be formulated with the following data structures available to the algorithm:

– Original PSL file name.

– Line defining the assertion statement.

- Assertion status bit name (signal name in the HDL), provided by the hardware generator.
- Special debug-enhancement compilation flags used their associated signal outputs.
- Width and name of associated counters. The width of the counter can default to the register interface bus width, but can be changed by a tag in a comment just before the PSL statement.

4.4.1.1 Implicit Checker Control Structures

Each hardware checker unit requires a few control points. First, a reset signal to force the checker's automata in its default state. Note that this is different from the *device reset* that will also reset the circuit along with the checkers. An *end of execution* control signal is required (per ***vunit***) to tell the hardware automata that any pending eventualities are reported as errors by the checkers [96]. This allows the use of the PSL statement ***eventually***. Finally, a way to reset *latched* assertion failures or sequence completion in the event that a subset of the checkers have to be re-started to monitor the re-apperance of a failure condition. Those control points are labeled as control register and must be accessible via the register interface as *read-write* elements.

The actual output from the automata generation is a single bit per mode (assertion failure, sequence completion, coverage, pre-condition) that pulses when an event occurs. A checker output can thus be turned into a multi-bit counter by using ***cover*** or ***assert*** statements and provide options in the comment preceding the PSL statement.

It may not be immediately obvious why a count of assertion failures can be useful for debug, so to explain, consider the following scenario. Suppose that in a system, a burst transaction of 10 cycles is performed. The assertion checkers capture that a protocol error has happened. This is certainly useful and is covered by a single assertion failure event latch in the hardware. However, having the knowledge that 5 assertion failures happened can hint that the problem may be related to the burst length or to the fact that cycles are occurring in pairs, for example. As a way to confirm or invalidate this hypothesis, a burst of 11 cycle or 12 cycles could

be generated within the system and based on the number of protocol violations cumulated, one can more rapidly identify the underlying cause of the assertion failures. With simple counters, isolating the problem by modifying the stimuli and repeating the tests will produce a signature that can then be used to better characterize the problem or prepare a directed test case in the simulation environment. If an on-chip problem can be replicated in the simulation environment, the task of locating and fixing it is tremendously reduced.

Thus, from the above discussion, the database will include the following:

1. References to the signals indicating an assertion failure, sequence completion or pre-condition completion status. Those signals are recorded for transformation into *Read-Only* (RO) *status* bits. In some generation modes, they can also be considered as *Clear on Write 1 (CW1) control* bits since this will help re-start many checkers in a single CPU write instruction.
2. Control points: For assertion failures, sequence completion or pre-condition completion, signals acting as control points can be used to hold in reset some checkers monitor such that the results of those checkers can be easily ignored. This allows the debug process to be re-started by partially enabling the hardware checkers to isolate a problem and eventually corner up a bug. These control points are recorded to map into *Read-Write* (RW) single-bit control registers and originate from inside the unit (they don't come from the circuit under debug or from the generated hardware checkers).
3. Assertion coverage counters or pre-condition counters: as opposed to single-bit status or control registers, the multi-bit values of those counters must be read in one atomic operation to avoid any race conditions or erroneous readings. From the system's point of view, these counters are *Read-Only* (RO) multi-bit vectors. Associated with each counter are control lines to allow them to be reset. Depending on the tolerable hardware overhead their control bits can all be independent or grouped.

4.4.2 Phase 2: Checker Grouping

In this phase, the checkers and their monitor/control are grouped in clusters that represent a sub-section of one of the targeted hardware unit. In our work, the natural grouping unit is a PSL *vUnit*. However, this doesn't have to be a "hard" rule, as more than one *vUnits* could be merged from the point of view of the generation of hardware registers, making the address decoding more efficient and reducing the hardware overhead of the implicit control structures. Typically, *vUnits* are self-contained entities in PSL so they are better suited to be encapsulated as a set of registers with a dedicated address decoder. Furthermore, *vUnits* tend to attach quite specifically to a given *entity* in a design. Keeping the required hardware signals localized limit the span (physical proximity) of metal wires feeding the hardware checkers. The last thing one needs in a large IC is assertion checker wires covering a large area of the die and causing routing problems and possible reductions clock rates due to the checker's circuit becoming part of the *critical timing* path.

Each checker group has a specific *bus size* associated with it. To avoid building the inter-locking logic that allows reads of counters larger than the bus width, the biggest single counter width in a register group must be less than or equal to the selected bus width. In typical systems, this means that the counters should be limited to 8, 16, 32 or 64 bits, depending on the bus size chosen for the decoder unit. This limitation can be circumvented by adding the interlock mechanism presented in Section 4.4.2.2, but this was not implemented in the RTL generator. The interlock mechanism, if implemented, would have to be added during this phase in the object database since the shadow register will be shared by a group of registers. The current automated register file generator tool will abort with an error if a counter is requested to be larger than the bus width of the module.

The event counters are all based on a binary counter with saturation logic which avoids wrap around when the counter reaches its maximum value. Therefore, the counters do not have to be very wide to provide useful information or to gather coverage. The degenerate case is a counter of width = 1 which becomes a "latch" of the condition. If in-silicon verification coverage has to be gathered, the saturating counter solution will provide at least a *lower bound* on the number of recorded

events.

A complete hardware checker is thus comprised of an automata produced by the checker generator connected to the circuit being monitored, a set of control registers and a set of event latching circuits and counters. The event latching circuit, counters and automata control is generated by the tool. The automata core circuit is produced by the MBAC tool which is detailed in the book by M. Boulé and Z. Zilic [96].

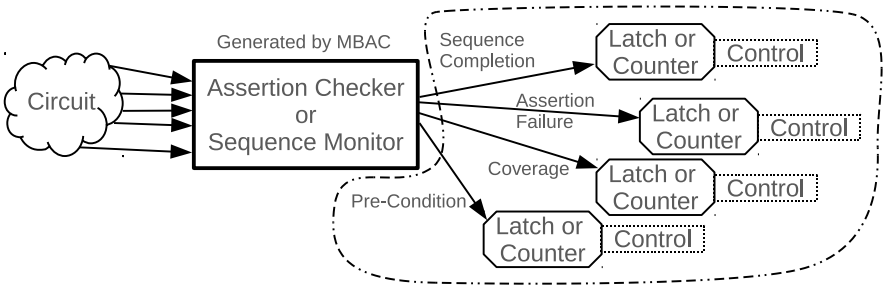


Figure 4.2: Circuit-level (hardware) view of a hardware checker and its associated control and status units

Figure 4.2 illustrates the hardware view of the assertion checkers in terms of registers *bitfields* (control or status). The dashed line surrounding the set of control and status registers represent the hardware that will be generated outside of the MBAC automata. The arrows entering the dashed line will represent input ports on the generated module. The figure doesn't show the local interconnect interface. At this point in the tool flow, registers are created based on the generation mode of MBAC and only exist as objects and are not yet bound to specific addresses.

4.4.2.1 Clear-on-read for Software-Based Counters

A clear-on-read is optionally supported by the HDL generator allowing the counters to be scanned in rapid succession, even though this breaks one of the firmware rules [111]. This alternate generation mode allows the read of the register without requiring any *write* access to the register control to reset its count. That way, the firmware can continuously accumulate counters without requiring double-buffering of the values (expensive in hardware). The clear-on-read method in combination with a firmware driver provides larger coverage counter values,

while reducing the hardware overhead, since the large counters end up residing in the main system memory which is a resource more abundant than hardware flip-flops. In that mode, it helps to avoid letting the register packing algorithm place multiple clear-on-read registers on the same address line. Multiple counters on the same address will add complexity to the firmware interface (requiring the firmware to mask, shift and add values to those many independent software-based counters). Hardware-based event counters have to be made sufficiently large with respect to the possible assertion firing rate (worst possible case is one firing per clock cycle) to impose a "refresh requirement" on the firmware such that it can easily be performed in the context of a standard operating system (1 ms being a realistic objective). Should the deadline be missed, the counters will saturate, indicating to the firmware that the count is not *exact* anymore. However, the information remains usable for debugging.

4.4.2.2 Atomic access of large counters

In applications where the data bus width to the register interface *is less than the counter value that needs to be read*, some hardware assistance is needed to correctly access those registers. If a counter value spans *two* addresses and *its counting rate is low enough to guarantee that the lowest portion of the counter cannot overflow more than once within two consecutive read cycles*, one can use a known firmware technique to read the least-significant part of the counter, followed by a read of the most-significant part, and finally re-read the least-significant part a second time. This should be done with the interrupts disabled to prevent a possible task switch which could violate the condition that the lower portion of the counter cannot overflow more than once. By noting the presence or absence of overflow in the lower-portion of the counter, the firmware can obtain a valid reading without increasing the hardware overhead.

The more usual approach is to use extra hardware such that when a CPU accesses the lower part of a multi-address register, the upper part is automatically saved during the same clock cycle and presented in one (or many) *shadow register(s)* that will be read subsequently by the firmware. This adds hardware cost (storage for the upper part of the register), but makes it simpler to interface to a

firmware library. Again, if multiple threads can possibly access the same counter, mutual exclusion to the register unit must be implemented to avoid losing the shadow register value mid-way through the transaction.

4.4.3 Phase 3: Register Map Generation

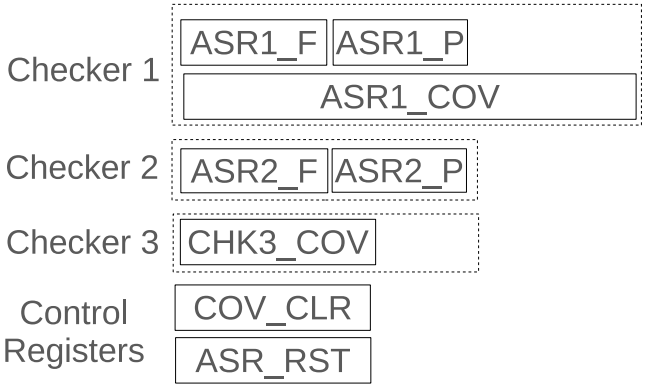


Figure 4.3: Logical Unpacked View

This phase considers each element (control, status or counter) as an object having a few attributes, namely *access type*, *bit width*, *bit position* and *offset from base address*. Initially, the collection of objects derived from the database of checkers have some parameters that are yet undefined, such as their bit position and their address offset from the base of the module. They can be viewed as a *loose* collection of objects as illustrated in Figure 4.3. The packing algorigthm is then used on those objects. Upon its completion, the objects will have a complete representation in the system address space. With this method, it is also possible to incorporate some objects with a pre-defined position and offset information which will be placed first in the memory map and then the newly generated registers can be overlaid. This way, it is possible to run the algorithm on a pre-existing map to "fill in" checkers without losing the previous bit positions. This could prove to be important if, for example, older firmware already assumes the presence of checkers at specific locations (for example in a previous revision of the physical device).

A few trade-offs are possible in this phase, especially on how to place the various control/status elements *bitfields* in each address location. The approach fol-

lows the *bin packing problem* for which algorithms exist in the literature [112]. In our implementation, an optimal solution is not required (the abstract bin packing problem being NP-complete [112]), and a heuristic approach (first-fit decreasing) was used to provide an acceptable solution. Section 4.5 details the packing algorithm.

Figure 4.4: Packed View

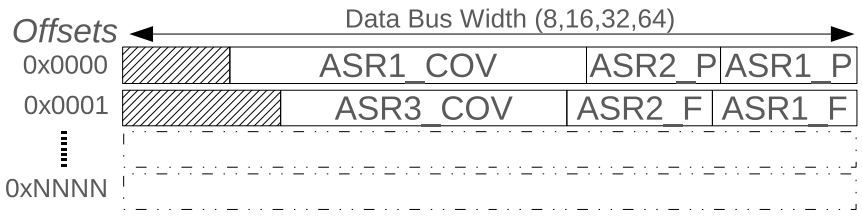


Figure 4.4 summarizes the final step in the process of aggregating the checker's outputs. At this point, all the objects now have a memory map representation and from this a set of firmware structures can be derived.

4.4.4 Phase 4: RTL Generation

The last phase requires that all the bitfield objects have their members unambiguously defined by the previous step. As explained in Section 4.2.2 and Section 4.2.3 a few ways exist to interface the register block to the CPU interconnect. Internally, the register file and address decoder do not depend much on the interconnect except at the handshaking level. The implementation targets the Wishbone interconnect, but trials were also done on the CoreConnect architecture used in the BEE2.

The first phase in the generation of the RTL is the creation of the system ports (clock, reset), the bus interconnect ports, and finally the control outputs and event/status inputs. Then a set of internal Verilog ***reg*** is created, such that the memory elements are defined. For each event input requiring a counter, a saturating counter unit (used to avoid counter wrapping) is *instantiated*, *parametrized* with its width and connected to the corresponding signal coming from the generated checker. Finally, the address decoder and logic is generated such that all the internal control and status points end up accessible from the SoC interconnect interface.

In this research, only single access cycles were supported, but with a more complex implementation of the interface controller, it is feasible to support *burst or block* transfers. This would improve the efficiency for modules with a large number of checkers.

4.4.4.1 RTL Language Selection

A first version of the HDL Generator (used to produce the RTL that will link the assertion checker outputs to the register file) was attempted using the VHDL language as the RTL output for familiarity reasons. However, VHDL being a strongly typed language, **a lot** of overhead was incurred in defining the software classes handling data types, type conversion functions and other elements needed to properly produce *understandable* RTL from the software-based object collection. In the end, Verilog was selected as the language of choice for the hardware RTL output. Verilog code is simpler to abstract as software-based objects and does not have any of the complexities of VHDL's strong typing.

4.4.4.2 HDL Classes

To make the software structure of the generator more efficient, an abstract base class ***HDLObject*** was created as a placeholder from which nets, registers and ports can be derived since they share a few common attributes. Other base classes were derived, each representing a different element of the hardware structures present in RTL. The list below outlines the main classes of the HDL generator:

- ***HDLObject***: The abstract base class for the low-level RTL objects. It is used to hold notable attributes: *Name*, a *Comment String* (useful to document the generated RTL output), a *Width* (number of bits) and a *Weight* which allow the generator to later *sort* out signals such that "lighter" ones "float" to the top of long lists. The *Weight* attribute is useful to control the RTL output and change the order of port declarations for example, to ensure that general signals like clock and reset are listed first. From this abstract base class, the following child classes are derived:
  - ***Net***: Represents a net in the RTL (verilog **wire**)
  - ***Register***: Represents storage in the RTL (verilog **reg**)

  - ***Port***: Represents a port on the module.
- ***Module***: Represents a hardware module. Member attributes include a list of ***Port***, list of ***Net***, list of ***Register***. The module thus contains the objects above in the same way that the final Verilog RTL module is structured.
- ***ModuleInstance***: Represents an instance of a hardware module. This is used to instantiate many Verilog primitives (manually coded blocks) within the generated code. As with ***Module*** class, the manually-coded hardware instances have to be described as Module *objects instances* in Python before they can be instantiated so their ports will end up automatically connected to the enclosing module internal nets.

4.4.4.3 Register Classes

A set of classes are used to describe registers. Here, the term *register* represents a single or group of bits that is accessible from the access port (typically for access by a CPU).

This code layer defines two classes, namely the ***BitField*** class and the ***Register*** class. The ***Register*** class can contain an arbitrary number of ***BitField*** instances, but practially speaking, the number of bitfields in a ***Register*** is never greater than the bit **Width** of the register. The ***BitField*** also carries a **Width** attribute, so they can represent either single or multi-bit elements. It also carries the *access mode* of the group of bits, for example *Read Only* (R) or *Read Write* (RW).

This ***Register*** class is also re-usable in any other context where automatic register generation is required. Defining a group of ***Registers*** containing control or status ***BitField*** members can be passed on to the HDL generator to generate a module that can interface to the local processor bus.

4.4.4.4 Checker Classes

In parallel to the *hardware description* classes, another set of classes describes the *assertion and sequence checkers* elements in an abstract way. At the level of this tool, they are represented by a name that matches the checker generator's output signal for the given checker/counter. Each assertion will contain a set of register classes by a name with extension (e.g. asrchk_14_**rst**) connected to the control registers

that are provided by the infrastructure.

4.4.4.5 Register Decoder Class

Once all the data structures are in place, the top-level objects are passed on to the ***RegisterDecoder*** class where it the registers along with the hardware checker classes will be transformed into RTL.

4.4.4.6 Firmware Driver Header File Generation

Section 4.6 explains how the register mapping can be integrated at the operating system level to be used by application code. To facilitate this process, we suggest that along the hardware generation phases, some firmware code be produced to later assist in the diagnostic. Since the register file definition for the hardware checkers is present in object form, one can derive the data structures representing the bitfields such that when this data structure is overlaid on the hardware memory map, the software can use meaningful names to manipulate the assertion checkers and control registers.

By mapping the hardware checkers to a C-based software structure, the debug process can be assisted by writing firmware routines that dynamically monitor and react on specific assertion checkers firing. As a benefit, debuggers such as GNU Data Display Debugger [3] will be able to *interpret* the overlayed memory structure and provide a way to browse meaningful data structures when examining the hardware checker outputs as illustrated in Figure 4.5.

It is very easy to introduce errors in this hardware-to-C bit mapping process if done manually. The mapping is also dependent on data bus width and target endianness due to the ambiguity in the definition of C-based bitfields. An automated mapping process can properly pad the data structures to maintain proper bit position and alignment to hardware registers in addition to eliminate human errors.

The associated C data structures expressing the register bit positions are given in the following listing to highlight the key elements.

3. `http://www.gnu.org/software/ddd/`

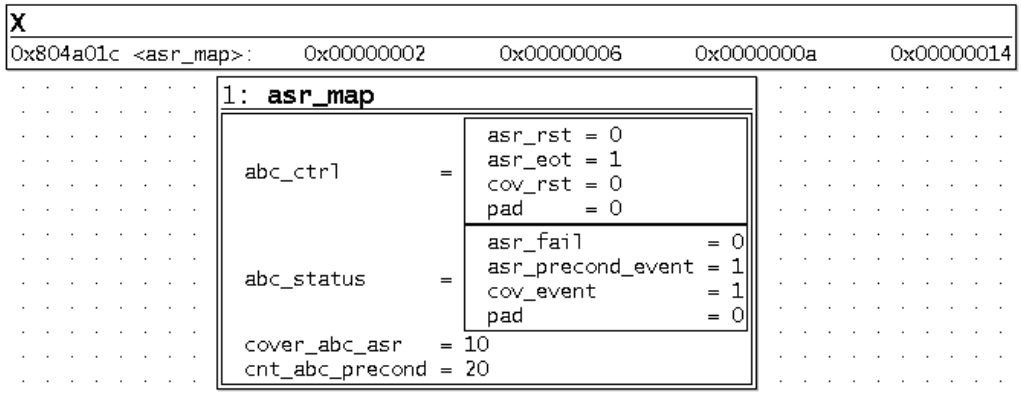


Figure 4.5: GNU Data Display Debugger screenshot of hypothetical hardware checker ***abc*** under debug. Top box illustrates the memory values of the hypothetical checker and the lower box illustrates its interpretation when remapped to a C-based data structure

Listing 4.1: Example C structures for assertion checker register map

```
/* Next two preprocessor directives adapted from ARM CMSIS */
#define     __I     volatile const /* defines read only permissions */
#define     __IO    volatile       /* defines read/write permissions */
/* Hypothetical checker 'abc' control */
typedef struct {
    unsigned asr_rst:1;
    unsigned asr_eot:1;
    unsigned cov_rst:1;
    unsigned pad:29;
} abc_ctrl_t;
/* [...] Status skipped to abbreviate example */
typedef struct {
    __IO abc_ctrl_t     abc_ctrl;
    __I  abc_status_t   abc_status;
    __I  uint32_t       cover_abc_asr;
    __I  uint32_t       cnt_abc_precond;
} asr_map_t;
```

A great source for inspiration in creating a proper firmware interface is ARM CMSIS [113] as it provides a well-defined, compatible way of assigning C data

structure overlays for memory-mapped hardware registers [4].

4.5 Bitfield Packing Algorithm

The packing algorithm can generate different memory layouts from a set of input bitfields:

1. **Densest** : Mapping as many bitfields as possible in the smallest numbers of registers (addresses). The advantage is that reading all the status and control register information will be faster and more efficient. Another benefit is that it will reduce the decoder overhead. One major inconvenience of this mode is that status, coverage and control may end up in the same address. "Clear-on-read" operations will affect many bitfields simultaneously which will make the firmware more complex. It will also be very difficult to intuitively understand the assertion state of the unit (for example by examining a dump of the memory) in the context of a debug session.
2. **By Type** : Separate the registers by type (Control, Assertion Status, Coverage) and pack bitfields accordingly. This allows the control registers to reside in their own address space that can then be protected from accidental writes. The assertion failure status (one bit per assertion checker) can be read out very efficiently and are maximally packed (all the bits in a given address are used since assertion failures are represented with a single bit per checker). Intuitively, by dumping the memory range representing the assertion failures, debuggers (even humans) can rapidly see if all the values are zero (no assertion failure) of if a bit is set. For example, a typical hardware block may host a few hundred assertion checkers which could be represented by a very small block of memory of a few 10s of addresses. Finally, the coverage counters can be packed in separate addresses as they are not as critical as assertion registers.
3. **By Assertion**: This is the least dense packing method. This generates a few addresses per assertion (depending on how much control and counters are

4. More specifically, refer to the templates and information located in the CMSIS/Template_-DeviceSupport of the CMSIS source archive.

enabled for that assertion). This packing is the easiest to use in debugging as each address relates only to one assertion. The drawback is that it will use many addresses and each address has little information associated with it. The address decoder overhead will also be more important. Propagating those assertions in a NoC (as will be explained in Chapter 5) will waste bandwidth as many addresses will have unused bits that are needlessly propagated.

The above memory layout options are not exclusive to each other. Within the hardware interface to the SoC bus, the address decoder can be expanded to offer multiple memory layout decoding options simultaneously (provided one can support the hardware overhead). For example, in a given block, offset 0x0000 to 0x00FF could address the assertion checkers organized using the *densest* packing style, while offset 0x0100 to 0x01FF could address the assertion checkers with the *by assertion* packing style. In such case, during the HDL generation, the signals are only declared once, but the read/write decoder appears twice to perform the address decoding logic for each modes.

To draw a parallel where multiple different accesses to the same hardware structures has been implemented, one can look at the latest ARM Cortex-M3 processing core and its use in embedded systems. To allow efficient atomic single bit manipulation, a process called bit-banding is used to re-map a portion of the RAM memory into a much larger address space such that a *single bit* in the RAM is mapped to a 32-bit address. This effectively "wastes" 31 bits in the address space for those remapped address. However, since no hardware is attached to those 31 "wasted" bits and typical embedded processors do not use all their memory space, this provides a good practical example of trading off address space for functionality.

One has to make a distinction between using a large addressing space to simplify the access to the information and the need to minimize the "wasted" memory bits for each address. The end use of the register file will dictate which packing mode to select. The algorithm presented can compact the checker outputs into as little address space as possible (dense packing mode) to allow faster (higher-performance) transfer of the status from the register file. This, for example, helps

when propagating the information inside a NoC. Thus the *densest* packing mode aim at improving bus utilisation when performing transfers. However, less *dense* packing modes offer a simpler way for the firmware to *interpret* the information.

Algorithm 2 BitField packing algorithm for the assertion checker bit fields

Require: List of bit fields (BFs) and associated width, $BusWidth$
Ensure: width(BF) in $BFs \leq BusWidth$

```
1: Sort BFs in descending width
2: AvailableWidth = BusWidth
3: Allocate RegisterTable and allocate Reg at base address
4: while list of BFs not empty do
5:     Pick first bitfield BF in BFs such that width(BF) ≤ AvailableWidth
6:     if Not Found then
7:         Allocate new Reg in RegisterTable
8:         AvailableWidth = BusWidth
9:     else
10:        AvailableWidth = AvailableWidth - width(BF)
11:        Link BF to Reg
12:    end if
13: end while
14: for all Reg do
15:    for all BF do
16:        Assign bit position (right-justified) of BF in the register
17:    end for
18: end for
```

Algorithm 2 details the problem of minimizing the number of empty bits in each address, with the *first-fit decreasing* heuristic. The algorithm can be called with different lists of bitfields. Thus, the tool ends up applying the algorithm multiple times in the ***by type*** and the ***by assertion*** packing modes since they separate the bitfield types.

4.5.1 Experimental Results

The assertion checker, sequence monitoring and coverage packing algorithm has been run against a generated set of checker outputs and coverage registers. While the set of assertions used to carry tests on MBAC are comprehensive in their coverage of possible temporal statement syntax and keywords, it is too limited in its *number of assertions* to properly exercise the assertion packing algorithms.

The packing algorithm does not depend on the complexity of the checker's expression. This complexity being abstracted out by the temporal automata, only the resulting output of the checker matters from the register generation perspective. The performance of the packing algorithm under various scenarios and packing modes is thus derived from a *generated* set of checkers, similar to constrained random-based verification when applied to a digital circuit. The use of randomly generated bitfields, constrained to emulate the typical distribution given by conventional checkers simplifies the testing of the packing routines by providing a larger data set.

A scenario is made up of a set of checkers, each having a set of control fields (between 2 and 5 control points), a set of coverage points (1 to 4 counters, each between 1 and the 32-bit of width) and finally a set of status flags (1 to 4 status bits, for the checker output and potential debug enhancements flags). As a result, some checkers will have bigger counters and a higher number of control and output flags, while others will have only a few control points and a few status bits. An histogram of the distribution of checkers used to create the largest scenario is shown in Figure 4.6

In order to show the algorithm behavior more consistently, the scenarios are setup such that the number of bitfields is monotonically increasing as checkers are added. Each scenario is thus a subset of the final 1000 checkers scenario.

Such scenarios provides ample variations between assertions checkers bitfield organization and better reflect how the packing algorithm will perform. The packing algorithm is tested with up to 1000 checkers.

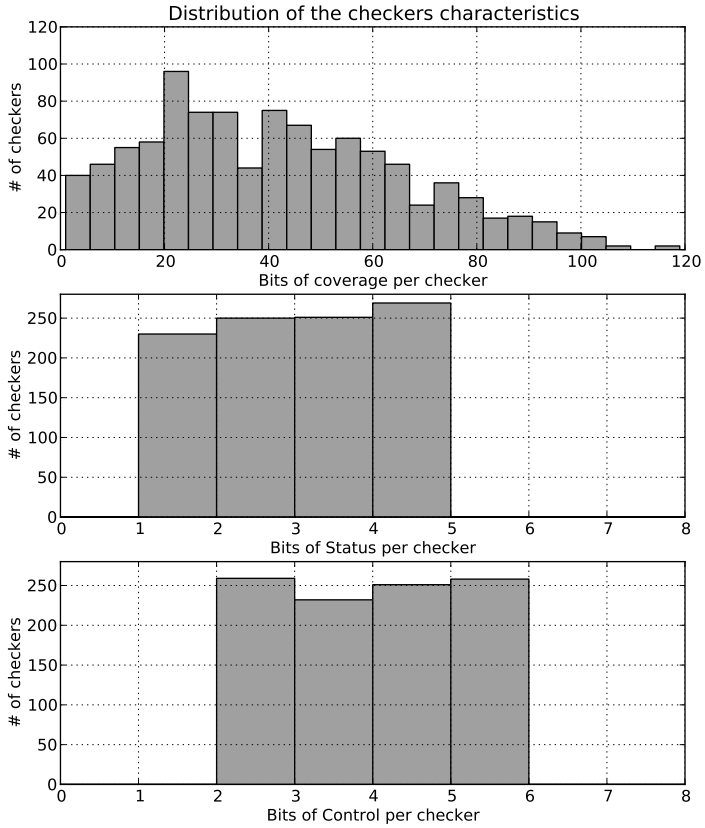


Figure 4.6: Distribution of the number of bits per checker for the *Coverage*, *Control* and *Status* bitfields.

4.5.1.1 Algorithm Execution Time

The register packing problem is run on the various scenarios. A set of checkers subject to the constraints explained above are generated, then packed with the three different methods presented Section 4.5. Figure 4.7 illustrates the effect on the runtime of the packing process. By specifically excluding the generation of the scenario and post-processing computations, factors non-related to the algorithm execution are excluded.

The packing process execution time is measured on an Intel Q6600 workstation operating at 2.40 GHz and running Python 2.6.5 (compiled with GCC 4.4.3) with a

data set size varying between 1 and 1000 checkers. Note that Figure 4.7 reports the execution time versus checker count on a log-log graph.

The *By Assertion* packing mode only has to pack a few bitfields in a small set of registers for *each checker*, thus the complexity of the problem is significantly reduced and mainly proportional to the number of checkers.

A notable observation is that for a set of 1000 checkers (8590 bitfields in this particular scenario), the packing time is just above one second. It is important to note that 1000 checkers represents a very large library and would represent an upper bound on what a practical design block would need in terms of memory mapped checkers. In designs containing more checkers, it is unlikely that they would be packed in the same entity (sharing the same address decoder logic) for physical reasons. This confirms that the comparatively slower execution time of Python will not be an issue.

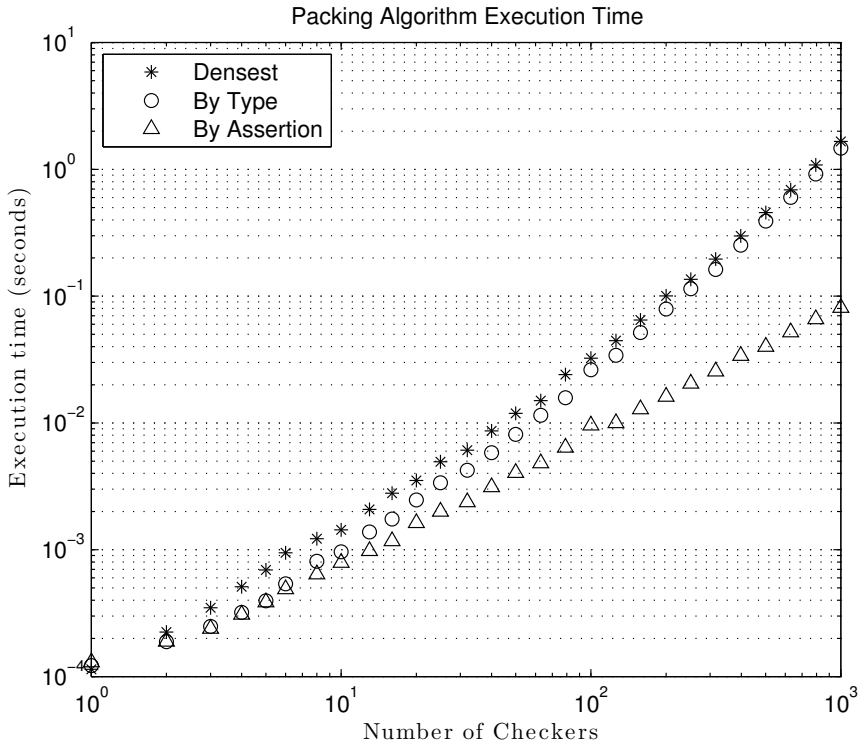


Figure 4.7: Execution time of the packing routine when subjected to the ***Densest***, ***By Type*** and ***By Assertion*** packing modes. The scenario covers from 1 checker (11 bitfields) to 1000 checkers (8590 bitfields)

4.5.1.2 Register Usage

Figure 4.8 reports the average number of registers required to hold the checkers in a scenario, for the three packing modes. One can observe that pre-classifying registers by type end up using only a few more registers in total when a lot of assertions are packed together. Clearly, the *By Assertion* packing mode ends up taking the most registers since it only allows bitfields related to the same checker to be put in the same registers. In addition, this packing mode segregates the control, status and coverage fields in their own set of registers, allocating a minimun of 3 registers per checker.

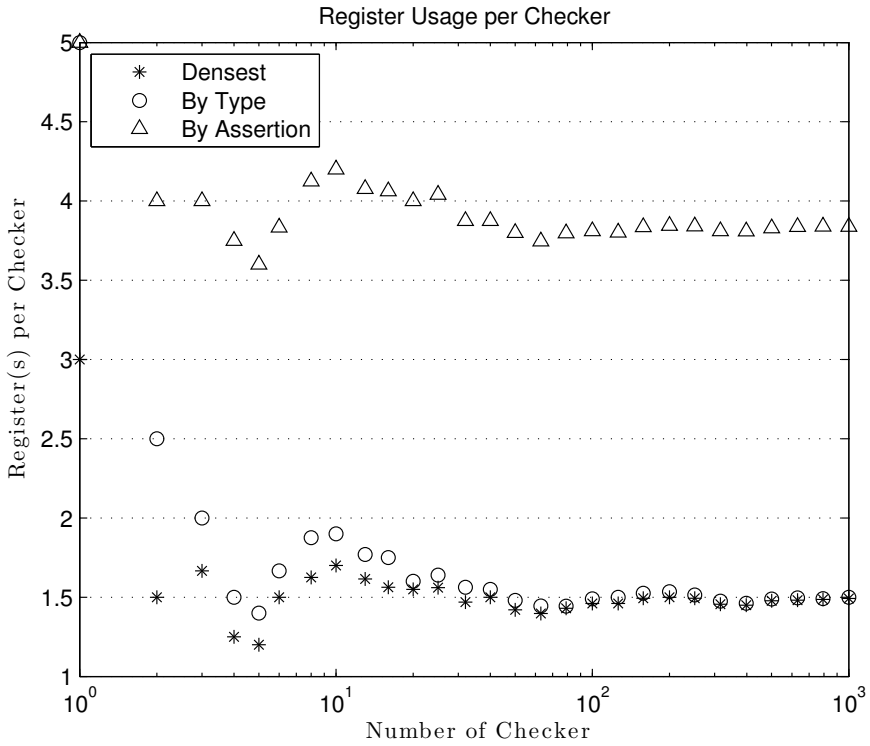


Figure 4.8: Average number of registers used per checker for each scenario from 1 checker to 1000 checkers.

4.5.1.3 Unused Bits in Registers

Figure 4.9 reports the total number of unused bits after the packing is performed. Unused bits have few consequences in terms of hardware resources (only a higher decoder overhead), but bring performance limitations when the checkers are integrated with the firmware or when one considers the resulting bus utilisation in the NoC utilisation presented in Section 5.4.2.1. The ***Densest*** packing mode can insert control and status bits between coverage counters, thus this packing mode result in the least number of unused bits after the algorithm completes.

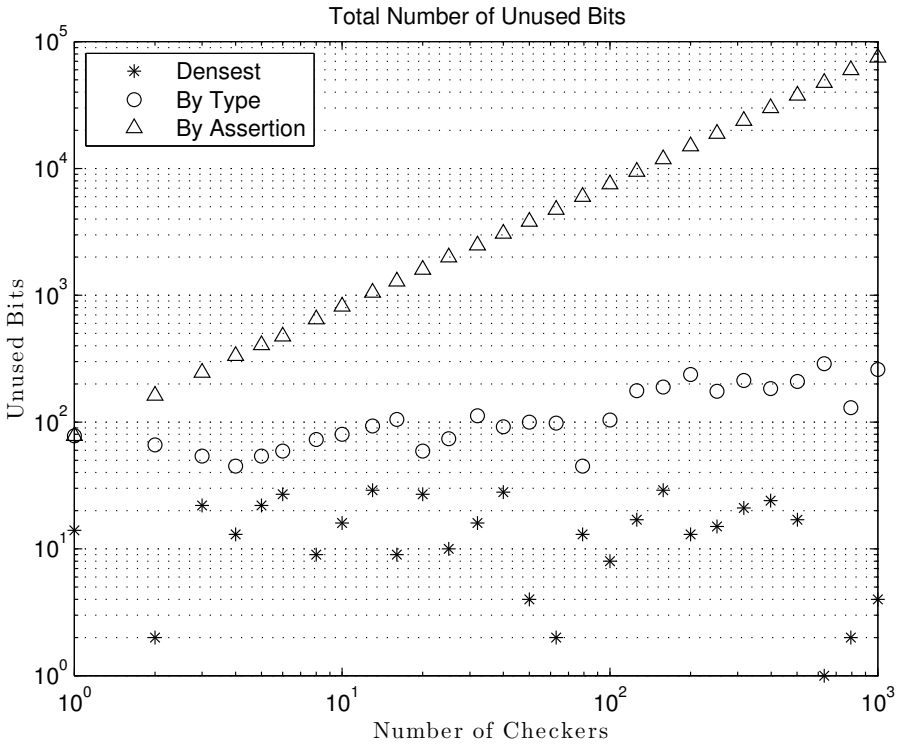


Figure 4.9: Unused bits left in the memory map after the packing process.

4.6 Operating System Integration

As covered previously, to extract the full benefits of the hardware checkers at the system level, they have to be supported by a firmware layer that can inter-

pret the various hardware bits and their more abstract meaning along with correlate events to provide a high-level "health report" on the device operation. The firmware thus has to be an integral part of the solution.

Systems that are complex enough to require hardware assisted assertion checkers are also likely to be using a modern operating system. In most cases, these will support virtual memory and user processes. The research carried on the integration of hardware assertion checkers focused on the well-know GNU/Linux operating system. The main factors in selecting this operating system is its suitability for integration in medium to large embedded system, the unfettered availability to the source code and documentation along with a very large and active developer community. GNU/Linux can also run on most modern processors making it suitable for integration with hard-IP CPU core FPGAs and prototyping on a personal computer. GNU/Linux makes use of virtual memory and separates user code from kernel code. This way, processes are isolated from each other and the virtual memory layer helps to build more stable systems for times when a lot of processes run simultaneously. This operating system is thus representative of what one can expect to run on a large SoC or NoC used in future applications.

4.6.1 Kernel Space and User Space

In Linux, code running as part of the OS and using physical (hardware) addresses is called *kernel space* code and in contrast, the code that runs in virtual address space is referred to as *user space* code. The distinction is important. Kernel space code has full, raw access to the hardware resources (in our case, the register file of the hardware checkers). However, the supporting libraries are limited. The kernel code is written in C and assembly and has access to only a small subset of libraries. As most of the code running in a GNU/Linux system is in user space, advanced libraries, scripting languages and database engines are all accessible only in user space.

In almost all cases, the designers will want an *Application Programming Interface* (API) to access the hardware devices. Instead of directly manipulating addresses and bits, an API will abstract away and make the hardware locations. This is not different when integrating hardware checkers. Some form of *device driver* has to be

implemented. However, writing a full-featured *kernel-based device driver* running inside the OS address space (physical memory) is quite complex and requires a significant development effort.

Leveraging the extensive user space libraries from kernel space is possible, but necessitates building two separate code structures, one residing in the kernel space and one in user space. This adds development effort due to the need to code the communication mechanism required to pass the data back and forth between the two domains.

To address the complexity problem, this section suggests moving most of the data manipulation in user space to simplify the integration and leverages on an existing kernel module called *User space I/O* (UIO) (detailed in Section 4.6.3) to assist in this process.

4.6.2 Prototyping Environment

In order to understand and evaluate the suitability of the proposed approach, the integration was prototyped on a re-programmable system running a customized Linux kernel. The BEE2 [2] prototyping system was selected as the platform. This hardware was generously provided to our laboratory by the Canadian Microelectronics Corporation. The BEE2 system offers what amounts to an infinitely re-programmable machine with the capabilities to emulate a vast array of different computer architectures. The BEE2 consists of 5 large Xilinx Virtex-II Pro FPGAs attached to a single board with multiple dedicated memory interfaces as detailed in Figure 2.7. The system also includes a networking interface and a centralized configuration manager IC that can load the individual FPGA configurations from a central CompactFlash card. Through a mechanism called SelectMAP, in-system, dynamic reprogramming of the user FPGAs is also possible. This driver was ported from the BORPH project (kernel 2.4) to the latest 2.6 Linux kernel at the time, allowing the control FPGA to re-program very rapidly (in less than a second), from the command-line, the 4 large user FPGAs provided by the BEE2. A notable feature of the Virtex-II Pro FPGAs is the presence of two PowerPC 405 hard IP CPU cores embedded in the FPGA fabric. By creating a SoC architecture that uses a PowerPC core along with a bank of DRAM memory, one can run one

(or many) instances of a modern OS.

The BEE2 proved to be a very good fit for the research on NoC architectures and also allowed system-level prototyping of hardware checker integration. The ease of re-programming the FPGA along with the presence of a network interface allowed the system to be highly flexible. Instead of hosting the kernel and user space code directly on the BEE2, the network interface was used to tether the BEE2 to a PC workstation running Linux.

Porting the U-Boot [5] system loader (similar to a PC BIOS) such that it could support the network interface of the BEE2 and allow the Linux kernel to be downloaded from the workstation on boot was addressed first. This allowed rapid updates to the kernel, an essential feature for efficient prototyping with a mix of kernel development and hardware.

By using another mechanism supported by Linux called a Networked File System (NFS), the whole user-space storage was effectively a remote file system served through a PC workstation. Using this NFS-mounted root file system allowed rapid modifications to the software image since any updates (PowerPC cross-compilation outputs) performed on the workstation were immediately reflected on the embedded system image, ready to be tried out on the PowerPC architecture. This method is also used with networked embedded systems that are lacking the required storage [22] space during prototyping and debug. Another benefit of this approach to assist debugging is the ability to store very large log files since they effectively reside on the workstation's *hard disk* and not on the embedded system.

The prototyping software environment, derived in part from the BEE2 BORPH Linux [93], from the ML-310 development platform by Xilinx and from Secret Lab's [6] open source device drivers allowed rapid integration of the multiple software modules required to bring the system up and running on the Linux 2.6 kernel. The source files and set of drivers were combined to allow the use of the various peripherals that were programmed in the control FPGA of the BEE2. Notable features of the resulting prototyping environment include:

– Flexible booting architecture including CompactFlash drivers for the kernel and network boot support.

5. http://www.denx.de/wiki/U-Boot
6. http://www.secretlab.ca

- Ability to re-program any of the *User FPGAs* by running a ***cat bitfile.dat > fpga<n>*** command at the Linux prompt.
- Support for read/write to the physical compact flash device allowing the Control FPGA bitstream to be re-programmed by the control FPGA itself (the changes would take effect after a power cycle).
- Two root file systems support : Busybox [7] and ELDK [8].
- Cross-compiled Python support with ***mmap()*** library for UIO experimentation. This allowed command-line scripting of hardware checker register access to experiment with dynamic debugging scenarios.
- Secure shell support (via the networking) for multi-terminal sessions to the BEE2 system allowing multiple processes to be simultaneously launched and controlled.
- Custom designed power control box designed with a solid-state relay allowing full remote power control on the BEE2 machine through the workstation and a small shell script.

This prototyping environment setup proved to be a very efficient way to experiment with various hardware and software combination, but also a very powerful mechanism to locate and fix bugs in drivers. The Busybox root file system offered fewer user space libraries, but would boot very rapidly (in less than 5 seconds) which allowed a very rapid cycle of tests in the event of kernel crashes. The more full-featured ELDK root file system supported many libraries, including database support (SQLite) and the Python programming language.

As a case in point to show the high flexibility of the prototyping environment and its efficiency for debugging, a subtle corruption bug due to incorrect flagging of non-cached pages was uncovered in the UIO driver present in the latest Linux kernel sources at the time (unknown to the community). By using a combination of hardware and software-based traces, the bug was corrected and the patch submitted to the kernel maintainer for integration in future releases of Linux [9]. This

7. http://www.busybox.net/
8. http://www.denx.de/wiki/DULG/ELDK
9. Interestingly, submitting this patch to fix the UIO drivers for the PowerPC architecture showed how efficient and streamlined the Linux OS bug reporting and fixing process is. It only took a few days for the patch to be integrated in the PowerPC kernel source tree and within the next release, it was merged into the mainline kernel. This patch fixed problems on other Motorola

patch is now part of the mainline Linux kernel, so the technique presented for assertion checkers hardware interfacing should work for the foreseeable future, no matter the architecture used.

Bugs in the kernel space are notoriously difficult to track and fix since they can cause corruption and hardware lock-up requiring the whole system to be restarted and the debug process re-started. Having a flexible development environment with support for multiple versions of kernel and user file systems proved to be an important factor in successfully tackling the most complex integration bugs.

4.6.3 UIO Kernel Module Details

In early 2007, a new kernel module called UIO was added to the Linux 2.6 kernel (2.6.23). This new module caters to users who want to control a hardware, memory-mapped device through its registers, but that do not require special handling from the Linux kernel itself. For example, a simple I/O card, or a specialized FPGA-based co-processor often only requires memory mapping and read/write access to those memory-mapped registers from the user application point of view. In contrast, specialized drivers such as networking or USB have to be adapted to the pre-existing kernel code infrastructure as they require extensive support from the kernel libraries. The UIO module handles best those hardware devices that require a direct memory-mapped interface to perform their intended function, but that do not benefit from the special handling provided by the kernel code. Memory-mapped assertion checkers and coverage counters directly fall into this category.

The UIO project goal is simply stated in the *osasdl.org* web site as follows:

"Provide a generic framework for handling devices in userspace. The device driver is split into a small kernel part, which contains the device setup and the primary interrupt handling, and a user space part, which handles the device functionality."

In short, this means that a small part of the driver has to reside in kernel space (the one performing the memory allocation reservation and device node bindings)

processors such as the MPC5200 and sparked many discussions since not all architectures handle page caching in the same manner. For the patch details, the commit hash starts with c9698d6b1a and can be looked up in kernel.org.

and the rest resides in user space. To help with the research, the implementation of the user UIO module can take parameters to customize the UIO module at load time such that one can specify which base address and which range of memory is going to be remapped. This way, the kernel module is more flexible and enables more rapid prototyping by eliminating the need to re-compile de kernel module when accessing different memory maps. The driver also provides the option of reserving (locking) the memory range such that other drivers do not accidentally (or voluntarily) request the same memory range which could lead to race conditions or raise data security issues.

One of the major benefits of using user space drivers to monitor and control hardware checkers is that if the device hardware is accessed incorrectly, such as outside the bounds of the memory range dedicated to the checkers, the user program will be terminated by the kernel but the system will stay up and running without side effects. Working in userspace through the UIO mechanism thus makes the system more tolerant to programming mistakes.

Finally, a driver running in kernel space also has to be constantly maintained as new kernel revisions are released due to the API changes in the kernel during its evolution. In contrast, the UIO mechanism benefit from one more level of separation from the kernel API and is subject to a much lower rate of change, reducing the maintenance burden. From the user-space program point of view, only major changes such as going from a 32-bit to 64-bit architecture would require maintenance.

4.6.4 UIO Driver structure

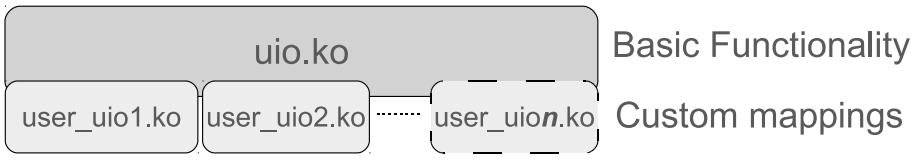


Figure 4.10: Userspace IO Driver Organization

In the Linux kernel, the base UIO driver (uio.ko), will take care of providing the basic UIO support layer. This part of the driver is maintained with the Linux

kernel and updated at each release to follow internal kernel API modifications. This module performs the management of the virtual *files* that are created in the **sysfs** virtual file system and similar supporting function that are typically required from regular in-kernel drivers. As illustrated in Figure 4.10, one can dynamically load user drivers. The user driver mainly provides information about the memory maps such as the base address, size of the memory map, a representative name and provide some parameters related to access type (exclusive, caching mechanism).

This approach has the advantage that most of the complexity of writing the device driver and attaching the filesystem hooks (to have it appear in /dev, /sys) are handled by the base UIO driver. The only thing left is define the memory maps and register them via the user driver part of UIO.

The end result of loading the user UIO module is the ability to access the mapped memory range from a new device entry residing in the /dev folder that provides direct access to the hardware memory. In a properly set up Linux system, loading the user UIO module will trigger an event with **udev** and a new /dev/uio**x** entry will appear in the filesystem (for example /dev/uio0).

4.6.5 UIO Operation and Register File Access

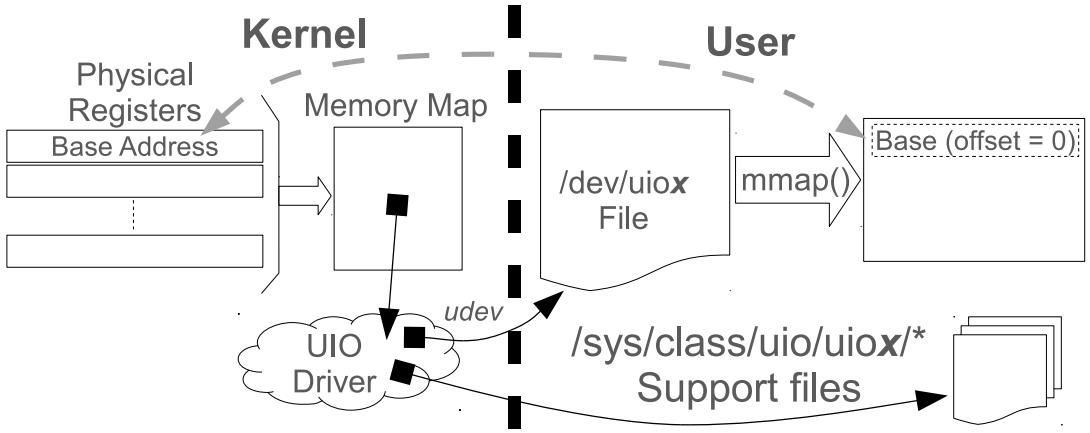


Figure 4.11: Userspace IO Register Mapping

The UIO user driver used for this research has expanded the example code that is part of the Linux kernel. The driver was modified to accept base address and

region locking parameter when loading the driver into the kernel. One can thus use the UIO device (/dev/uiox) and have it mapped to a parameterizable location in memory. Usually, this location will be set to point at the base of the device's address space of interest. In our case this was the hardware checker control and status information that was generated from the previously discussed register interface generator. From this point on, the userspace program will "see" the registers of the hardware unit from offset 0x0 of the virtual file. Thus the offset values for registers can directly be correlated with the datasheet of the IP hardware checkers generated previously.

4.6.5.1 UIO Module Versus Full Physical Memory Access

Users familiar with the details of the Linux operating system know that the special device file /dev/mem already allows memory mapping or hardware register in user space. It also allows it to be done without any special kernel module. However, this raw access mechanism to physical memory has many implications in larger systems and cannot realistically be considered a scalable solution. The single /dev/mem virtual file enables the reader to access anywhere in the system memory. Though quite useful to observe memory in a *live* system, this mechanism bypasses any memory protection and permissions. In a multi-user system, giving one user read or write access to /dev/mem breaks all the built-in permission infrastructure and leaves the system extremely vulnerable to attacks and breaches of information. Any keystrokes typed by any user in the system end up somewhere in the RAM memory and thus could then be observed by another user through this mechanism. In general, only the superuser has access to /dev/mem and its permissions are set to be very restrictive [10].

In contrast to the unbridled access to physical memory of /dev/mem, the UIO mechanism offers fine and granular permission support. Therefore, hardware checkers aimed to be user-accessible could be defined with specific permissions on the /dev/uiox file associated with the checkers' memory map, such that only those with the privilege can read or control the checkers. In such cases, the users

10. Some of the security concerns of /dev/mem are being addressed in the kernel from 2.6.27 through the CONFIG_STRICT_DEVMEM compilation option, including limiting the direct access to the RAM resources.

(most likely a daemon process) being granted access to this specific region of hardware memory still has a very restricted window on the physical memory. Furthermore, the user program does not require any information on the physical memory location of the checkers since this is handled when the module is loaded in the kernel. Therefore, by using all the built-in mechanisms for file permissions present in GNU/Linux, UIO offers a scalable, secure solution for the large systems envisioned in the future, yet provides a quick and efficient infrastructure for research and prototyping on memory-mapped hardware interfaces.

4.6.5.2 Software Interface to UIO

The use of the mmap() system call is required on the /dev/uiox file to access the physical registers. The mechanisms involved require the *Memory Management Unit* (MMU) to perform the translation from the virtual memory of the user space to the physical addresses of the system. Since this translation is universal in all MMU-based systems, it is very fast and extremely optimized, leading to little loss of performance. After the call to mmap(), the pointer provided by the call can simply be read or written and this will result in a direct access to the underlying hardware registers.

The Python scripting language also supports the mmap() call, meaning that the registers could be accessed *interactively* during a debug session. In conjunction with the database of assertion and sequence checkers (Python offers libraries that interface to almost every databases format), the scripting interface can be used to "query" the device interactively from the Python prompt. In the support of interactive debugging scenario, only a set of supporting libraries (database interface, memory map location information) would be required to support the methodology. The debug tool interface can be the Python interpreter itself.

Python does suffer from performance limitations, especially if registers have to be read in rapid sequences or if a lot of data is being generated by the hardware. A combination of an optimized C driver linked to the Python interpreter can address those concerns.

4.6.6 Estimating the development effort saved by using UIO

The development effort of the UIO driver can be estimated using the COCOMO [114] basic model and a utility called *sloccount* by David A. Wheeler [11]. By analyzing the number of lines of source code needed for a given piece of software, along with a set of metrics gathered from analyzing various projects, the COCOMO model can provide an estimate of the work required to produce a body of source code. This is a rough estimate, but nevertheless, it can help evaluate how much time can be saved by using UIO as an alternative to coding a full kernel driver with a similar set of features.

	uio.c	remap-range.c
Source Lines of code	702	80
Development Effort Estimate (Person-Month)	2.02	0.18
Memory Utilisation on the BEE2 (bytes)	13720	4328

Table 4.1: Comparison of source code and module complexity between the base UIO driver and a derived user level driver. Memory utilisation measured on the BEE2 PowerPC kernel version: `2.6.24-rc5-xlnx-jsc-xlnx-nfs-g669cb9c0` (note that this version is slightly older than the one presented in the CMC demonstration)

Using the *semidetached* COCOMO model parameters (to account for the more difficult task of writing kernel code) and running it on the uio.c sources of kernel 2.6.37-rc2 we can obtain a measure of its complexity. Since the uio.c source code contains the necessary procedure calls to support a well-behaved driver that will properly translate the user space to kernel space system calls, driver registration and other maintenance routines, it relieves the designer from having to support this boilerplate code and focus on the interesting part, namely the hardware register access. The same *sloccount* routine is executed on the *remap-range.c*, an example UIO driver provided by the author as an application note for the Canadian Microelectronics Corporation (the source is provided in Appendix A.1). The result of the comparison is summarized in Table 4.1. The remap-range.c driver, if coded to sup-

11. `http://www.dwheeler.com/sloccount/`

port all the features offered by the uio.c code, would represent approximately the same complexity as uio.c itself. Furthermore, the structure of the remap-range.c driver is amendable to template-based code generation and thus fit well in the proposed methodology.

The skills and experience of the programmer can make those estimates vary significantly, but it remains that the creation of the user driver of UIO is quite simple and straightforward, saving the designer a significant amount of development effort.

Finally, another benefit is that the UIO driver is maintained as part of the operating system and does not need to be adapted at every kernel revision. The abstraction layer provided by the de-coupling from the kernel source dependencies will further minimize long term maintenance efforts.

4.6.7 Limitations of UIO

One limitation of UIO in its current embodiment would be the size of the memory map. UIO has a small granularity (working in multiple of kernel pages of 4kB). In most assertion checker applications, a few pages of 4kB would be sufficient since one page can hold 32 thousands assertion checker outputs. Usually, a checker group (re-programmable cluster of assertion checkers and sequence monitors) would map to a few pages of kernel space. As checker groups are distributed within the physical system, their memory map would also reflect the hardware topology since address decoding is better performed in a hierarchy to speed it up and re-use the comparator circuits.

UIO opens up a few memory address pages in the userspace so that program can entirely take over the hardware interfaces. It does limit the use of hardware-based features like DMA and interrupts (only partially supported by UIO), but otherwise, frees up the software to manipulate the hardware at will. One can relate this to smaller *embedded systems* in which the software has direct control of the registers.

There is no absolute limit on how many UIO drivers can exist on a given system and the number of memory maps per UIO device is a compile-time option of the driver which can be adjusted at the expense of higher RAM utilisation by the base

driver. The /dev entries themselves have numbered names (uio0, uio1, uio2), but that can be changed to give them meaningful names (e.g. **/dev/unita_assertions**, **/dev/unitb_checkers**). Furthermore, the driver can detail the memory map giving it meaningful static information in such a way that the drivers can be dynamically located in the sysfs virtual filesystem (**/sys/***) [21].

Choi et al. have proposed to generate the device driver based on a set of XML specifications [115] which offers a different framework from the proposed UIO-based system. Their approach is more suitable for specific drivers that benefit from full ***ioctl()*** call flexibility, specialized ***/sys*** interface control points or where the interface deals with FIFO buffers or similar streaming hardware units. The UIO driver may not be the best solution in those specific cases. Generating drivers using the approach proposed by these authors appear to require more development effort, but could offer better support for more uncommon type of hardware interfaces.

4.7 Chapter Summary

This chapter provided a method to package assertions and sequence checkers in a scalable representation amendable to large-scale automated integration. Opening up the visibility on the assertion checking mechanisms to the user software in complex systems enables many advanced bug analysis methods to be carried on the available information. Exporting internal hardware status to the higher-level applications also enables remote diagnostics and other uses of the internal checkers as means to increase reliability and data security.

Chapter 5

Integration of Checkers in a NoC

5.1 Overview

This chapter proposes a generalized extension to the memory mapping concept of hardware checkers when applied to envisioned future NoC-based systems. The underlying register-grouping algorithms that aggregate the checkers in a memory map are still used, but hardware mechanisms are added to autonomously scan the status information of the memory mapped checkers and propagate those to a separate module for aggregation. The resulting NoC-based system will end up having distributed hardware checkers capable of autonomously *centralizing* failures in addition to supporting the local CPU-based register mapping.

Since NoC-based systems are built by repeating (tiling) similar hardware structures (router unit, processor interface), hardware checkers could end up replicated many times in a NoC. The replication of all those checkers offers great support for debug. However, the added cost (silicon area and power) may not always be justifiable.

To address this new problem, a method based on computed metrics derived from the size overhead of the checkers along with an assessment of their debug value will assist the engineers in their selection and placement of hardware checkers in those types of distributed systems. The proposed approach aims at reducing the overhead while maintaining an acceptable cost/benefit compromise.

This proposed design methodology was designated as the *Test, Monitoring and Debug (TMD) infrastructure*. In an ASIC design flow process, one can expect it to result in a reduction in the total cost of the infrastructure, yet maintain a quantifiable measure of the final device support for debug. Some of the structures are also applicable to monitoring the operation of the system and can also be used in the chip testing process, hence the reasons behind the TMD naming of the method.

The term *Quality of Design* (QoD) is proposed as a way to evaluate the value of hardware checkers positioned within the NoC-based system based on their effectiveness for debugging. Fully characterizing the benefits to the debugging process and producing accurate metrics of the QoD is a significant undertaking and requires multiple silicon iterations, field failures and many debug sessions to produce the factors.

The approach is thus proposed from a design methodology point of view, with experimental results gathered from the ASIC synthesis of a RTL-level model representative of a sample checker and trace monitor. This chapter attempts to objectively measure the worth of dedicated circuits in terms of their test, debug and monitoring contributions to a system.

5.2 An Overview of Networks-on-Chip

The progression of silicon technology has allowed engineers to build systems with increasing levels of complexity in each generation. As discussed in Section 2.1.5, at a certain integration level, the NoC [36, 37] paradigm becomes appealing, but also confronts designers with new challenges, especially in ensuring reliable and failure-free operation when those devices are deployed in the field. While application software errors occurring during the system operation above the NoC hardware layer can be, to a large extent, dealt with by the system, communication failures inside the NoC can often be catastrophic. In an analogy to traditional computer systems, a *double bus fault error* is often an unrecoverable failure, leading to system downtime, while other higher-level failures are correctable and the system can recover.

During their operation, systems employing NoCs can experience errors due to

a multitude of conditions such as single-event upsets [116], faults due to untested or unverified corner cases or network deadlocks/livelocks. Thus, it becomes important to detect imminent failures or errors within the IC and to be able to address their effects whenever possible. The detection must be carried out as close as possible to the source of the failure such that any inconsistencies can be reported as early and accurately as possible. This capability facilitates the diagnostics of the problem and may provide a solution to circumvent it and possibly recover from it, for example by re-initializing a part of the device and re-starting the process.

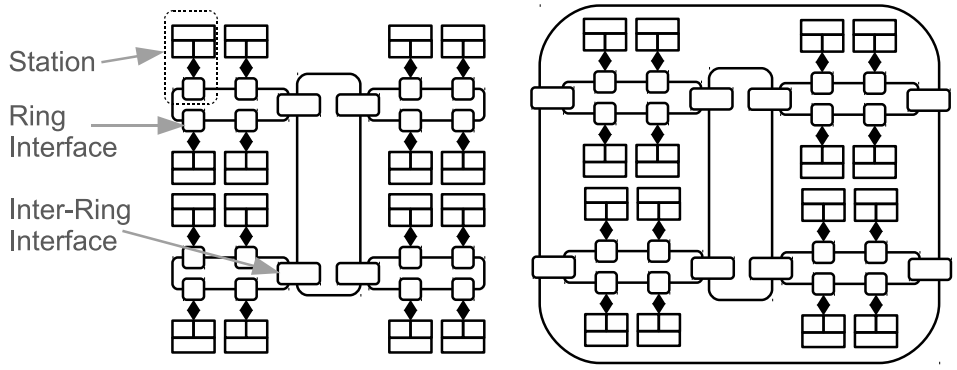
In a NoC, a single error may also trigger a slew of consequent errors in a way similar to alarm showers in supervisory environments [117]. Furthermore, the circuit may operate on several clock domains, a feature required to lower the power consumption through the use of frequency and voltage scaling techniques. Thus, errors detected by hardware checkers can be reported in an unpredictable order, giving the wrong impression of the original cause for the *shower* of errors. Having information on the temporal sequence of errors will simplify diagnostics by facilitating the identification of the root cause of the failure.

5.2.1 Debugging Network-on-Chip

Different methods have been proposed to replace the functionality of a logic analyzer with more powerful on-chip alternatives. The IBM team that worked on the Cell microprocessor [118] is a prime example of this effort. They chose to include on-chip a parallel bus architecture to specifically route debug information packets.

While attractive, this approach adds a significant amount of area and wiring. Ciordas [119, 120] and a team at Philips research labs have studied the reuse of the *existing network* for purposes of debugging and monitoring targeting specifically their *Æthereal* NoC platform and its trade-offs, and found that re-use of networking resources saves a large silicon area compared to a dedicated debug bus. Other efforts [121] in the *testing* field have considered the re-use of the on-chip network, but did not consider their real-time debugging uses.

5.3 Experimental Context



(a) Hierarchical-ring Architecture **(b)** Hyper-ring Architecture

Figure 5.1: Variations on a hierarchical-ring NoC architecture. The hyper-ring adds a secondary path for data at the global level. Refer to Figure 2.4b to view the details of a station.

Our work focuses mainly on a few specific topologies of NoC, namely the two-level hierarchical-ring [15] architecture and some of its variants. Figure 5.1a illustrates the basic topology along with the hyper-ring of Figure 5.1b, which adds a redundant path for the traffic at the global level. The NoC is built from the following key components:

- ***Processing Element*** **(PE)** . This represents a CPU along with some instruction and data cache memory. It may also be attached, via the local bus, to a dedicated memory controller allowing a Non-Uniform Memory Architecture (NUMA) to be built.
- ***Ring Interface*** **(RI)** . This is the bridge between the CPU local bus (which can be Wishbone, AMBA or CoreConnect, for example) and the high-speed, on-chip network. The Ring Interface can be associated with a *network adapter* if one wishes to use a workstation analogy.
- **Station**. A station is the combination of the Processing Element and Ring Interface.
- **Local Ring**. This is the first level of interconnection between Processing Elements. This local ring allows low-latency communication between the Processing Elements situated at this level. The traffic between those stations will

not affect other local rings.

- **Global Ring**. This is the second level, global data exchange medium. It allows messages to pass between Stations that are not on the same Local Ring.
- ***Inter-Ring Interface*** **(IRI)** . This structure allows the messages to be passed on to the Global Ring from the Local Ring when required, based on the addressing information.

Our prototyping architecture proposed four low-level *local* rings attached to one central high-level *global* ring. Every flit (flow control digit, the smallest unit of data between stations) passing on the ring contains information about the sender's as well as the target's address ring and station. The encoding was selected to be very explicit and optimized for speed and decoding simplicity. In doing so, the acheivable clock rate on an FPGA architecture was ahead of similar NoC architectures proposed at the time [15].

The rings are designed to account for the case in which each station is operating in a different clock domain. Therefore, most of the FIFO queues in the rings are asynchronous. Furthermore, all the rings are unidirectional, that is, flits sent by a station in any ring are not immediately accessible by the other stations with the exception of the next one in the directional ring order. This property ensures that data is received by the target nodes in the order they were sent within the same end-to-end link. This simplifies the routing and reduces the hardware overhead as no re-ordering buffers are needed.

The in-order arrival of flits is quite helpful in the context of debugging. In our case, however, a problem occurs when data must pass from one ring to another through the central, top-level ring. In that case, the temporal order of the data generation is lost in the network. Data traversing the network is buffered at each station; two flits generated at a specific time in two different rings might not attain a target node in an exact temporal order due to the differential propagation delay of the information. This delay is based on local congestion levels and relative clock domain frequencies in the network. All of this is further complicated by the dynamic clock scaling supported by our experimental platform. A timestamp mechanism at the architectural level has been investigated by colleages [16], but will not be covered in this thesis.

5.4 Distributed Hardware Checkers

In order to carry the methodology presented in the previous chapters in a NoC, a subset of the PSL statements are selected for hardware implementation following an analysis of their hardware cost overhead. These statements are then translated to synthesizable RTL by the checker generator. The assertion circuits are then mapped into CPU-accessible registers, as explained previously. During normal execution, the outputs of the checkers are monitored by local PE CPU resources in order to provide the *monitoring* part of the methodology. An eventual assertion failures would be used as a starting point for the debugging process as explained in Chapter 3.

5.4.1 Processor Control of Checkers

Checkers can be controlled by the local Station's processor for testing or online monitoring through one or more CPU-writable registers which can control *the values of parameters within the assertion checker*.

This contrasts to the method covered in Chapter 4 in which the CPU-accessible registers were dedicated to providing control and statue to the hardware checkers, but had no influence on the temporal logic (besides the reset of the automaton).

The approach presented here allows some CPU-accessible registers to modify the behavior of the checker's automata in real-time. In doing so, a layer of flexibility is added, namely that some CPU-programmable registers can manipulate the checker's function in a dynamic manner. This allows novel uses of checkers beyond validating internal static sequences. One example of this type of checker in a NoC context is flit (or worm) tracing. In this example, the checker has inputs that can be *programmed* by a CPU to track a particular set of flits throughout the system. They could represent a cache coherency message or any other specific protocol-related transaction usually present in a NoC. This monitor can be replicated in many parts of the NoC and thus provide means to obtain a distributed trace. As NoC-based systems use memory address space to offer a uniform programmer's view of the registers, a *single* CPU can monitor and control a *distributed set of checkers* via the built-in NoC memory mapping mechanism.

The following example shows how to leverage the expressive temporal logic possiblities of PSL.

5.4.1.1 Flit Tracer

A *flit* tracing module is written in PSL and some elements of the flit (for example the source and destination addresses) are left as inputs in the PSL code, thus they will be translated as primary inputs by the checker generator. Those signals are then attached to a CPU-controlled register bank allowing the modifications of parameters within the tracer.

A ***cover*** statement is written in PSL which automatically infers a CPU-accessible register in hardware, allowing the events in the flit tracer to be counted.

Example 4 A flit tracer written in PSL.

```
vunit IRI_north(InterRingIF) {
  default clock = (posedge NR_Clk_p);
  sequence NR_P1 = {
       NR_DIng_DataValid     == 1
    && NR_DIng_SrcGlobalRing == Reg_Src1Global
    && NR_DIng_SrcLocalRing  == Reg_Src1Local
    && NR_DIng_Data          == Reg_Src1Data
  };
  // NR_P2 and NR_P3 are defined similarly
  // but with different register inputs (Reg_)
  assert always NR_P1 |->
    eventually! {NR_P2; [*]; NR_P3};
  cover NR_P1;
  cover NR_P2;
  cover NR_P3;
}
```

Example 4 lists the code required to build this tracer. Once transformed into a hardware module by the tool flow, several interesting debug features are added to the station. The three *cover* statements allow 3 types of flits to be counted when they pass on an interface. They can be used as monitoring points within the station. The *assert ... eventually!* statement shows how to structure a PSL statement to monitor a particular trio of flits (programmable by the CPU interface) that are

required to pass on the interface in sequence, but that could be interleaved with traffic from other stations. The data bus content of the flit is also used as part of the sequence.

In this particular example, an *assertion failure* simply indicates that the three flits did not pass in the expected order, but not that the hardware is misbehaving in any way [1].

This tracer is simplified to keep the example concise and easy to follow. A more practical tracer would use a PSL *sequence* as the basis for its pre-condition check allowing it to monitor deeper in the protocol of the NoC packet since that would allow it to analyze *worm* headers (sequences of *flits*) spanning multiple clock cycles.

When synthesized for an Altera Stratix II architecture, that particular hardware checker uses 75 ALUTs and 9 flip-flops (exclusive of the hardware overhead of the register bank storing the parameters). Most of the ALUTs are used inside the automata to compare the 32-bit NR_DIng_Data bus with the CPU Registers. Interestingly, only a few flip-flops are required to define the temporal expression of the *assert* statement.

5.4.1.2 Distributed Flow Control Monitor

Example 5 PSL FIFO Flow Control Monitor

```
// default clock and reset removed to keep example concise
property StopBurst = always
  { LowThresh } |=>
  { [*2] ; ~BurstEn };
property FlowCtlDown = always
  { StopDown & DataValid }  |=>
  { [*2] ; ~DataValid };
assert StopBurst;
assert FlowCtlDown;
```

Example 5 illustrates two simple assertions that can be translated to hardware and are used to assess the proper behavior of the output FIFO in a given NoC

1. It could be re-written as a sequence that has an associated *cover* statement with similar results.

station.

The first property validates that when the FIFO threshold is getting low, the station stops bursting data and goes to an interleaved mode of transmission as a way to reduce the traffic rate (eventually blocking the sender through backpressure propagation). The second property validates the proper operation of the flow control itself. It states that when *StopDown* is asserted while the station is sending data, the station should stop sending within 2 clock cycles. In the NoC implementation, this delay was due to the pipeline effect of the upstream controller, taking some time to react to the StopDown signal that is being asserted. Finally, the complete assertion requires the sampling *clock* to be fully defined (the local clock for the FIFO) and may include *reset* conditions to abort the sequence checking.

Those two assertion statements, when compiled in hardware, will constantly monitor the station's FIFO behavior and latch the assertion failure within one clock cycle of its occurrence. In this specific case, an *assertion failure* is a *real* failure in the sense that it indicates a problem with the backpressure flow-control mechanism of the NoC. Such simple assertion statements have minimal hardware requirements and benefit from being present at every FIFO in a NoC. From experience gained in building a RTL hierarchical-ring NoC, a flow-control bug between NoC RIs result in very complex data corruption problems that are tough to pinpoint since the problem can originate from any relay point in the path and its symptoms manifest themselves much later when a given CPU receive an incorrect cache lines or corrupted message. The corruption problem would initially be flagged as a software bug and could result in many wasted hours trying to find the cause when the real culprit is the underlying NoC transport layer [2].

As a last point to consider, the flow-control problems monitored by the above PSL checkers would only manifest themselves when the NoC experiences heavy traffic scenarios (required to fill up the interconnect FIFOs). In the FPGA-based hierarchical-ring NoC that was developed, "heavy NoC traffic" represents a data throughput of approximately 6.4 Gbps [15] between *every* station in the design, in

2. Note that some proposed NoC architectures perform data integrity checks at the *worm* level to detect transport layer failures. The proposed method does not preclude the use of link-level data protection mechanisms. In the event that a data integrity failure is detected, the assertion checker log will certainly help to confirm if the error is due to a transient failure or a bug in the flow control, for example.

the worst case. With this level of in-system throughput, even the simple data flow control mechanisms presented cannot be practically analyzed externally to the device. To give an order of magnitude of the task at hand, a 16-station NoC, at full capacity, would need the `FlowCtlDown` property checked at rates up to 4 billion (16 checkers x 250 MHz checking speed) times per second. Since that property is very simple, it depends on only two single-bit signals, namely `StopDown` and `DataValid`. Supposing that one wishes to export those flow controls externally to the device to debug an internal problem, this would amount to exporting 32 wires with an uncompressed aggregate bandwidth of 8 gigabits per second (2 signals x 250 MHz operation x 16 stations egress FIFO). While possible with data compression and a hardware serializer, it is clear that those checkers are better off staying buried deep inside the silicon.

5.4.2 Propagation of Assertion Failures

Each group of assertion checkers within a station can be aggregated and monitored by a dedicated unit, central to the NoC. If one module detects a particular assertion failure, it records it and also propagates a special *management flit* (M-flit) within the NoC to relay the information to the station responsible for analyzing failures. For practical purposes, a M-flit is exactly like a regular flit, but has an extra bit present in its addressing field. Using the NoC routing structure to propagate assertion failures is much faster than relying on the local station CPU to handle the error via software-based mechanisms. Furthermore, as the results are centralized, a more accurate picture of assertion failure event chains can be obtained by recording the arrival of assertion failure m-flits from a single location. The assertions failure messages within the network are always associated with their source address (as part of the NoC transport layer), thus clearly identifying the hardware unit responsible for the message. In the event of a failure in the NoC data transport mechanism (interconnect), the assertion information can still be accessed through the local station CPU interface as it remains stored in the assertion checker's flip-flop, but may require more debugging effort to correlate with other assertion failures.

5.4.2.1 Assertion Flit Generation Mechanism

As explained previously, when an assertion checker automata detects a violation, it latches the event. By adding a second register that monitors the previous event latch with the current event latch, a digital difference detector can be created (***xor*** on the vectors followed by ***or_reduce***). The digital difference acts as the trigger for the following process. Even with a very large assertion status vector, the digital difference can be built with pipelining to perform the ***or_reduce*** operation to avoid affecting the critical timing path [3]. The overall system-level is not sensitive to the minute delay of detecting the assertion failure vector difference since it will also take some time to propagate it to the aggregation unit.

Once a scan of the assertion status failure table is initiated via the trigger, the hardware process prepares a *worm*, by first setting up up the header flit and specify the destination for the assertion failure. The destination is the *station* (or a dedicated NoC unit) that will perform the aggregation. The assertion status decoder table must have been generated in the ***By Type*** mode described in Section 4.5 to maximize the utilisation of the data bus as the hardware will simply scan the registers using a counter as the address pointer. In a way very similar to the CPU accessesing the register table, the assertion status table is read out word by word, building up the *worm* that will be propagated to the aggregator unit. The NoC routing mechanism is used, so from the *Station*'s point of view, the hardware assertion unit is just another source of NoC traffic.

A propagation problem can appear when attempting to debug a deadlocked NoC when the debug messages are carried by the networking infrastructure. Thus an extra bit is added to the M-flit bit enabling it to follow a bypass mechanism. This makes those flits propagate faster through the NoC around congested nodes. By virtue of its simplicity, this mechanism is also a lot easier to verify prior to the silicon release. This new type of flit was called a High Priority Flit (HPF). As can be seen in Figure 5.2 The HPFs effectively bypass all FIFOs and therefore can be lost during their transport if multiple errors occur simultaneously in the NoC (due to the usual output buffer requirements). As a result this architecture will transport

3. An hypothetical thousand bits wide assertion failure vector cannot complete an ***or_reduce*** operation within one clock cycle in a fast system. In addition, the difference detector is inherently a multi-cycle timing path since it does take many clock cycles to prepare and send the *worm*.

a certain number of HPFs before starting to drop information.

A more sophisticated NoC architecture may use *Virtual Channels*, and give a high priority channel to *assertion failure reporting flits*, effectively obtaining a similar deadlock bypass mechanism. Our NoC architecture does not use Virtual Channels mainly because they add significant hardware overhead in the controller and require larger on-chip memories.

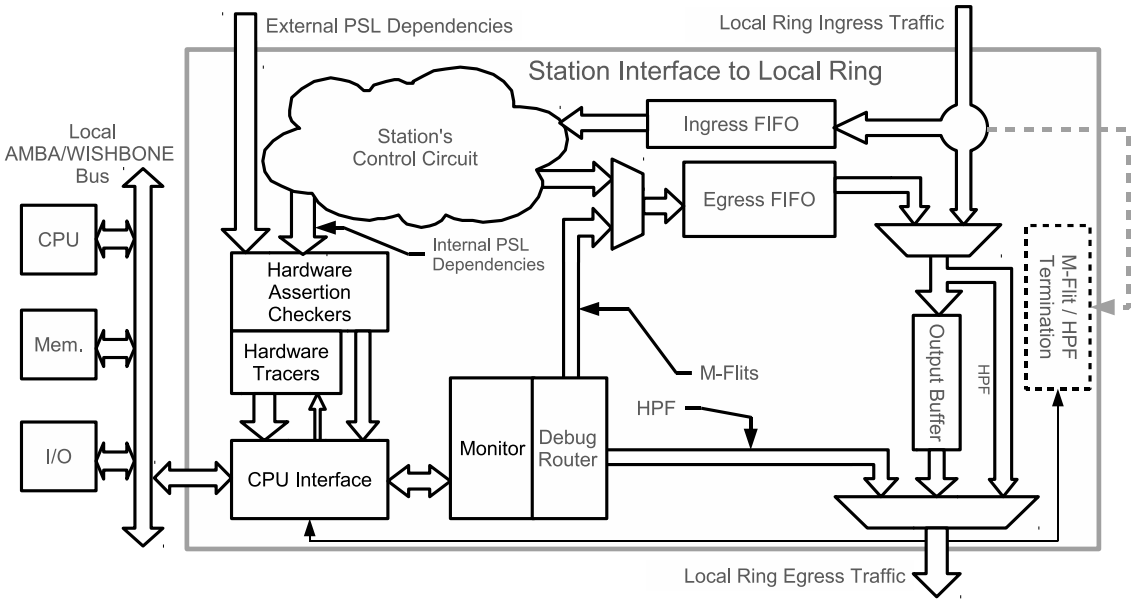


Figure 5.2: Detailed block diagram of the NoC Station showing the Assertion checkers in the Ingress/Egress Path providing protocol checking. Also illustrated are the two possible paths for the M-flits: via the egress FIFO or directly to the output multiplexer as High Priority Flits (HPF).

Our proposed architecture contrasts with the proposed assertion processor structure by Kakoee et al. [122]. In their architecture, assertion processors shift out serially the assertion failure(s) and propagate them between assertion processors in a way analogous to a scan chain. In this work, the assertions are memory mapped for in-system processing by localized firmware in addition to being transported by the NoC layer to a central aggregation unit.

5.5 Quality-driven Design Flow

One could think that improving quality through the addition of test, debug, and monitoring infrastructure can be achieved in a relatively straightforward manner, by distributing many checkers inside a device. However, even if future devices will provide considerable logic density, the repetition of large circuit structures like hardware checkers and their associated register interface can end up using too many resources when the same circuit is replicated multiple times. In addition, the physical location of the checkers within the NoC along with its topology play a key role in augmenting the value of this additional debug hardware. The checker's value is further enhanced if they can assist in the post-production *testing* of the device.

To address the complexity of selecting which checkers can be converted to silicon and their eventual physical location within the NoC, a quality-driven flow is proposed to maximise the debug benefits.

5.5.1 Major Considerations

Intuitively, all that is needed to improve design quality is to add the infrastructure IP and possibly add redundancy to compensate against failures. The difficulty lies in integrating the quality effort into the traditional design flow in a *systematic* approach during pre- and post-fabrication stages. In our view, there are two aspects of the design that improve the final quality:

1. *Verification* can be performed at the block and system levels. Block level verification checks that individual components conform to specifications, and system level verification checks that the components function correctly when interacting with each other.
2. A *Testing*, *monitoring*, and *debugging* (TMD) infrastructure:
   - The NoC reuse as a *test* access mechanism [85] is appealing, yet it poses its own challenges in guaranteeing the bandwidth and latency for streaming the tests to all subsystems. The communication bandwidth and processing capabilities of the PE are needed to predictably route test data, but possibly also for the self-tests and on-line *testing*.

- Runtime *monitoring* includes the ability of the system to detect error conditions. Those error conditions might be caused by unverified corner cases or by the presence of a timing fault or silicon defect that might have escaped initial testing.
- Integrated *debugging* hardware enables the system to diagnose the cause of errors, and to react appropriately (e.g. re-route traffic around a faulty node). The added visibility gained by the presence of the debugging modules will facilitate the localization of the root cause of problems during both pre- and post-fabrication debug.

5.5.2 The Test, Monitoring and Debug Flow

Figure 5.3 illustrates the design flow when high-level verification and a *Test, Monitoring and Debug* (TMD) infrastructure is added.

Typically, system-level verification is performed *after* block-level verification. So, in addition to verification typically being performed late in the development process, system-level verification is usually performed *last*. This fact has profound implications for NoC development because of the paradigm shift from *computation centric* to *communication centric* design [37]. The performance of current SoCs is mainly limited by the computational cores, and so the system interconnect can be selected almost as an afterthought. Conversely, the large SoCs and NoCs of the future will be limited by the available communication bandwidth. The bandwidth utilisation can be very dynamic and performance degrades very rapidly when the interconnect saturates. This makes the design of the interconnect architecture critically important. If system-level verification is performed at late stages of development, problems with a chosen interconnect architecture will have significant schedule impacts. It is therefore necessary to start system-level verification as early as possible and to include it in the architectural exploration cycle. Furthermore, verification components can be reused and refined during the implementation phases for regression testing. To gain proper execution speed of the model, higher-level, more abstract simulations are needed and can be done in SystemC. This will result in a better understanding of the effects of the NoC topology on the application performance. This exploration phase also helps clarify which moni-

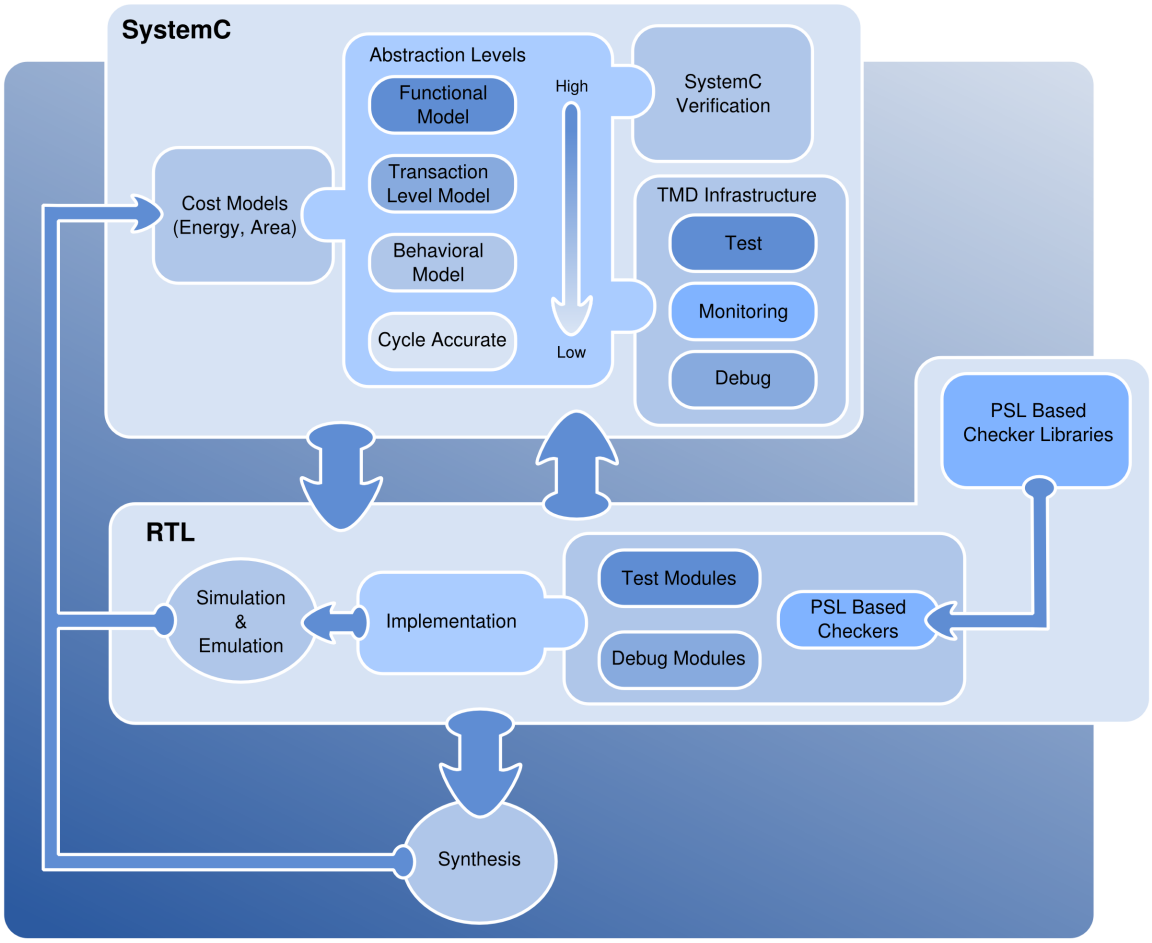


Figure 5.3: The quality of design (QoD) flow incorporates system debug and monitoring infrastructure through the use of debug and assertion modules, and reuses the NoC for test and verification.

toring points are needed in the NoC to observe the dynamic system performance. Those monitoring points can then be recorded as *valuable* for in-silicon implementation.

In addition, the reliability aspect of the NoC architecture can be considered in the evaluation of the final quality of the architecture. This recent work is being addressed in our group by M.H. Neishaburi [123] and his work on Reliability Aware NoC Router (RAVC).

5.5.3 Integration in System Design Flows

As future designs will start with high-level design that may incorporate, for example, SystemC synthesis as part of an *Electronic System Level* (ESL) design flow, one can expect to see PSL-type of expressions used at the higher levels of abstraction [124]. As such, it will be possible to incorporate PSL into the high-level SystemC models used for architectural exploration; that is, the "Monitoring" box in the SystemC portion of the design flow shown in Figure 5.3 would be replaced by PSL that could be fed into the checker generator tool directly, obviating the need to translate SystemC assertion and hand-coded temporal checkers to PSL.

When an error condition has been detected by the monitoring hardware, the system will be required to diagnose the problem and take appropriate action. Therefore, to complement the checker hardware, debug modules are added to the design as shown in Figure 5.3. The interaction between debug and checker hardware will be further discussed in later sections; specifically, we will show a concrete example of how we applied the quality design flow to the two ring-based interconnects. The debug hardware provides a mechanism for the diagnosis of, and possible recovery from detected runtime errors.

5.5.4 Design Space Exploration

Design space exploration is the process of comparing the properties (e.g area, speed, power) of several system configurations for the purpose of selecting the most appropriate candidate. Figure 5.3 shows how high-level cost models are integrated into the SystemC prototyping phase to help drive early design space ex-

ploration. The cost models, developed by using FPGA or ASIC synthesis results of a specific design instance are being fed back in the design process. For example, the synthesis results of a specific configuration of a hierarchical ring interconnect [15] can be used to develop a cost function that takes as input certain design parameters (e.g. bus-width, FIFO depths), and allows different configurations to be compared without the need to synthesize each instance. The use of cost models early in the design phase increases the probability that resource and performance constraints will be met by the final implementation, and thus is an important step in improving quality of design.

One of the contributions of our work was to automate the process from PSL to RTL, synthesis and extraction of the hardware cost. In doing so, PSL statements can directly be weighted against each other based on their hardware cost and usefulness in silicon.

While verification is an extremely important part of the usual design flow, it does not have an impact on the resource usage of the final implementation. However, the TMD infrastructure used in our QoD flow will eventually translate structures into hardware. It is therefore necessary that the overhead of the TMD infrastructure also be characterized and included in the cost models so that its resource requirements are accounted for during the design space exploration stages. Also, TMD cost models can be used to optimize the included TMD components to meet resource constraints. For example, it is probably too expensive to include all of the assertion checkers, or to instantiate all the debug modules at *every node* in the network.

5.5.5 Quantifying Quality

Improving quality by adding the TMD infrastructure to the design flow seems quite directly related to the extent of the hardware additions. However, how does one quantify the improvement? And, how can the quality of two different designs be compared when performing the architectural exploration?

A representative quantitative measure will enable the quality evaluation of several architectures such that the one with the best score can be selected. A quality

function that aims at measuring the quality of a specific design instance d_i is:

$$Q(d_i) = \lambda Q_V(d_i) + \rho Q_{TMD}(d_i) + \sigma Q_{NoC}(d_i) \,, \tag{5.1}$$

where Q_V is a numerical representation of *verification quality* of the components used in the design, Q_{TMD} is a measure of the quality of the TMD infrastructure hardware, and Q_{NoC} is a measure of the quality of the network architecture/topology. Q_{NoC} can also factor in the reliability aspect of the router when one has to trade-off this parameter during the design exploration phase.

The term Q_V can be a combination of the functional coverage of each component and the completeness of the system-level verification effort (i.e. the higher the coverage, the higher the quality score) and can come from the coverage database or an evaluation by qualified engineers.

To facilitate the comparison of architecture with differing numbers of nodes, the terms terms Q_V and Q_{TMD} can be expressed as an average value per node. For example, the value of Q_{TMD} can be computed by solving

$$Q_{TMD}(d_i) = \frac{\alpha Q_T(d_i) + \beta Q_M(d_i) + \gamma Q_D(d_i)}{|d_i|} \,, \tag{5.2}$$

where Q_T, Q_M, Q_D are the quality scores of the test, monitoring, and debug hardware, respectively. Dividing by $|d_i|$, the number of nodes in the architecture, yields an average Q_{TMD} value per node. The calculation of Q_{TMD} described by (5.2) assumes that the capabilities of the TMD infrastructure hardware can be numerically represented. One possible way to score assertions is to simply rank each one using a numeric value. The calculation of Q_{NoC} will be discussed further in Section 5.5.11.3. The constants $\{\alpha, \beta, \gamma, \lambda, \rho, \sigma\}$ in (5.1) and (5.2) assign relative weights in the calculation of Q.

5.5.6 The Cost of Quality

Under ideal circumstances, one would wish to add as much extra TMD hardware as possible to maximize quality. However, the added functionality comes at the cost of extra resource requirements, which may exceed the budget allowed for

the design. It will therefore be necessary to find a way to balance quality against the required overhead. Similar to the calculation of quality, we express the resource requirement of a design instance d_i as:

$$R(d_i) = R_0(d_i) + R_{TMD}(d_i) \,, \tag{5.3}$$

where R_0 is the resource requirement of the design without TMD infrastructure, and R_{TMD} is the resource requirement of the TMD infrastructure components, which can be expressed similarly to (5.2).

5.5.7 Optimizing Quality vs. Cost

In addition to integrating TMD infrastructure to increase quality, the design flow shown in Figure 5.3 is also used to support design space exploration and optimization such that performance and resource constraints can be met. To achieve this end, cost models are developed with the aid of FPGA and ASIC synthesis results, which are used to drive exploration at *higher levels of abstraction* using SystemC instead of RTL. The feedback loop in Figure 5.3 shows how synthesis results from the lower abstraction levels are used at the SystemC level to develop better cost models. Only a subset of the possible configuration of parametrizable components are synthesized to obtain a few data points, which can be used to infer relatively accurate estimates for the range of possible configurations. This is key to enabling rapid high-level exploration and optimization, because synthesizing every possible design instance variation and comparing resource requirements would require significant computation resources.

Under resource constraints, optimizing quality versus cost (resource requirements) is a multi-objective optimization problem where the goal is to maximize quality and minimize cost. The objective function used for optimization can be defined as the ratio $Q : R$, thus the optimal design d_i can be found by solving

$$\max \left[\frac{Q(d_i)}{R(d_i)} \right] \,, \tag{5.4}$$

where R and R_{TMD} can be constrained if desired. When comparing two designs,

the one with the greatest $Q : R$ ratio exhibits the best quality versus cost tradeoff.

The RTL box in Figure 5.3 shows how the checker/assertion hardware is selected from large libraries (a significant portion of those come from re-use of the verification assertions, cover and sequence statements converted to hardware). A practical design will have resource constraints, so only a subset of the TMD capabilities will go in hardware, and the problem becomes one of how to efficiently select the best subset such that the quality is maximized for the resource constraint R_{TMD}. The problem of packing the assertion checkers under resource constraints has been addressed through temporal multiplexing of assertion checkers as explained in Section 3.3. In practical designs, this means incorporating programmable logic fabric on the ASIC SoC device [125] [4].

5.5.8 FPGA Emulation in Quality-driven Architecture Exploration

It is accepted that FPGA emulation/prototyping can bring decisive advantages to the multiprocessor and computer architecture research. Using such platforms, a number of NoC topologies have been studied, where the transaction-level SystemC models were augmented with RTL modules to be run directly on FPGAs. The modeling can be made more accurate when needed, but also, the inherent complexity and difficulties in validating a given proposal will only be possible by accurate models executed at the speed of FPGA circuits (to reach an acceptable verification quality Q_V).

The NoC architectural studies performed can be augmented with the energy and area models, derived from more refined RTL block implementations. In short, quality considerations benefit well from FPGA emulation to advance architectural research and offer a flexible testing ground to refine the quality metrics.

4. Companies such as Menta S.A.S. offer such type of IP products in soft (RTL) or hard-IP format.

5.5.9 Networking and Quality of Service

The networking protocol and data structures require careful considerations to incorporate support for the TMD infrastructure. The networking header definitions have to include fields such that the routers can autonomously terminate transactions that are destined to the monitor and debug units, or the router itself. The NoC may need to support virtual channels that allow differentiated services of TMD traffic. One such service is the transport of assertion information, which is characterized by small messages (i.e. low bandwidth) requiring low latency and best effort delivery. To support debugging, the NoC has to have at least one communication channel that can bypass all the flow control and force the delivery of the payload without considering the congestion level in the FIFOs (e.g. overwrite user data to break out of a deadlock situation). These bypass channels (refer to Figure 5.2 for an example) need to support complex error conditions in the NoC and to be able to control the PE at a very low level.

5.5.10 Other Networking Considerations

Bypass mechanisms can assist in increasing the yield if the full NoC capabilities are not required (e.g. a reduced functionality, lower-cost device). Bypass logic and associate control points have a small overhead, but their benefit during debug (isolating a complete station) would weigh in very favorably for their presence in the silicon. Though bypass requires a time-to-live counter as part of protocols such that packets destined to a disabled endpoint do not endlessly loop in the device, the ability to bypass nodes as a way to improve yield is likely to be an important consideration for very large and dense designs which inherently have a greater probability of on-chip defect.

For an arbitrary NoC architecture, the maximal number of hops can be predetermined and the time-to-live counter size adjusted accordingly. Built-in performance monitors report expired packets to the software layers so that they are aware of the incorrect routing tables. Since the network includes the necessary interfaces to take over the control of the PE resources (the CPU, mostly), debugging and diagnostic transactions can be initiated while the NoC is still operating with

one (or more) of the PEs in bypass mode. In many applications, this can mean a reduced level of service to the end-user, but not a complete failure of the NoC.

5.5.11 Quality Comparison

The proposed quality-driven flow will be presented by comparing the quality versus cost of the hierarchical and hyper ring topologies. The aim is to show how this integrates the TMD infrastructure hardware into the NoC.

To compare the quality of both architectures, we consider the individual terms of (5.1).

5.5.11.1 Quality of Verification

As previously discussed, the value of Q_V from (5.1) is meant to quantify the thoroughness of the verification effort performed on each individual component as well as at the system level (i.e. Q_V can be expressed as a function of block and system level verification completeness). Since both architectures are constructed using the same components (i.e. RIs and IRIs), the block-level verification value contributed to Q_V is approximately the same. Furthermore, the similarity of the architectures allowed the reuse the same application code and synthetic test benches to perform system-level verification. Therefore, one can reason that

$$Q_V(d_{hierarchial-rings}) \approx Q_V(d_{hyper-rings}) \,. \tag{5.5}$$

For the purposes of this example, they are close enough to be considered equal.

5.5.11.2 Quality of TMD Infrastructure

The quality index of the TMD infrastructure is different for the two architectures because the monitoring and debugging capabilities is higher in the hyper-rings architecture. The calculation of Q_{TMD} is dependent on several factors such as location (or placement) and capabilities of the TMD infrastructure hardware. For example, the placement of debug and monitoring hardware at the inter-ring interfaces brings more quality to the design because they are the bridge between the global and local rings, hence more of the traffic will pass through those nodes.

The number of network input ports of each component can be used to quantify the Q_{TMD} values of the ring-interface (RI) and inter-ring interface components as:

$$\begin{aligned} Q_{TMD}(d_{RI}) &= 1\,, \\ Q_{TMD}(d_{IRI}) &= 2\,. \end{aligned} \tag{5.6}$$

The hierarchical ring architecture is composed of 16 RIs, and 4 IRIs (20 nodes), and the hyper-rings architecture has 8 IRIs (24 nodes). Using (5.6) and (5.2), the quality index for the TMD infrastructure for both architectures is:

$$\begin{aligned} Q_{TMD}(d_{hierarchial-rings}) &= \frac{16 \cdot Q_{TMD}(d_{RI}) + 4 \cdot Q_{TMD}(d_{IRI})}{20} &&= 1.20\,, \\ Q_{TMD}(d_{hyper-rings}) &= \frac{16 \cdot Q_{TMD}(d_{RI}) + 8 \cdot Q_{TMD}(d_{IRI})}{24} &&= 1.33\,. \end{aligned} \tag{5.7}$$

It should be noted that (5.7) assumes that the all the components in the architecture contain TMD infrastructure hardware, and as such, the comparison is straightforward. However, under resource constraints, one may wish to restrict the number of IRI and/or RI components that are TMD enabled, thereby resulting in an asymmetric distribution of TMD hardware. In this case, the comparison and selection between the two architectures becomes an optimization problem with a potentially large solution space.

5.5.11.3 Quality of NoC Architecture

The calculation of Q_{NoC} from (5.1) is an open problem as there are a large number of network characteristics that may be taken into account when comparing the quality of two NoC architectures. For example, the bisection bandwidth, an often studied property, may be included along with the node degrees, network diameter, path diversity, etc. For the purpose of this example, it is sufficient to evaluate the relative quality of each architecture by using a general understanding of their properties. The hyper-ring architecture has a higher bisection bandwidth and path diversity, which is important for debugging as one global ring can be reserved for debug traffic to provide quality-of-service. In this example, the property of path

diversity is of primary concern, so the quality of the NoC topology is based on the number of paths between any two nodes in network. For the two architectures under consideration, Q_{NoC} is defined as

$$\begin{aligned} Q_{NoC}(d_{hierarchial-rings}) &= 1\,, \\ Q_{NoC}(d_{hyper-rings}) &= 2\,, \end{aligned} \tag{5.8}$$

since the hyper-ring architecture has 2 possible paths between any two nodes.

Our representation of Q_{NoC} has been deliberately kept simple, but for a real application, we would factor in other architectural characteristics such as node degree, network diameter, etc. The difficulty in defining the quality index of a specific architecture lies in the large number of properties that can be considered. Also, the importance of each property can vary depending on the application domain, problem, resource and packaging constraints.

The NoC community is putting forth a lot of effort to agree on benchmarks that, ultimately, would allow various NoC architectures and implementations to be compared [89].

5.5.12 Hardware Resources and Quality

A distinction is made between hardware checkers derived from temporal logic and debug units which are *purposely designed*. Hardware checkers derived from temporal logic tend to focus on a very particular aspect of the design such as a protocol property that should remain valid at all times. One such example would be the following PSL statements that check for valid worm length and proper ending of worms in a NoC:

The `ValidWormLen` checker from Example 6, when translated to hardware, is quite small (comparable to a 6-bit counter). The `ExpectEnd` checker uses only one to two flip-flops (depending on the output buffering settings) and a few gates of logic. The Q_{TMD} value for such a checker could be weighed quite highly and since their hardware overhead is small, they would be good candidates to be preserved in silicon.

A faulty behavior in a module of the NoC will trigger the chain of events dis-

Example 6 NoC Worm Checker in PSL.

```
property ValidWormLen = always StartOfWorm |=>
                          { [*63] ; EndOfWorm };
property ExpectEnd = {StartOfWorm |=> eventually! EndOfWorm
                      abort AbortedWorm};
assert ValidWormLen;
assert ExpectEnd;
```

cussed above and will result in a meaningful error message being logged after a lookup in the assertion database is performed via the user-space monitoring daemon software. One or more assertion failures in an operation might indicate, for example, that a certain part of the circuit is faulty (timing errors, defective gates or electromigration failure, to name a few possible sources of problems). The NoC debug infrastructure can then make use of local scan test access to run more specific tests to assist in isolating the fault.

Debug units, hand-crafted to provide advanced capabilities that are beyond what can be created from statements in PSL, can also be considered. In a NoC, an example would be a worm backtrace buffer that can memorize the last n worms that were routed out of a module along with a time stamp of the interval between those worms. Such hardware monitors can be quickly designed in an ESL flow and imply memory (n entries deep and with a bit width capable of holding routing information about the worm and its time stamp). The Q_{TMD} of this debug monitor can be high, but it carries a significant area overhead. A good compromise would be to instantiate it in only a specific subset of stations (located at key, high-traffic points in the NoC) such that they would benefit the most in troubleshooting a problem. For the hierarchical and hyper ring topologies studied, this type of debug unit yields the greatest benefit when located at the inter-ring interfaces since they tend to process more NoC traffic.

5.5.13 Comparing Quality/Cost Ratios

The topology quality metric Q_{TMD} can be better illustrated with an example. A comparison between the hierarchical-ring and hyper-ring topologies was per-

Table 5.1: Area and power comparison of the TMD quality in the hierarchical-ring and hyper-ring topologies for two frequency of operations

Target Frequency	Total Cell Area (mm^2)	Total Power (W)	Q Quality	$Q : R_A$ Area Score	$Q : R_P$ Power Score
Hierarchical-ring					
500 MHz	5.10	2.26	2.20	0.43	0.97
250 MHz	4.95	1.11	2.20	0.44	1.98
Hyper-ring					
500 MHz	5.98	2.66	3.33	0.55	1.25
250 MHz	5.82	1.31	3.33	0.57	2.54

formed using the RTL version of the two NoC interconnects. They were synthesized using Synopsys DC Ultra (Version X-2005.09) targeting the TSMC 0.18 μm standard-cell [126] library and operating at 1.8 V. A selection of results for two different target frequencies are summarized in Table 5.1. Note that for this example, we assign equal weights to each term in (5.1) and (5.2) (i.e. the multipliers are equal to 1). As previously discussed, the weights can be adjusted depending on the design constraints and requirements.

From the Q_{TMD} values from (5.7) and the synthesis results obtained in Section 5.5.13, we can solve for the $Q : R$ ratio (5.4) of each topology. Those results are summarized in Table 5.1. The higher ratio obtained for the hyper-rings tells us that the quality increase due to the addition of the second global ring and the TMD infrastructure can be achieved at a relatively low cost. We can therefore observe that the hyper-rings obtain a superior $Q : R$ score than the hierarchical-rings.

Using only the Q_{TMD} scores, this topology would already be selected for its higher *quality*. In many designs where the redundancy in the center ring and higher bandwidth offsets the slightly higher resource usage, the Q_{NoC} for the hyper-ring is also going to be higher (the exact factor is application dependant). Therefore, those calculations would lead us to use the hyper-ring instead of the hierarchical ring for a better *quality*.

This simplified example is only meant to convey the general spirit of the proposed method. The application domain, cost constraints and weighting coefficients for the various elements driving the overall quality scores must be carefully

selected using heuristics or analysis of prototyping results. The optimization problem can then be solved using known algorithms, aiming for the highest *quality* while meeting the constraints.

5.6 Chapter Summary

This chapter has shown that by using appropriate hardware support, the memory mapped assertion and sequence checkers can be integrated in future distributed systems using the NoC paradigm such that they can benefit from the centralization of assertion and sequence status information.

Through the provision of specialized virtual channels and priority routing of information, assertion checker failures triggered across multiple locations in a NoC can be reported with some information on the source of the error and also a relatively accurate temporal relationship. Finally, it is proposed that a set of quality scores be used in the design exploration process as a mean of optimizing the cost benefit tradeoffs of integrating a large number of checkers in future NoC-based systems.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Since the complexity of hardware systems is not going to decrease in the immediate future, one has to mitigate the project risks by following a design flow that assists in the debugging at every level. By proposing a method that *re-uses* existing verification checker libraries, the proposed workflow aims to reduce the burden of integrating meaningful checkers in digital hardware systems. The proposed approach is *scalable* and *modular* because of its use of memory space and independent user-space drivers.

Systems integrating hardware assertion checkers and their associated firmware libraries will have a self-contained debug infrastructure that can be leveraged even as the devices are deployed in the field. Checkers that are in *close proximity* to the source of a problem will assist the debugging process, making it more efficient and more focused. A system-level crash on a multi-core NoC would be a nightmare to analyze with only a snapshot of the device's state. However, with a multitude of very pinpointed and low-level checkers, the signature of a complex bug can be analyzed with software tools and the order of event, location and nature of the hardware modules involved in the crash can be extracted. The debugging process thus has a solid foundation to proceed through the reproduction of the failure in the verification environment.

Already, the verification process dominates large design schedules. Only through a comprehensive validation plan that includes coverage analysis, and ideally formal verification of critical components, can companies hope to deliver nearly flawless devices that make so many useful products possible. Hardware-based verification acceleration seems like the only feasible approach when tackling system-level integration of the larger devices. Workstations simply cannot cope with the burden of simulating ever larger silicon designs. Hardware-based coverage at or near speed of operation will thus be required to guarantee a proper level of confidence required to tape-out future silicon designs.

It is clear that the proposed method involves a certain level of hardware overhead. Projects need to weigh and score the benefits of each hardware checker that will end up in the silicon against their overhead and cost. The proposed TMD methodology can assist in selecting the appropriate checkers and physically locate them in large NoC or complex SoC so as to maximize their benefits. The hardware overhead of checker integration could be significant, but so are the benefits when one has to debug such systems. In the end, compromises will have to be made as with any other design process. As technology evolves, the cost related to adding debugging support hardware will go down while the complexity and cost of debugging complex problems will go up. This only strengthens the need for a comprehensive debug methodology that work across the design flow and in all layers of the system, from the low-level bus interfaces to the operating system integration.

Finally, the proposed method of integration of hardware sequence checkers as user-space, software-accessible modules has many other potential uses. The massively parallel architecture offered by FPGA-based sequence checkers has applications well beyond debugging or verification coverage. A few examples were proposed, but many other applications can be explored.

The combination of the automated translation of temporal language to digital hardware coupled with the generation of the corresponding software interface can bring performance benefits to many applications that require timely and efficient analysis of patterns in very large data sets.

The technical nature of the proposed debug infrastructure, while complex, can

be tackled with algorithms, design automation and logic gates. Yet, there remains one problem that is not so technical in nature: there is a really strong need for teaching advanced verification methods and debug methodology to future digital engineers. The ABV techniques and use of temporal logic as a way to express hardware specifications has been proven to be beneficial in many large design projects. The largest IC design houses use it extensively in their design flows. Yet, very few courses in electrical engineering (even at the graduate level) target these topics. Perhaps because of the maturity required to understand the usefulness of formally specifying designs, or the already complex nature of hardware assignments given out to students and their lack of experience with hardware design languages, verification is very often relegated as a secondary problem.

As noted by teachers [127, 128, 129, 130], if the verification methodology is taught up front with as much emphasis as the tools and techniques to make the circuit, students with a solid background in verification and debugging would have the insight that will help them efficiently perform their hardware design process and deliver quality results.

6.2 Future Work

In addition to addressing the need for a more efficient and flexible debugging process, other applications can be derived from the combination of hardware checkers and memory mapping automation. The following ideas can provide realistic and interesting research paths that would reuse the mechanisms and extend the reach of the concepts to other computer engineering problems.

6.2.1 Software Debugging and Data Integrity Checking

Besides its usefulness in pinpointing hardware problems and guiding the silicon debug process, hardware sequence checkers can assist advanced software debugging. One has to view a part of the presented solution in the reverse direction; in other words, from userspace software to hardware checkers.

Dynamic query-based debugging has been proposed as a way to rapidly identify a problem [131] in software systems. In typical implementations, the system will dynamically monitor a set of properties, for example the relationship between variables. The software monitors those elements to ensure that they meet their properties. However, this adds *considerable overhead* since the system has to dedicate a lot of CPU time to monitoring properties. Furthermore, those properties are only validated *periodically* as the CPU also needs time slices to perform useful calculations. In multi-core systems, another CPU can be reserved for checking. Once again, if there are many variables involved requiring many concurrent checks, the dedicated core will run out of processing power to handle all those operations. In presenting their approach [131], the authors noted: "The dynamic query debugger does not allow to query temporal properties of the objects. [...] Such functionality would involve using temporal logic and program execution tracing and is beyond the scope of this system."

Our presented hardware assisted approach to temporal checking could address the problem of temporal execution tracing and monitor temporal properties of objects. The hardware processes performing property checking in real time are inherently parallel. To support this software debugging approach, one would need a set of tools similar to the proposed hardware sequence and assertion checkers,

namely a system with access to programmable fabric (a FPGA compute card with access to a memory-mapped system bus, such as PCI or PCI-Express), the checker generator, the hardware register layer and a user-space driver enabled to expose the hardware-mapped memory address range to the user program.

Software objects (variables) would have to be forced to be located into the UIO memory space map. This can be done by allocating the data structure that needs to be monitored such that its base address coincides with the memory map pointer provided by the mmap() call.

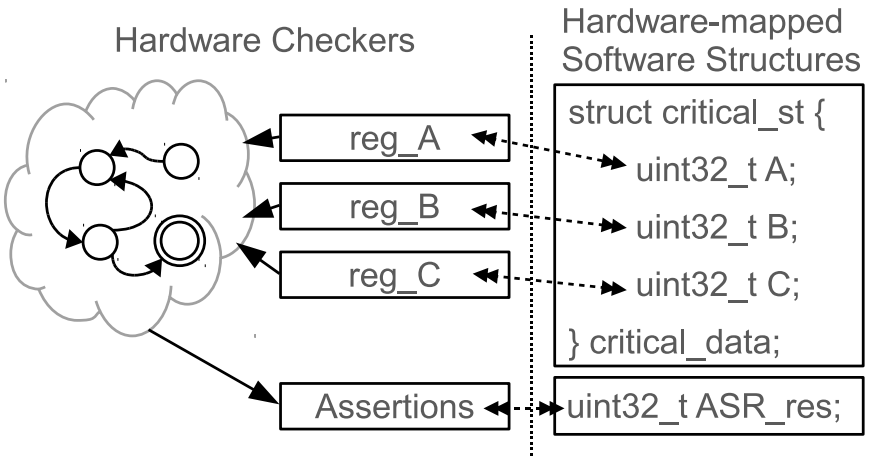


Figure 6.1: Hardware-based temporal checkers for software-based structures.

The hardware registers would have to be generated based on a selection of software data structures that need to be temporally checked. Those variables would then be assigned to corresponding hardware registers by means of the memory map. In this approach, the hardware doesn't need to modify the variables; it simply provides register-based storage and a secondary access for the hardware checkers to constantly monitor those variables. Figure 6.1 illustrates this proposed mechanism. Since the software variables would directly correspond to a hardware register, they can also be simultaneously read by multiple hardware "threads".

Temporal expressions could then be written using the variable names (they would have a corresponding hardware signal name) like in the current process when dealing with hardware signals. The checker generator can create the hardware automata required to simultaneously check all the temporal relationships be-

tween the variables. Using the same process for monitoring the assertion checkers and coverage counters, software debugging based on temporal properties of objects would be possible. Instead of a hardware "clock", each read or write access strobe to the variable block would be used to trigger a change in the hardware automata.

The limitations of this method would be:

- The hardware resources are finite and one cannot create temporal checkers that end up requiring too much corresponding hardware.
- Dynamic data structures would not be directly supported.
- The frequency of operation (access) to those variables would impose a small performance penalty. The software variables accessed often are cached by the CPU. Depending on the logic depth required by the checkers and external bus performance, the access would be slower than a cache hit. In most systems, the access time for those variables would be similar to SDRAM or SRAM access on an external peripheral, so overall, the speed penalty would be acceptable.
- The method would be limited to monitoring "simple" data types, for example, characters, integers or Booleans. It would not be possible to monitor indirection (pointers) except for address bound checking.

The method could be very valuable as a way to ensure that critical pieces of software always respect a set of properties, for example assuming a set of variables A,B,C:

- A is always written before B
- At all time, A+B must be greater than C
- If A < B it implies that within some amount of time C < 0 will occur.

This proposed mechanism would be able to detect any deviance from the rules within a few clock cycles, which from a software point of view, is quasi-instantaneous.

In addition to the software front-end to map the objects to the hardware memory-mapped structure, the error reporting would be similar to the case of hardware assertion checking. An assertion failure in the hardware or the software would be reported by a bit indicating the failure, which can trigger an exception mechanism (e.g. interrupt) to signal the problem to a high-level layer.

Open problems remain in the definition of the *sampling* event when dealing

with the software variables. Languages such as PSL are defined with a *clock* acting as the sampling event. Translating this sampling mechanism to software accesses to a variable can be addressed in multiple ways, with clocked time intervals or *hits* in memory as possible sampling events. One could consider hardware events as part of the Boolean layer of the language and some syntax *sugaring* added to support implicit events such as reading or writing a variable.

Once properly refined, this new hardware-assisted debugging process could lead to applications of particular interest in the software domain, for example critical pieces code that have to always obey certain fundamental relationships under all conditions. One could conceivably accompany a software device driver with its corresponding hardware checker unit that would be used to constantly monitor the software data structures. As more advanced process technology merges the CPU with in-system re-programmable logic, this could allow novel debugging methods to be developed.

6.2.2 High-throughput Pattern Matching

In many applications, there is a need to rapidly check for patterns in data streams. Telecommunication systems often call network elements capable of inspecting packets passing in their structures and keeping track of high-level protocol states "smart switches". A common example is a TCP/IP connection that go through a few states to be established. "Smart switches" are able to monitor connection attempts and can observe that, for example, a remote address attempts many connections in a small period of time. This could mean that an attacker is trying to breach the system. For low-bandwidth applications a CPU can inspect the header of packets passing through a switch. However, in core routers (those handling the traffic of entire companies or cities), the problem becomes quite difficult to handle due to the very high rate of packets passing through the router. Using temporal expressions along with a variation on the **Assertion Threading** method presented in Section 3.2.5, one can utilize the expressive power of a temporal language to filter through sequences of packets.

The difference from **Assertion Threading** is that in this method, the dispatcher is not trying to inspect *pipelined events*, but rather *context-based events*. The dis-

patcher would use the context information of the packet (in IP-based hardware switches, this is typically a hash computed from the source and destination addresses and ports) to launch *tokens* in banks of hardware-based checker automata. If the initiation rate on each bank is low enough, each assertion checking automaton state can be *saved* in RAM memory and the hardware checkers themselves can be time-multiplexed to allow many contexts to be checked for patterns, while keeping the flip-flop and LUT overhead acceptable. This process is known as *pipeline sharing* and many examples can be found in the literature.

In such an application, the sampling event is the packet itself since the protocols changes state by exchanging bits in the packet headers. The problem with timeouts and similar autonomous behavior from the sender and receivers can be explored further should one wish to refine this proposed idea.

This work is presented with the hope that someone with a totally different background and application domain will one day find a novel use for this technology to benefit a new and different area of research.

6.2.3 Assertion Clustering and Trigger Units

The proposed memory mapping and register packing algorithm presented in Chapter 4 treats each checker as an independent unit and processes them as such. Recent work by M.H. Neishaburi [132] was shown to be able to extract common fan-in among checkers. Through modifications of the packing algorithm, a preselection of checkers could be performed such that they share as many primary inputs as possible (or allocate a score to those who do).

In doing this check before the packing process, a set of checkers could be partitioned into different memory map decoders such as to minimize the required physical fan-in for a given checker group.

Even more recent work by M.H. Neishaburi et al. [133] shows the application of the sequence checkers for hierarchical trigger generation and other uses in post-silicon debugging. Their work highlights the broad range of uses for sequence checkers. At the time of this writing, the publications are not yet indexed, but interested readers can refer to the bibliography for the article titles [134, 135, 136].

Appendix A

Examples from the BEE2

This Appendix provides more details and source code to give a better understanding of the BEE2 support files needed to enable UIO integration. The kernel boot log of the BEE2 is also shown to better understand how the system starts up and provides insight on the required device drivers and order of operations.

A.1 UIO Range Remapping Kernel Module

The following listing shows the small kernel device driver that can be loaded to enable User-Space IO re-mapping of a programmable hardware base address.

Some C programmers tend to shy away from using the ***goto*** statements. In Linux kernel drivers, however, the goto statement is used to unwind the loading process should some error occur during the various memory and device allocation processes that are part of setting up the driver. Using ***goto*** statements help ensure that the de-allocation process follows a mirror of the allocation and that the code does not end up too *nested* up in ***if / else*** statements.

Listing A.1: Userspace I/O Range Remapping Kernel Driver

```
/*
 * UIO driver for mapping an address range to userspace
 * Accepts base address, size and reservation parameters at module loading
 *
 * (C) 2008 Jean-Samuel Chenard
 *
 * Licensed under GPLv2 only.
 *
 */

#include <linux/module.h>
#include <linux/device.h>
#include <linux/platform_device.h>
#include <linux/uio_driver.h>
#include <linux/ioport.h>
#include <linux/moduleparam.h>

#include <asm/io.h>

#define UIO_NAME        "UIO Range Map - CMC Demonstration"
#define UIO_VERSION     "0.1.1"

static unsigned int uio_baseaddr    = 0x00000000;
static unsigned int uio_addrrng     = 16;
static int      uio_reserve_mem         = 1;

// Accept module parameters at load time
module_param( uio_baseaddr, uint, S_IRUGO );
module_param( uio_addrrng,  uint, S_IRUGO );
module_param( uio_reserve_mem,  bool, S_IRUGO );

static struct uio_info *simhw_uio = NULL;

static int setup_uio_mem( struct uio_info *uio, const char *name,
    unsigned char index, unsigned long addr, unsigned long size) {
        struct resource *p_res = NULL;

    uio->mem[index].name = name;
        uio->mem[index].addr = addr;
        uio->mem[index].size = size;
        uio->mem[index].memtype = UIO_MEM_PHYS; // We want physical memory mapping (hardware)

        printk(KERN_INFO "UIO Setup Memory baseaddr=0x%lX, size=0x%lX\n", addr, size );
```

```
        if( uio_reserve_mem ) {
                p_res = request_mem_region(uio->mem[index].addr,
                                           uio->mem[index].size, name);
                if( p_res == NULL ) {
                        printk(KERN_ALERT "I/O Region Reservation failed\n");
                        return -ENODEV;
                }
        }


        uio->mem[index].internal_addr = ioremap(uio->mem[index].addr,
                                                uio->mem[index].size);

        if ( uio->mem[index].internal_addr == NULL ) {
                printk(KERN_ALERT "I/O Remap failed for %s\n", name);
                goto out_release_region;
        }

        return 0;

out_release_region:
        if( uio_reserve_mem )
                release_mem_region( uio->mem[index].addr, uio->mem[index].size );
        return -ENODEV;
}

static void release_uio_mem(struct uio_info *uio, unsigned char index ) {
        iounmap( (volatile void*) simhw_uio->mem[index].internal_addr );
        if( uio_reserve_mem ) {
                release_mem_region( simhw_uio->mem[index].addr, simhw_uio->mem[index].size );
    }
}

static int __init simhw_init(void) {
        printk(KERN_INFO "UIO SimHW Loading\n");

        simhw_uio = kzalloc(sizeof(struct uio_info), GFP_KERNEL);
        if (!simhw_uio)
                return -ENOMEM;

        simhw_uio->name = UIO_NAME;
        simhw_uio->version = UIO_VERSION;
        simhw_uio->irq = UIO_IRQ_NONE;

    // When calling setup_uio_mem, one can give meaningful names to the mapping(s)
        if( setup_uio_mem( simhw_uio, "some_registers", 0,
                           uio_baseaddr, uio_addrrng ) != 0 )
                goto out_free;

        if ( uio_register_device(&platform_bus, simhw_uio) )
                goto out_release_regs;

        return 0;  // All is good !

out_release_regs:
        release_uio_mem( simhw_uio, 0 );
out_free:
        kfree (simhw_uio);
        return -ENODEV;
};


static void __exit simhw_rel(void)
{
    printk(KERN_INFO "Removing UIO\n");
        uio_unregister_device(simhw_uio);
        release_uio_mem( simhw_uio, 0 );
```

```
        kfree (simhw_uio);
}

module_init(simhw_init)
module_exit(simhw_rel)

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Jean-Samuel Chenard");
```

A.2 UIO Register Access in Python

The following snippet of Python code shows how easy the user access to the registers become once the kernel user module is loaded.

Listing A.2 Userspace I/O access in Python

```
# Python-based access to UIO memory map example

import mmap
import binascii

# Points to the UIO device (virtual file)
uio_name = '/dev/uio0'

f = open(uio_name,'r+b')
# MMAP a chunk of data (512 bytes) from the uio0 virtual file
map = mmap.mmap(f.fileno(),512)

# Read a 32 bit register between byte offsets 0x000 and 0x004 from the
# hardware memory map
test_reg = map[0x000:0x004]

print "Read from the mem map : ", binascii.hexlify(test_reg)
```

A.3 BEE2 Boot Log

Below is the boot log capture of the BEE2 showing the custom bootloader, network kernel loading and networked file system. The command line shows the customized kernel version and the shutdown sequence. Some settings were incorrect in the root file system, such as the swap file setting (unnecessary in the application). However, those small configuration issues did not prevent the sytem from working well.

The boot log also shows how fast the system starts up. On powerup, there is about a 2 second delay for the control FPGA to load up and initiate the U-Boot from the CompactFlash card. U-Boot has a 3 second delay to allow the user to stop the boot process and edit commands or perform register manipulations. Then, the kernel is transfered via tftp from the PC workstation. This process takes approximately 3 seconds. De-compressing the kernel then takes approximately 1 second. From that point, the kernel displays the system time, from which one can see that the initialization of the peripherals takes approximately 4 seconds. Once initialized, most of the remaining time spent to finish is taken by the network intialization and mounting of the network root file system (approximately 5 seconds).

```
U-Boot 1.3.1-gb7e58b32 (Jan 25 2008 - 22:54:27)


### HW ID not found in environment ###
*********************************************
* Welcome to the BEE2 - U-Boot session       *
* Brought to you by the                      *
* McGill Integrated Microsystems Laboratory  *
*                                            *
* Ported to BEE2 by Jean-Samuel Chenard      *
* Release 1.0a                               *
*********************************************

DRAM:  128 MB
Using default environment

In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
TFTP from server 192.168.0.110; our IP address is 192.168.0.105
Filename 'uImage'.
Load address: 0x400000
Loading: T #################################################################
 ################
done
Bytes transferred = 1187568 (121ef0 hex)
## Booting image at 00400000 ...
   Image Name:   Linux-2.6.24-rc5-xlnx-jsc-xlnx-n
```

```
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    1187504 Bytes =  1.1 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
[    0.000000] Linux version 2.6.24-rc5-xlnx-jsc-xlnx-nfs-g669cb9c0 (jsamch@amirix) (gcc
  version 4.0.0 (DENX ELDK 4.1 4.0.0)) #9 Thu Jan 24 12:49:33 EST 2008
[    0.000000] Xilinx Generic PowerPC board support package (Xilinx ML300) (Virtex-II Pro)
[    0.000000] Zone PFN ranges:
[    0.000000]   DMA             0 ->    32768
[    0.000000]   Normal      32768 ->    32768
[    0.000000] Movable zone start PFN for each node
[    0.000000] early_node_map[1] active PFN ranges
[    0.000000]     0:        0 ->    32768
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 32512
[    0.000000] Kernel command line: console=ttyUL0
   ip=192.168.0.105:192.168.0.110:192.168.0.1:255.255.255.0:bee2:eth0:off mem=512M
   nfsroot=/home/jsamch/busyfs root=/dev/nfs rw
[    0.000000] Xilinx INTC #0 at 0x40614000 mapped to 0xFDFFF000
[    0.000000] PID hash table entries: 512 (order: 9, 2048 bytes)
[    0.000197] Console: colour dummy device 80x25
[    0.001067] Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
[    0.002325] Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
[    0.033519] Memory: 127360k available (1848k kernel code, 616k data, 104k init, 0k highmem)
[    0.124533] Mount-cache hash table entries: 512
[    0.129590] net_namespace: 64 bytes
[    0.134211] NET: Registered protocol family 16
[    0.137778] Registering device uartlite:0
[    0.138716] Registering device xsysace:0
[    0.139667] Registering device xilinx_emac:0
[    0.173786] NET: Registered protocol family 2
[    0.208774] IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
[    0.211466] TCP established hash table entries: 4096 (order: 3, 32768 bytes)
[    0.211910] TCP bind hash table entries: 4096 (order: 2, 16384 bytes)
[    0.212253] TCP: Hash tables configured (established 4096 bind 4096)
[    0.212293] TCP reno registered
[    0.225551] sysctl table check failed: /kernel/l2cr .1.31 Missing strategy
[    0.225633] Call Trace:
[    0.225653] [c7c17e80] [c00084f4] show_stack+0x4c/0x174 (unreliable)
[    0.225748] [c7c17eb0] [c00305d8] set_fail+0x50/0x68
[    0.225824] [c7c17ed0] [c0030c60] sysctl_check_table+0x670/0x6bc
[    0.225878] [c7c17f10] [c0030c74] sysctl_check_table+0x684/0x6bc
[    0.225929] [c7c17f50] [c001df0c] register_sysctl_table+0x5c/0xac
[    0.225997] [c7c17f70] [c0258b78] register_ppc_htab_sysctl+0x18/0x2c
[    0.226052] [c7c17f80] [c0252848] kernel_init+0xc8/0x284
[    0.226121] [c7c17ff0] [c0004af8] kernel_thread+0x44/0x60
[    0.233110] Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
[    0.234397] io scheduler noop registered
[    0.234440] io scheduler anticipatory registered (default)
[    0.234465] io scheduler deadline registered
[    0.234647] io scheduler cfq registered
[    0.303732] uartlite.0: ttyUL0 at MMIO 0x40600003 (irq = 1) is a uartlite
[    0.303804] console [ttyUL0] enabled
[    0.562715] RAMDISK driver initialized: 4 RAM disks of 10000K size 1024 blocksize
[    0.572798] System ACE at 0x40618000 mapped to 0xC9002000, irq=0, 125184KB
[    0.578796]  xsa: xsa1 xsa2
[    0.595698] tun: Universal TUN/TAP device driver, 1.6
[    0.599554] tun: (C) 1999-2004 Max Krasnyansky <maxk@qualcomm.com>
[    0.607517] xilinx_emac xilinx_emac.0: MAC address is now  2: 0: 0: 0: 0: 0
[    0.613276] XEmac: using sgDMA mode.
[    0.616825] XEmac: not using TxDRE mode
[    0.620572] XEmac: not using RxDRE mode
[    0.629295] XEmac: Detected PHY at address 0, ManufID 0x0013, Rev. 0x78e2.
[    0.635045] eth0: Dropping NETIF_F_SG since no checksum feature.
[    0.643891] eth0: Xilinx 10/100 EMAC at 0xFFFD0000 mapped to 0xC9008000, irq=2
[    0.649925] eth0: XEmac id 1.1a, block id 32, type 1
```

```
[    0.655819] mice: PS/2 mouse device common for all mice
[    0.660274] TCP cubic registered
[    0.663217] NET: Registered protocol family 1
[    0.667695] NET: Registered protocol family 17
[    0.672918] RPC: Registered udp transport module.
[    0.676601] RPC: Registered tcp transport module.
[    4.184141] IP-Config: Complete:
[    4.185925]       device=eth0, addr=192.168.0.105, mask=255.255.255.0, gw=192.168.0.1,
[    4.193699]      host=bee2, domain=, nis-domain=(none),
[    4.198921]      bootserver=192.168.0.110, rootserver=192.168.0.110, rootpath=
[    4.207522] Looking up port of RPC 100003/2 on 192.168.0.110
[    4.229133] Looking up port of RPC 100005/1 on 192.168.0.110
[    9.290755] VFS: Mounted root (nfs filesystem).
[    9.294447] Freeing unused kernel memory: 104k init
init started: BusyBox v1.8.1 (2008-01-30 14:51:17 EST)
starting pid 139, tty '': '/etc/init.d/rcS'
=====================================
Welcome to JSC's BEE2 Root filesystem
=====================================

Mounting kernel file systems...done.
Starting System loggers...done.
Bringing up the lo and eth0 network interface (IP=192.168.0.105)...done.
Starting the dropbear ssh server...done.
Starting the telnet server...done.
Setup completed.  System is ready.

Please press Enter to activate this console.
starting pid 160, tty '': '/bin/sh'
# uname -a
Linux bee2 2.6.24-rc5-xlnx-jsc-xlnx-nfs-g669cb9c0 #9 Thu Jan 24 12:49:33 EST 2008 ppc unknown
# poweroff
starting pid 165, tty '': 'umount'
# umount: cannot remount /dev/root read-only
umount: cannot umount /: Device or resource busy
umount: cannot remount rootfs read-only
umount: cannot umount /: Device or resource busy
starting pid 167, tty '': 'swapoff'
swapoff: /etc/fstab: No such file or directory
The system is going down NOW!
Sending SIGTERM to all processes
Sending SIGKILL tRequesting system poweroff
[   34.129069] Power down.
[   34.130250] System Halted
```

A.4 BEE2 Control FPGA Device Utilisation

The following log shows the size of the control FPGA of the BEE2 when the system is built to support the ported Linux 2.6 kernel, Ethernet, DDR DRAM, UART console and in-system reprogramming port (SelectMAP).

```
Design Summary:
Number of errors:      0
Number of warnings:   63
Logic Utilization:
  Number of Slice Flip Flops:       7,543 out of  66,176   11%
  Number of 4 input LUTs:           7,966 out of  66,176   12%
Logic Distribution:
  Number of occupied Slices:        7,861 out of  33,088   23%
Total Number of 4 input LUTs:         10,613 out of  66,176   16%
  Number used as logic:             7,966
  Number used as a route-thru:        606
  Number used for Dual Port RAMs:   1,580
    (Two LUTs used per Dual Port RAM)
  Number used as Shift registers:     461

  Number of bonded IOBs:              247 out of     996   24%
    IOB Flip Flops:                   276
    IOB Master Pads:                    1
    IOB Slave Pads:                     1
    IOB Dual-Data Rate Flops:         109
  Number of PPC405s:                    2 out of       2  100%
  Number of JTAGPPCs:                   1 out of       1  100%
  Number of Block RAMs:                93 out of     328   28%
  Number of GCLKs:                      8 out of      16   50%
  Number of DCMs:                       3 out of       8   37%
  Number of GTs:                        0 out of      20    0%
  Number of GT10s:                      0 out of       0    0%

   Number of RPM macros:           36
Total equivalent gate count for design:  6,472,048
Additional JTAG gate count for IOBs:  11,856
Peak Memory Usage:  617 MB
Total REAL time to MAP completion:  14 mins 48 secs
Total CPU time to MAP completion:   12 mins 34 secs
```

A.5 UIO and Remap-Range Memory Utilisation

Capture of the kernel uio.ko and remap-range.ko modules loaded on the BEE2. Note that the machine states **ML300** but the actual hardware running the commands is the BEE2. The reason is that the ML300 is the ancestor reference platform to the BEE2 design and some of the Linux driver code still refers to the ML300 in some areas of the kernel (and there is little incentive to modify it).

```
# cat /proc/cpuinfo
processor       : 0
cpu             : Virtex-II Pro
clock           : 300MHz
revision        : 8.160 (pvr 2001 08a0)
bogomips        : 297.98
machine         : Xilinx ML300
plb bus clock   : 100MHz
# ./lsuio
uio0: name=UIO-Remapper, version=0.0.1, events=0
        map[0]: addr=0x40600000, size=16
# lsmod
Module                  Size  Used by    Not tainted
remap_range             4328  0
selectmap               9580  0
uio                    13720  1 remap_range
# ls -l uio.ko remap-range.ko
-rw-r--r--    1 root     root       113808 Mar 22  2008 remap-range.ko
-rw-r--r--    1 root     root       146895 Mar 22  2008 uio.ko
# uname -a
Linux bee2 2.6.24-rc5-xlnx-jsc-xlnx-nfs-g669cb9c0
      #9 Thu Jan 24 12:49:33 EST 2008 ppc unknown
```

Bibliography

[1] P. Dorsey, "Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency," tech. rep., Xilinx, Oct. 2010. Accessed Jan 31, 2011.

[2] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: a high-end reconfigurable computing system," *IEEE Design & Test of Computers*, vol. 22, pp. 114–125, Mar./Apr. 2005.

[3] B. Goertzel, "Human-level artificial general intelligence and the possibility of a technological singularity: A reaction to Ray Kurzweil's The Singularity Is Near, and McDermott's critique of Kurzweil," *Artificial Intelligence*, vol. 171, no. 18, pp. 1161 – 1173, 2007. Special Review Issue.

[4] R. Gronheid, G. Vandenberghe, K. Ronse, E. Hendrickx, V. Wiaux, A.-M. Goethals, P. Jansen, M. Maenhoudt, and R. Jonckheere, "Lithography options for the 32 nm half pitch node and beyond," *IEEE Transactions on Circuits and Systems*, vol. 56, pp. 1884–1891, Aug 2009.

[5] L. Scheffer, L. Lavagno, and G. Martin, *Electronic Design Automation for Integrated Circuits Handbook*. Taylor & Francis, 2006.

[6] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, April 1965.

[7] H. Foster, A. Krolnik, and D. Lacey, *Assertion-based design*. Kluwer Academic Pub, second ed., 2004.

[8] D. Josephson and B. Gottlieb, "The crazy mixed up world of silicon debug," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 665 – 670, oct 2004.

[9] G. Van Rootselaar and B. Vermeulen, "Silicon debug: scan chains alone are not enough," in *International Test Conference Proceedings*, pp. 892–902, Sept. 1999.

[10] D. Agans, *Debugging: the 9 indispensable rules for finding even the most elusive software and hardware problems*. Amacom Books, 2002.

[11] M. Boule, J.-S. Chenard, and Z. Zilic, "Adding debug enhancements to assertion checkers for hardware emulation and silicon debug," in *International Conference on Computer Design*, pp. 294–299, Oct 2006.

[12] M. Boule, J.-S. Chenard, and Z. Zilic, "Debug enhancements in assertion-checker generation," *IET Computers & Digital Techniques*, vol. 1, pp. 669–677, nov 2007.

[13] M. Boule, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *International Symposium on Quality Electronic Design*, pp. 613–620, mar 2007.

[14] J.-S. Chenard and Z. Zilic, "Efficient memory mapping of hardware assertion and sequence checkers for on-line monitoring and debug," *Under submission*, 2011.

[15] S. Bourduas, J.-S. Chenard, and Z. Zilic, "A RTL-level analysis of a hierarchical ring interconnect for Network-on-Chip multi-processors," in *Proceedings of the International SoC Design Conference*, October 2006.

[16] J.-S. Chenard, S. Bourduas, N. Azuelos, M. Boule, and Z. Zilic, "Hardware assertion checkers in on-line detection of faults in a hierarchical-ring network-on-chip," in *Workshop on Diagnostic Services in Network-on-Chips*, pp. 371–375, DATE, Apr. 2007.

[17] S. Bourduas, J.-S. Chenard, and Z. Zilic, "A quality-driven design approach for NoCs," *IEEE Design & Test of Computers*, vol. 25, pp. 416–428, Sept-Oct 2008.

[18] J.-S. Chenard, "Application note: Configuring, building and running linux 2.6 on the bee2 with the busybox user environment," tech. rep., CMC Microsystems, June 2009.

[19] J.-S. Chenard, "Application note: Extending the flexibility of bee2 by using u-boot to load the linux kernel via ethernet," tech. rep., CMC Microsystems, Mar. 2010.

[20] J.-S. Chenard, "Application note: Using linux userspace i/o for rapid hardware driver development," tech. rep., CMC Microsystems, Feb. 2011.

[21] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 3rd ed., Nov. 2005.

[22] K. Yaghmour, *Building Embedded Linux Systems*. O'Reilly, 1st ed., Apr. 2003.

[23] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*. O'Reilly, 3rd ed., Feb. 2005.

[24] C. Hallinan, *Embedded Linux Primer: A Practical Real-World Approach*. Prentice Hall PTR, 1st ed., Sept. 2006.

[25] M. W. Chiang, Z. Zilic, K. Radecka, and J.-S. Chenard, "Architectures of increased availability wireless sensor network nodes," in *Proceedings of the 2004 International Test Conference*, pp. 1232 – 1241, oct. 2004.

[26] J.-S. Chenard, Z. Zilic, C. Y. Chu, and M. Popovic, "Design methodology for wireless nodes with printed antennas," in *Proceedings of the 42nd Design Automation Conference*, pp. 291 – 296, june 2005.

[27] J.-S. Chenard, Z. Zilic, and M. Prokic, "A laboratory setup and teaching methodology for wireless and mobile embedded systems," *IEEE Transactions on Education*, vol. 51, pp. 378 –384, aug. 2008.

[28] B. Vermeulen, N. Stollon, R. Kuhnis, G. Swoboda, and J. Rearick, "Overview of debug standardization activities," *IEEE Design & Test of Computers*, vol. 25, pp. 258–267, May-June 2008.

[29] NVidia Corporation, *NVIDIA CUDA C Programming Guide*, 2010 (accessed March 1, 2011). `http://www.nvidia.com/object/cuda_home_new.html`.

[30] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E, "ATTILA: a cycle-level execution-driven simulator for modern GPU architectures," in *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pp. 231 – 241, 2006.

[31] S.-T. Shen, S.-Y. Lee, and C.-H. Chen, "Full system simulation with QEMU: An approach to multi-view 3D GPU design," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pp. 3877 – 3880, June 2010.

[32] Q. Hou, K. Zhou, and B. Guo, "Debugging GPU stream programs through automatic dataflow recording and visualization," *ACM Trans. Graph.*, vol. 28, pp. 153:1–153:11, December 2009.

[33] A. Ho, S. Hand, and T. Harris, "Pdb: pervasive debugging with xen," in *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pp. 260 – 265, nov. 2004.

[34] E. Borghei, R. Azmi, and A. Ghahremanian, "Improving driver reliability through inlined reference monitor based on virtualization," in *IEEE International Conference on Intelligent Computing and Intelligent Systems*, vol. 3, pp. 324 – 328, oct. 2010.

[35] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006.

[36] W. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proceedings of the Design Automation Conference*, 2001.

[37] L. Benini and G. D. Micheli, "Networks on chips: a new SoC paradigm," *IEEE Computer*, vol. 35, Jan. 2002.

[38] I. S. 1850-2005, *IEEE Standard for Property Specification Language (PSL)*. New York, NY, USA: Institute of Electrial and Electronic Engineers, Inc., 2005.

[39] R. Beers, "Pre-rtl formal verification: An intel experience," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 806–811, 2008.

[40] M. Vardi, "Formal techniques for systemc verification; position paper," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pp. 188–192, 2007.

[41] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-chip Designs: For System-on-a-chip Designs*. Springer, 2002.

[42] ARM Ltd., *ARM AMBA Documentation*, 2011 (accessed March 31, 2011). `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html`.

[43] Opencores, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, 2011 (accessed March 1, 2011). `http://cdn.opencores.org/downloads/wbspec_b4.pdf`.

[44] *SystemC Version 2.0 User's Guide — Update for SystemC 2.0.1*, 2010. Available at `http://www.systemc.org`.

[45] M. Smirnov and A. Takach, "A SystemC superset for high-level synthesis," in *Forum on Specification Design Languages*, pp. 1 –6, Sept. 2009.

[46] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 8 –17, 2009.

[47] P. Coussy and A. Morawiec, *GAUT: A High-Level Synthesis Tool for DSP Applications From C Algorithm to RTL Architecture*. Springer, 2008.

[48] *Functional specification for SystemC 2.0*, 2010. Available at `http://www.systemc.org`.

[49] C. Montemayor, M. Sullivan, J.-T. Yen, P. Wilson, and R. Evers, "Multiprocessor design verification for the PowerPC 620 microprocessor," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 188–195, Oct 1995.

[50] Y.-M. Kuo, C.-H. Lin, C.-Y. Wang, S.-C. Chang, and P.-H. Ho, "Intelligent random vector generator based on probability analysis of circuit structure," in *International Symposium on Quality Electronic Design*, pp. 344–349, March 2007.

[51] K. Shimizu and D. Dill, "Deriving a simulation input generator and a coverage metric from a formal specification," in *Proceedings of the Design Automation Conference*, pp. 801–806, 2002.

[52] J. Tong, M. Boule, and Z. Zilic, "Airwolf-TG: A test generator for assertion-based dynamic verification," in *IEEE International High Level Design Validation and Test Workshop*, pp. 106 – 113, nov. 2009.

[53] C. Y. Lin, S. Cao, J. An, F. Han, and Q. Fan, "A network based functional verification method of IEEE 1394a PHY core," in *IEEE Computer Society Annual Symposium on VLSI*, pp. 245–250, April 2008.

[54] F. Vitullo, S. Saponara, E. Petri, M. Casula, L. Fanucci, G. Maruccia, R. Locatelli, and M. Coppola, "A reusable coverage-driven verification environment for network-on-chip communication in embedded system platforms," in *Workshop on Intelligent solutions in Embedded Systems*, pp. 71–77, June 2009.

[55] A. Piziali, *Functional verification coverage measurement and analysis*. Springer, 2004.

[56] A. Pnueli, "The temporal logic of programs," in *Annual Symposium on Foundations of Computer Science*, pp. 46–57, Nov 1977.

[57] B. Turumella and M. Sharma, "Assertion-based verification of a 32 thread SPARC CMT microprocessor," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 256–261, 2008.

[58] S. Hessabi, A. Gharehbaghi, B. Yaran, and M. Goudarzi, "Integrating assertion-based verification into system-level synthesis methodology," in *Proceedings of the 16th International Conference on Microelectronics*, pp. 232–235, 2004.

[59] A. Gharehbaghi, M. Babagoli, and S. Hessabi, "Assertion-based debug infrastructure for SoC designs," in *Internatonal Conference on Microelectronics*, pp. 137 –140, 2007.

[60] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proceedings of the Design Automation Conference*, (New York, NY, USA), pp. 7–12, ACM, 2006.

[61] I. H. R. Laboratory, "FoCs formal verification," Feb. 2010. `https://www.research.ibm.com/haifa/projects/verification/focs/`.

[62] M. Boule and Z. Zilic, "Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties," in *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, pp. 69–76, 2006.

[63] M. Boule and Z. Zilic, "Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation," in *Proceedings of the 12th Asia and South Pacific Design Automation Conference*, pp. 324–329, IEEE Computer Society, 2007.

[64] W. de Boer and B. Vermeulen, "Silicon debug: avoid needles respins," in *Electronics Manufacturing Technology Symposium, 2004. IEEE/CPMT/SEMI 29th International*, pp. 277–281, July 2004.

[65] K. Goossens, B. Vermeulen, and A. Nejad, "A high-level debug environment for communication-centric debug," in *Design, Automation and Test in Europe*, pp. 202–207, April 2009.

[66] J. Geuzebroek and B. Vermeulen, "Integration of Hardware Assertions in Systems-on-Chip," in *IEEE International Test Conference*, ITC Proceedings, pp. 412–421, 2008.

[67] B. Vermeulen, "Design-for-debug to address next-generation SoC debug concerns," in *Test Conference, ITC 2007. IEEE International*, p. 1, Oct. 2007.

[68] E. Larsson, B. Vermeulen, and K. Goossens, "A distributed architecture to check global properties for post-silicon debug," in *Test Symposium (ETS), 2010 15th IEEE European*, pp. 182 –187, May 2010.

[69] S. Goel and B. Vermeulen, "Data invalidation analysis for scan-based debug on multiple-clock system chips," in *European Test Workshop, 2002. Proceedings. The Seventh IEEE*, pp. 61–66, Nov. 2002.

[70] C. Pyron, R. Bangalore, D. Belete, J. Goertz, A. Razdan, and D. Younger, "Silicon symptoms to solutions: applying design for debug techniques," in *Test Conference, 2002. Proceedings. International*, pp. 664 – 672, 2002.

[71] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Trans. Computers*, vol. 36, no. 4, pp. 471–482, 1987.

[72] B. Peischl and F. Wotawa, "Error traces in model-based debugging of hardware description languages," in *Proceedings of the International Symposium on Automated Analysis-driven Debugging*, pp. 43–48, 2005.

[73] Y.-C. Hsu, B. Tabbara, Y.-A. Chen, and F. Tsai, "Advanced techniques for rtl debugging," in *Proceedings of the Design Automation Conference*, (New York, NY, USA), pp. 362–367, ACM, 2003.

[74] Y.-C. Hsu, F. Tsai, W. Jong, and Y.-T. Chang, "Visibility enhancement for silicon debug," in *Proceedings of the Design Automation Conference*, (New York, NY, USA), pp. 13–18, ACM, 2006.

[75] H. Yi, S. Park, and S. Kundu, "A design-for-debug (dfd) for noc-based soc debugging via noc," in *Asian Test Symposium, 2008. ATS '08. 17th*, pp. 289–294, Nov. 2008.

[76] A. Gharehbaghi, M. Babagoli, and S. Hessabi, "Assertion-based debug infrastructure for soc designs," in *Microelectronics, 2007. ICM 2007. Internatonal Conference on*, pp. 137 –140, Dec. 2007.

[77] B. Quinton and S. Wilton, "Programmable logic core enhancements for high-speed on-chip interfaces," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, pp. 1334 –1339, sept. 2009.

[78] N. Nicolici and H. F. Ko, "Design-for-debug for post-silicon validation: Can high-level descriptions help?," in *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*, pp. 172–175, Nov. 2009.

[79] H. F. Ko and N. Nicolici, "Combining scan and trace buffers for enhancing real-time observability in post-silicon debugging," in *Test Symposium (ETS), 2010 15th IEEE European*, pp. 62 – 67, may 2010.

[80] M. Gao and K.-T. Cheng, "A case study of time-multiplexed assertion checking for post-silicon debugging," in *High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International*, pp. 90 –96, June 2010.

[81] V. Raghunathan, M. B. Srivastava, and R. K. Gupta, "A survey of techniques for energy efficient on-chip communication," in *Design Automation Conference, 2003. Proceedings*, pp. 900–905, June 2–6, 2003.

[82] C. A. Zeferino, M. E. Kreutz, L. Carro, and A. A. Susin, "A study on communication issues for systems-on-chip," in *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*, pp. 121–126, 2002.

[83] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys*, vol. 38, no. 1, 2006.

[84] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Proc. of DATE*, pp. 250–256, ACM Press, 2000.

[85] A. M. Amory, K. Goosens, E. J. Marinissen, M. Lubaszewski, and F. Moraes, "Wrapper Design for the Reuse of a Bus, Network-on-chip, or Other Functional Interconnect as Test Access Mechanism," *IET Computers and Digital Techniques*, vol. 1, no. 3, pp. 197–206, 2007.

[86] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock rate versus IPC: the end of the road for conventional microarchitectures," in *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 248–259, 2000.

[87] C. Nicopoulos, V. Narayanan, and C. R. Das, *Network-on-Chip Architectures: A Holistic Design Exploration (Lecture Notes in Electrical Engineering)*. Springer, Oct. 2009.

[88] E. Salminen, A. Kulmala, and T. D. Hämäläinen, "Survey of network-on-chip proposals," tech. rep., OCP-IP, Mar. 2008.

[89] C. Grecu, A. Ivanov, R. Pande, A. Jantsch, E. Salminen, U. Ogras, and R. Marculescu, "Towards open network-on-chip benchmarks," in *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pp. 205–212, May 2007.

[90] N. Stollon, B. Uvacek, and G. Laurenti, "Standard debug interface socket requirements for OCP-Compliant SoC," tech. rep., OCP-IP Debug Working Group, 2007.

[91] S. Bourduas, *Modeling, evaluation, and implementation of ring-based interconnects for network-on-chip*. PhD thesis, McGill University, 2008.

[92] B. Team, "Building BEE2: a case for high-end reconfigurable computer (HERC)," tech. rep., UC Berkeley, Jan. 2004. Accessed Oct 5, 2007.

[93] H. K.-H. So and R. W. Brodersen, "Improving usability of FPGA-Based reconfigurable computers through operating system support," *Proceedings of 16th International Conference on Field Programmable Logic and Applications*, pp. 1–6, 2006.

[94] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, pp. 1138 – 1149, june 2008.

[95] M. Boulé and Z. Zilic, "Incorporating Efficient Assertion Checkers into Hardware Emulation," in *Proceedings of the 23rd IEEE International Conference on Computer Design (ICCD'05)*, pp. 221–228, 2005.

[96] M. Boule and Z. Zilic, *Generating Hardware Assertion Checkers for Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer Science, 2008.

[97] Yu-Chin Hsu, B. Tabbara, Y. Chen, and F. Tsai, "Advanced techniques for RTL debugging," in *Proceedings of the 40th Design Automation Conference (40th DAC)*, pp. 362–367, 2003.

[98] Y.-C. Hsu, F. Tsai, W. Jong, and Y.-T. Chang, "Visibility Enhancement for Silicon Debug," in *Proceedings of the 43rd Design Automation Conference (43rd DAC)*, pp. 13–18, 2006.

[99] Accellera Organization, Inc., "Property Specification Language – Reference Manual, v.1.1." `www.eda.org/vfv/docs/PSL-v1.1.pdf`, 2004.

[100] M. Siegel, A. Maggiore, and C. Pichler, "Untwist your brain - efficient debugging and diagnosis of complex assertions," in *46th ACM/IEEE Design Automation Conference*, pp. 644 – 647, july 2009.

[101] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient Detection of Vacuity in Temporal Model Checking," *Formal Methods in System Design*, vol. 18, no. 2, pp. 141–163, 2001.

[102] S. Ruah, D. Fisman, and S. Ben-David, "Automata Construction for On-The-Fly Model Checking PSL Safety Simple Subset," Tech. Rep. H-0234, IBM, 2005.

[103] M. Boulé and Z. Zilic, "Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation," in *Proceedings of the 12th Asia and South Pacific Design Automation Conference (ASP-DAC2007)*, pp. 324–329, 2007.

[104] M. Boulé and Z. Zilic, "Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties," in *Proceedings of the 2006 IEEE International High Level Design Validation and Test Workshop (HLDVT'06)*, pp. 69–76, 2006.

[105] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, California: The Morgan Kaufmann Series in Computer Architecture and Design, third ed., 2003.

[106] M. Platzner, "Reconfigurable Computer Architectures." `http://citeseer.ist.psu.edu/490784.html`.

[107] N. Abel, S. Manz, F. Grull, and U. Kebschull, "Increasing design changeability using dynamic partial reconfiguration," *IEEE Transactions on Nuclear Science*, vol. 57, pp. 602–609, Apr. 2010.

[108] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. McGraw-Hill Book Company, 1999.

[109] B. Cohen, S. Venkataramanan, and A. Kumari, *Using PSL/ Sugar for Formal and Dynamic Verification*. Los Angeles, California: VhdlCohen Publishing, 2004.

[110] Y. Oddos, K. Morin-Allory, and D. Borrione, "Assertion-based verification and on-line testing in horus," in *3rd International Design and Test Workshop*, pp. 249 –254, Dec. 2008.

[111] G. Stringham, *Hardware/Firmware Interface Design: Best Practices for Improving Embedded Systems Development*. Newnes, 2009.

[112] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 3, no. 4, pp. 299–325, 1974.

[113] ARM Ltd., *ARM CMSIS V2.00*, 2011 (accessed March 31, 2011). `http://www.onarm.com/cmsis/download/`.

[114] B. W. Boehm, *Software Engineering Economics*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 1981.

[115] Y. H. Choi, W. I. Kwon, and H. N. Kim, "Code generation for linux device driver," *International Conference on Advanced Communication Technology*, vol. 1, pp. 734–737, 2006.

[116] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das, "Exploring Fault-Tolerant Network-on-Chip Architectures," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pp. 93–104, 2006.

[117] D. Spoor and R. Chu, "Building causal models for operator aiding in a supervisory control environment," in *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics*, pp. 344–347, 1993.

[118] M. Riley, N. Clestrom, M. Genden, and S. Sawamura, "Debug of the cell processor: Moving the lab into silicon," in *IEEE International Test Conference*, pp. 1–9, October 2006.

[119] C. Ciordas, K. Goossens, A. Rădulescu, and T. Basten, "Noc monitoring: Impact on the design flow," in *IEEE International Symposium on Circuits and Systems*, pp. 1981–1984, 21-24 May 2006.

[120] C. Ciordas, T. Basten, A. Rădulescu, and K. Goossens, "An event-based monitoring service for networks on chip," in *ACM Transactions on Design Automation of Electronic Systems*, pp. 702–723, October 2005.

[121] É. F. Cota, M. E. Kreutz, C. A. Zeferino, L. Carro, M. Lubaszewski, and A. A. Susin, "The impact of NoC reuse on the testing of core-based systems," in *VLSI Test Symposium*, pp. 128–133, 2003.

[122] M. Kakoee, M. Neishaburi, M. Daneshtalab, S. Safari, and Z. Navabi, "On-chip verification of nocs using assertion processors," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pp. 535 – 538, aug. 2007.

[123] M. H. Neishaburi and Z. Zilic, "Eravc: Enhanced reliability aware noc router," in *International Symposium on Quality Electronic Design (ISQED)*, pp. 591 – 596, Mar. 2011.

[124] A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamidem, and Y. Lahbib, "Combining system level modeling with assertion based verification," in *Quality of Electronic Design, 2005. ISQED 2005. Sixth International Symposium on*, pp. 310 – 315, march 2005.

[125] S. Wilton and R. Saleh, "Programmable logic ip cores in soc design: opportunities and challenges," in *IEEE Conference on Custom Integrated Circuits*, pp. 63 – 66, 2001.

[126] Artisan Components Inc., Sunnyvale, California, USA, *TSMC 0.18µm Process 1.8-Volt SAGE-X Standard Cell Library Databook*.

[127] M. Velev, "Integrating formal verification into an advanced computer architecture course," *IEEE Transactions on Education*, vol. 48, p. 216, May 2005.

[128] M. Radu and S. Sexton, "Integrating extensive functional verification into digital design education," *IEEE Transactions on Education*, vol. 51, pp. 385 – 393, Aug. 2008.

[129] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander, "Debugging from the student perspective," *IEEE Transactions on Education*, vol. 53, p. 390, Sept. 2009.

[130] P. Nagvajara and B. Taskin, "Design-for-debug: A vital aspect in education," in *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, pp. 65 –66, June 2007.

[131] R. Lencevicius, *Advanced Debugging Methods*. Kluwer Academic Publishers, 2000.

[132] M. Neishaburi and Z. Zilic, "Enabling efficient post-silicon debug by clustering of hardware-assertions," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 985 – 988, march 2010.

[133] M. Neishaburi and Z. Zilic, "Hierarchical trigger generation for post-silicon debugging," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pp. 1 –4, april 2011.

[134] M. Neishaburi and Z. Zilic, "A fault tolerant hierarchical network on chip router," in *Proceedings of IEEE International Symposium on Defect and Fault Tolerance*, pp. 120–128, Oct. 2011.

[135] M. Neishaburi and Z. Zilic, "Debug-aware axi-based network interface," in *Proceedings of IEEE International Symposium on Defect and Fault Tolerance*, pp. 399–407, Oct. 2011.

[136] M. Neishaburi and Z. Zilic, "Hierarchical embedded logic analyzer for accurate root-cause analysis," in *Proceedings of IEEE International Symposium on Defect and Fault Tolerance*, pp. 445–453, Oct. 2011.

Glossary

ABV
assertion-based verification. 50, 67, 70, 167

ALU
arithmetic and logical unit. 30

AMBA
advanced microcontroller bus architecture. 42

API
application programming interface. 124, 129

ASIC
application specific integrated circuit. 3, 5, 26–28, 40–42, 68, 138, 155, 156

BFM
bus functional model. 44, 45, 48, 98

CPU
central processing unit. 6, 12, 27, 29–31, 40, 57, 63, 97–100, 103, 109, 111, 124, 125, 137, 140, 142, 145, 157, 171

DfD
design for debug. 23, 52–54, 59, 70

DUV
device under verification. 43–50, 69

EDA
electronic design automation. 1, 3, 7, 59

ESL
electronic system level. 152, 161

FPGA
field programmable gate array. 3, 26–28, 35, 40–42, 50, 64, 70, 74, 80, 84, 89, 101, 124, 125, 155, 156

GPU
graphic processing unit. 29, 30, 35

HDL
high-level description language. 37–39, 52, 53

IC
integrated circuit. 5, 6, 8, 23, 35, 37, 40, 44, 46, 56, 99, 102, 107, 125, 139, 167

IP
intellectual property. 3, 5, 7, 26, 42, 43, 45, 48, 70, 131

IRI
inter-ring interface. 141

LUT
look-up table. 42, 89

MMU
memory management unit. 132

NoC
network-on-chip. 6, 9, 14, 17, 23, 33, 34, 42, 48, 49, 52, 53, 58–61, 63, 64, 69, 101, 118, 123, 124, 126, 137–139, 141, 142, 144–150, 152, 156–163, 165, 166

OS
operating system. 31, 100, 124–127

PE
proccessing element. 140

PSL
property specification language. 35, 38, 48, 69, 72, 98, 104, 105, 107, 142, 152, 161, 171

QoD
quality of design. 138

RAM
random access memory. 100

RI

ring interface. 140, 145

RTL

register transfer level. 17, 39, 41, 43, 44, 50–53, 61, 63, 67, 111, 155, 156, 162

SoC

System-on-a-Chip. 6, 7, 23, 42, 53, 54, 100, 101, 111, 117, 124, 125, 150, 156, 166

SVA

system verilog assertions. 38

TMD

test, monitoring and debug. 150, 153–159, 162, 166

UIO

User space I/O. 125, 127–129, 132, 133, 135