MONITORAÇÃO DINÂMICA DE ASSERÇÕES PARA DEPURAÇÃO EM SILÍCIO

CELINA GOMES DO VAL

MONITORAÇÃO DINÂMICA DE ASSERÇÕES PARA DEPURAÇÃO EM SILÍCIO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof Claudionor José Nunes Coelho Jr

Belo Horizonte

Julho de 2011

CELINA GOMES DO VAL

DYNAMIC MONITORING OF ASSERTIONS FOR POST-SILICON DEBUG

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof Claudionor José Nunes Coelho Jr

Belo Horizonte

July 2011

© 2011, Celina Gomes do Val.
Todos os direitos reservados.

Val, Celina Gomes do.
V135m Monitoração Dinâmica de Asserções para Depuração em Silício / Celina Gomes do Val. — Belo Horizonte, 2011
xxvii, 76 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais — Departamento de Ciência da Computação

Orientador: Prof Claudionor José Nunes Coelho Jr.

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Monitoração dinâmica de asserções para depuração em silício

CELINA GOMES DO VAL

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. CLAUDIONOR JOSÉ NUNES COELHO JÚNIOR - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ANTÔNIO OTÁVIO FERNANDES
Departamento de Ciência da Computação - UFMG

PROFA. ANDREA IABRUDI TAVARES
Departamento de Computação - UFOP

PROF. DIÓGENES CECÍLIO DA SILVA JÚNIOR
Departamento de Engenharia Elétrica - UFMG

Belo Horizonte, 08 de agosto de 2011.

Acknowledgments

First, I would like to thank my advisor and former boss, Professor Claudionor Coelho Jr., for his support throughout my graduation as well as his technical guidance.

Additionally, I would like to thank all my colleagues from the laboratory (LECOM) and those from Jasper. During the period I spent in these places, I had many experiences that contributed to my professional growth. I would especially like to thank Fabiano Peixoto and Andrea Iabrudi for their great teaching, for their good advice and for keeping me focused.

I would like to acknowledge support from FAPEMIG and also the Xilinx® and Jaspergold® university programs, all of which were crucial to this project development.

I cannot overstate how much I appreciate the support I received from my family and friends. Only they know how difficult this journey was for me and, despite my lack of free time, they were always trying to amuse me. In particular, I would like to thank Damian for his amazing proofread and also my parents for looking after me. Without my parents help I probably would have starved during this time.

Finally, I would like to thank Ian, for bringing me joy when I needed the most. Moreover, for keeping me company during my long studying nights and not letting the distance drive us apart.

I wouldn't have achieved this important goal if it wasn't for all of you.

"I have no special talents. I am only passionately curious."
(Albert Einstein)

Resumo

A crescente demanda de mercado combinada ao aumento da complexidade e desempenho dos circuitos integrados atuais resultam em uma grande pressão sobre a depuração de silício, que é a última fase de desenvolvimento de um circuito. Ao contrário das técnicas de validação pré-silício, a depuração em silício possui duas grandes limitações: a falta de controlabilidade e a baixa visibilidade dos sinais internos. Com isso, esta etapa requere um esforço muito maior para a identificação da causa de um erro. A depuração em silício tem se tornado um gargalo na produção e custo de novos circuitos integrados e, por isso, tem sido considerada um tópico de interesse emergente para pesquisadores, bem como para a indústria de eletrônica.

O presente trabalho propõe uma nova técnica para melhorar a visibilidade interna dos circuitos durante a depuração em silício, na qual o grupo de sinais a ser observado é escolhidos dinamicamente. Para isso, desenvolveu-se uma arquitetura embutida de depuração para controlar a seleção desse grupo, assim como um software para integrar essa arquitetura com o circuito alvo.

A arquitetura embutida é composta por um módulo de depuração, que controla uma cadeia de asserções para identificação de falhas, bem como uma memória interna para captura dos sinais. De acordo com a falha identificada, um grupo específico de sinais deve ser capturado.

A determinação dos sinais que compõem cada grupo deve ser feita antes da produção do circuito. Para isso, desenvolveu-se um software capaz de selecionar sinais relevantes a uma asserção, sendo a escolha baseada em seu cone de influência (*cone of influence* ou COI). Este software também é capaz de agrupar as asserções de acordo com a semelhança entre seus COIs. Com isso, o programa gera um novo código para o circuito alvo, incluindo o módulo de depuração bem como as conexões necessárias para o seu funcionamento.

A fim de se testar a arquitetura proposta, foi realizado um estudo de caso com um processador baseado na arquitetura MIPS32 com 40 asserções embutidas. O estudo de caso permitiu avaliar a eficiência do sistema proposto, bem como reproduzir o fluxo

completo de depuração. Um segundo teste englobando outros dois projetos foi realizado com o intuito de analisar o algoritmo de seleção de sinais baseado em cones de influência.

A adição da arquitetura de depuração desenvolvida não impactou o desempenho do circuito, além disso, a área necessária para sua inclusão foi de 43 registradores e 307 LUTs (*Lookup Tables*). Isso atesta a viabilidade do uso do sistema embutido para depuração, já que não houve um impacto significativo. Além disso, o módulo de depuração foi capaz de capturar os dados necessários para a reprodução do erro por uma ferramenta de verificação formal.

Entretanto, o método proposto para seleção de sinais baseado no cone de influência das asserções se mostrou ineficiente, não havendo grande diferença entre os conjuntos de sinais que compõe o cone de influência das propriedades nos CIs analisados. Isso se deve ao fato dos circuitos atuais, em sua grande maioria, possuírem uma complexidade muito grande, sendo que seus componentes se encontram muito interligados entre si.

Visto a dificuldade encontrada, um segundo método para seleção de sinais foi proposto e implementado. Este se baseia não somente no conjunto de sinais que compõem o cone de influência, mas também na hierarquia funcional do circuito estabelecida durante a fase de desenvolvimento.

Por sua vez, essa segunda alternativa se mostrou muito mais promissora e capaz de empregar um foco maior em certas áreas do design que, provavelmente, são mais relevantes para a análise de cada propriedade. Apesar disso, uma melhor investigação deve ser feita a fim de se refinar o resultado dessa técnica, fazendo com que a mesma seja menos dependente do conhecimento do engenheiro quanto à estrutura do circuito, bem como seu processo de desenvolvimento e teste.

Abstract

The increasing demand for shorter time-to-market, combined with increased complexity and performance requirements put a tremendous pressure on post-silicon debug, which is usually the last step prior to chip release. In contrast to pre-silicon techniques, post-silicon debug have two main limitations, controllability and observability, which cause the failure analysis and identification to require significantly more effort. Post-silicon debug is thus becoming a potential bottleneck in productivity and cost. Therefore it is emerging as one of the most important topics for research and the EDA industry.

In this thesis, we present a novel technique to improve observability of circuits during post-silicon debug, where the choice of signals to be monitored is done accordingly to a failure detected. This technique requires an on-chip module to control signal selection, as well as an off-chip system to correctly integrate the debug architecture with the target circuit.

The on-chip architecture employs a scan-chain to identify any assertion failure and a trace buffer to store the state of a limited set of internal signals. A design for debug module was then developed to integrate the identification of the assertion that has failed, acquired by the scan-chain, to a system that chooses which signals must be captured by the trace buffer in the next execution.

One important aspect of the system is the correlation between the signals chosen to be captured and the assertion that has failed. Connection software was developed to do this selection in an early stage of development, before tape-out. This software implements the signal selection based on the cone of influence (COI) of the embedded assertions and it is also capable of gathering the assertions in clusters. Besides that, the developed software generates new register transfer level (RTL) code for the circuit under debug, including all the design for debug logic and the connections necessary to its proper functioning.

The proposed architecture was tested using a MIPS32 based processor with 40 assertions. This case study is presented to illustrate the efficiency of our architecture. The entire debug flow is reproduced, from design integration, then going through the

bug detection and finally the extraction of relevant data to reproduce the error using a design verification tool. Furthermore, a separate study was done to analyze the method proposed for signal selection and it included the analysis of two more designs.

The on-chip architecture had no impact over chip performance and the area overhead was minimum, thus it could be employed to post-silicon debug without any major impact. Additionally, the system was capable of extracting relevant data for the creation of lighthouses which were used to produce failing scenarios using a formal verification tool.

On the other hand the signal selection based only on the cone of influence of the assertions was not sensitive enough to decrease the area of the chip to be analyzed. In more complex systems, the connection between the internal components of the designs analyzed were too high, resulting in cones of influence similar to the entire circuit and also between all properties. A second approach was then suggested considering the hierarchical position of signals in the cone of influence of each property. Although the signals of the COI sets are almost the same, the distance and the path to an assertion checker define the hierarchical configuration of its COI.

The tests performed demonstrated that the second approach is a promising technique for selecting which areas of the chip should be observed according to an assertion failure. However, that approach still relies on the validation engineer knowledge of the system in addition to the development process, so it can be properly configured. In order to automate this process, more research should be done to refine the selection of which hierarchical instances should be considered for the signal selection.

Resumo Estendido

O processo de desenvolvimento de um circuito integrado possui várias etapas que vão desde sua especificação até seu lançamento ao mercado. Esse processo é geralmente dividido em duas fases, a pré-silício e a pós-silício, sendo que cada uma também é subdividida em menores etapas.

A fase pré-silício envolve no desenvolvimento e testes de modelos abstratos do circuito. O primeiro modelo é geralmente descritivo, feito em linguagem natural e consiste na especificação das funções a serem executadas, bem como na descrição do comportamento esperado.

Uma vez especificado, inicia-se a fase de implementação de um novo modelo, a qual é feita utilizando uma ou mais linguagens de descrição de *hardware*. A abstração desse novo modelo é feito em mais baixo nível, principalmente no nível de registradores (*register transfer level* ou RTL). É a partir da descrição em RTL que a maioria dos testes de validação funcional, seja via verificação formal ou simulação, são realizados a fim de se verificar o comportamento correto do circuito desenvolvido.

Uma terceira etapa precede a fabricação de circuitos em silício, nessa ferramentas de síntese são utilizadas para converter a descrição RTL em modelos de topografia do circuito. Esses novos modelos passam por uma série de testes para avaliar se o seu comportamento é o mesmo dos modelos esquemáticos, das etapas de desenvolvimento anteriores.

No entanto, as séries de testes aplicadas antes que se fabrique os circuitos, nem todos os erros são detectados e podem causar grande prejuízo caso sejam detectados somente após o lançamento do circuito no mercado. Logo, o primeiro lote de circuitos produzidos não é lançado no mercado. Ao invés disso, os primeiros *chips* são utilizados para novos testes de validação e possível depuração.

Os testes em silício possuem a vantagem de poderem ser aplicados ao circuito em si, sem perda de detalhes devido a qualquer abstração, além de poderem cobrir uma maior quantidade de estados, já que a velocidade de execução do circuito integrado é muito superior à execução de testes em modelos abstratos. Com isso, erros causados

por falhas elétricas ou erros de fabricação, bem como aqueles de difícil controle, são mais facilmente detectados durante esta etapa.

Apesar dessas vantagens, a depuração do silício é muito mais trabalhosa do que em qualquer outra etapa pré-silício. Isso se deve à falta de acesso das estruturas internas do circuito, bem como a menor controlabilidade dos erros. Ou seja, além de ser mais difícil de reproduzir um erro consecutivamente, não é possível determinar os eventos internos ao circuito que gerou o erro.

Devido a importância da fase de depuração em silício, do inglês *post-silicon debug*, será apresentado nessa dissertação um sistema desenvolvido para melhorar a depuração de erros funcionais detectados no silício. Esses erros possuem a característica de serem mais facilmente reproduzidos e possuem como maior obstáculo a falta de visibilidade interna do sistema em depuração.

O sistema de depuração proposto é composto de dois componentes principais, um software e um hardware que pode ser embutido no circuito a ser depurado.

O software desenvolvido possui como principal função a escolha adequada de sinais internos a serem observados durante as sessões de depuração. Supondo que a depuração se baseará no uso de asserções embutidas, o software é capaz de selecionar diferentes grupos de sinais a serem observados de acordo com a asserção analisada. Essa seleção de sinais se baseia no conceito de cone de influência (*cone of influence* ou COI) de cada propriedade, bem como na localização dos sinais em relação à mesma e no potencial de expansão de valores de cada sinal.

Além disso, o software desenvolvido também é capaz de fazer a inclusão automática do módulo de depuração desenvolvido no circuito a ser analisado. Uma última função do software é ser capaz de analisar os dados obtidos do circuito após uma sessão de depuração.

Já o módulo de depuração se baseia na aplicação de duas técnicas diferentes usadas para o aumento de visibilidade. Registradores em cadeia são utilizados para conectar as asserções embutidas no código ao módulo de depuração. Por sua vez, este possui um processador de asserções que é capaz de extrair através dessa cadeia, qual foi a falha detectada.

O módulo possuí um *trace-buffer*, que é uma memória embutida no sistema usada para capturar valores assumidos por alguns sinais durante sua execução. O mesmo é controlado por gerenciador de memória que foi adicionado ao módulo de depuração, o que permite que a memória seja utilizada somente em sessões de depuração, bem como permite a configuração de intervalos de captura.

Enfim, o módulo de depuração também possui uma unidade de controle para integrar as duas informações, ou seja, é capaz de escolher qual grupo de sinais deve ser

monitorado, dada a falha identificada pelo processador de asserções.

A fim de se checar o funcionamento correto do sistema proposto, bem como avaliar a sua eficiência, dois testes diferentes foram realizados. O primeiro teste, teste de sanidade, consistiu na escolha de um projeto de pequeno porte, um processador baseado na arquitetura MIPS32, para testar todo o fluxo de depuração, assim garantir que o sistema se comporta como o proposto.

O segundo teste foi realizado para avaliar qualitativamente a melhor estratégia de escolha de sinais de acordo com asserções a serem embutidas. Para este segundo teste, mais três projetos foram escolhidos e analisados.

A partir da primeira etapa de testes foi possível concluir que o sistema proposto é viável. A adição da arquitetura de depuração desenvolvida não impactou o desempenho do circuito, enquanto que a área necessária para sua inclusão foi razoável, apesar de que ainda requere melhorias. Isso atesta a viabilidade do uso do sistema embutido para depuração, já que não houve um impacto muito grande, o aumento na quantidade de registradores foi de 2.2%, já de *lookup tables* (LUTs) foi de 8.2%. Além disso, o módulo de depuração foi capaz de capturar os dados necessários para a reprodução do erro por uma ferramenta de verificação formal.

Quanto à análise de escolha de sinais, a primeira estratégia abordada de escolher os sinais a partir do cone de influência da propriedade se apresentou falho, dado que todas as propriedades do projeto tendem a apresentar COI semelhantes. Logo, uma segunda estratégia foi sugerida, a qual se baseia na configuração do COI da propriedade e na divisão de módulos do projeto.

Esta segunda estratégia se mostrou mais promissora, capaz de reduzir a quantidade de sinais de interesse para depuração de uma propriedade. Entretanto, essa trata-se de uma heurística, não havendo garantias de que escolherá a melhor área a ser observada, portanto sua configuração deve ser cuidadosamente avaliada pelo engenheiro de validação.

Com isso, o presente trabalho apresenta um novo conceito na escolha de sinais para depuração em silício, o de focar em certas áreas do circuito ao invés de considerar toda a lógica do circuito, bem como um método para escolha automática do foco. A escolha se baseia no monitoramento de asserções embutidas, a partir das quais é possível determinar áreas mais interessantes para serem analisadas. Os resultados apresentados mostram que a arquitetura é viável, entretanto requer maior investigação quanto à escolha das áreas de foco.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The process of design verification plays an important role in guaranteeing the integrated circuit (IC) product quality. Different verification methods are applied during most stages of development of an IC - Figure 1.1.

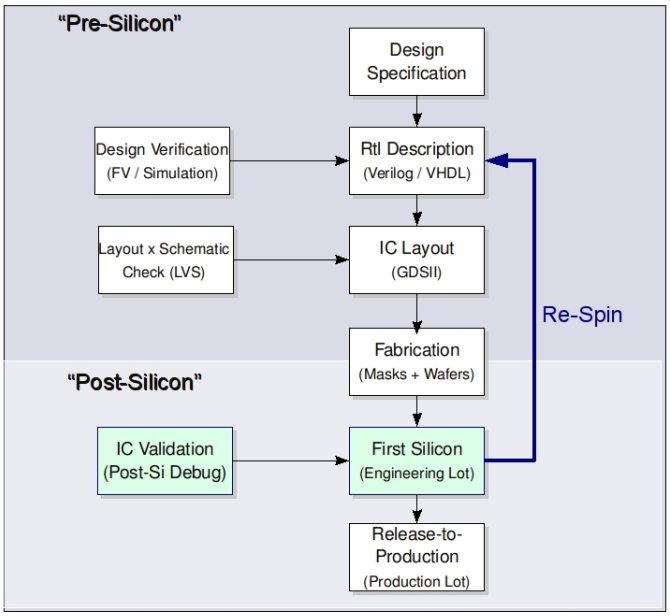

Figure 1.1. IC development stages and verification methods.

These stages are commonly divided into two main phases, pre-silicon and post-silicon, by the production of the first silicon. During the pre-silicon phase, the verification is done using abstracted models of the circuit. The main stages of development and validation are the following:

Design Specification: In this first stage, the expected behavior and the design in-

terface are specified and documented. Concurrent to this stage, verification engineers start to develop functional tests.

RTL Description: Then, the design is implemented using a hardware description language (HDL), usually Verilog or VHDL. This implementation is done at the register transfer level (RTL) of abstraction which is later synthesized to a lower abstraction level. The RTL description is used to design validation in order to find logical bugs, using both formal verification (FV) and simulation.

IC Layout: After the synthesis, files with the layout description (e.g. GDSII) are generated. These files contains data to draw shapes of the layout that represent the electrical components of the circuit. In this stage, electronic design automation (EDA) softwares are used to verify whether the IC layout corresponds to the schematic diagram of the design, method known as Layout Versus Schematic Check (LVS).

The pre-silicon validation techniques mentioned are used to eliminate functional bugs in circuit models. Despite its relevance, for increasing system complexity, existing verification methods are not sufficient to cover all the possible design errors that may exist in the device (Constantinides et al. [2008]).

Inevitably, some bugs still make it onto silicon, especially those that are hard to characterize. They can be very costly once manufactured products are shipped. One famous bug is the Pentium FDIV bug, which cost the company 475 million dollars to replace flawed versions (Beizer [1995]).

It is then necessary to identify any bug that remains in the design as soon the first silicon is available. Therefore, it is becoming a common practice to use the first silicon lot for verification purpose only.

The process of finding, locating and identifying design bugs in the post-silicon phase is known as post-silicon debug. The bug detection is the first step of the post-silicon debug and it can be achieved by running tests with the manufactured circuit. These tests include focused tests to stress one aspect of the die, Random Instruction tests, or even OS-level applications.

Once a wrong behavior is detected, the next step is to locate which IC component triggered the observed error. In this stage, data is collected from the circuit execution and analyzed in order to isolate the failing component. Only after that, the root cause of the error can be identified.

Bugs detected and identified during post-silicon debug can only be fixed at RTL level, which causes a re-spin (i.e., the IC development process goes back to RTL de-

scription stage).

Once the post-silicon debug is completed, the design is released to production and reaches the market.

1.1 Motivation

Only during post-silicon debug is possible to detect certain kinds of bugs, like electrical bugs and logical bugs triggered after thousands of cycles. A practical example is reported by De Paula et al. [2008] as:

"The stimulus to generate the crash (i.e., booting the OS for 30 seconds) is far too deep to replay in simulation or by single-stepping the die, and trying to hit the crash state with full-chip formal verification exceeds the capacity of the formal tools".

However, the increasing demand for shorter time-to-market, combined with increased complexity and performance requirements put tremendous pressure on post-silicon debug, which is usually the last step prior to chip release. Post-silicon debug is thus becoming a potential bottleneck in productivity and cost(Keshava et al. [2010]), causing it to emerge as one of the most important topics for research and the EDA industry.

In contrast to pre-silicon techniques, post-silicon debugging techniques have two main limitations, controllability and observability. The lack of observability is due to a limited number of chips output and scan registers, while the controllability is affected by non-deterministic behavior of the majority of today's chips (e.g. asynchronous communication and multiple clocks domains).

Consequently, the failure analysis and identification during post-silicon debug requires significantly more effort. In this project, we focus on the enhancement of observability, which has two main impacts in post-silicon debug:

Bug Detection: The detection of a bug during manufactured IC validation requires the failure activation as well as its observation. A test can effectively trigger a bug. But due to limited visibility, the failure might not propagate to any observation point and the test will give a false impression of correctness.

Bug Identification: After finding a bug, it becomes necessary to determine its root cause. The effectiveness of the debug strategy relies on which signals are selected to be observed. The selected signals should carry on evidences of the bug (i.e., they should bring enough information to restore the failure scenario).

In order to improve observability during the debug process, design for debug (DfD) modules have been employed to observe internal signals of the circuit under debug (CUD) . The DfD logic is included on the die so it can collect information during real-time executions. A big challenge in this area is the development of efficient DfD modules with low area and power overhead.

1.2 Goals

The main goal of the proposed work is to present a novel architecture capable of increasing internal visibility of a circuit under debug by only monitoring signals significant to a specific failure analysis.

In order to achieve that, we target the following secondary objectives:

Visibility Enhancement: The use of a trace buffer allows the capture of internal signals values during real-time execution and the values captured can be transfered to a computer for further analysis once the execution has finished.

Efficient Signal Selection: The signal selection will be sensitive to the analyzed failure (i.e. it will be previously selected from the assertion's cone of influence).

Automatic Integration: Given a design described in RTL, the system will be able to read the design and its assertions and automatically connect them to the DfD system, as well as select which signals should be monitored.

1.3 Text Organization

The dissertation is organized as follows: Chapter 2 presents the state of art of post-silicon debug and the techniques commonly used to increase internal visibility of the circuit under debug.

Chapter 3 presents an overview of the proposed system, which is detailed in the following chapters. Chapter 4 details the on-chip architecture, while Chapter 5 details the integration software and shows how to exploit the concept of cone of influence in post-silicon debug.

Then, chapter 6 exemplifies and evaluates the usage of the proposed architecture with a case study, where the DfD system developed is integrated to a small processor. In addition, it presents more case studies of the signal selection COI based and assertion clustering.

Finally, chapter 7 discusses the main contributions and limitations of this work and suggests future work to overcome these limitations.

Chapter 2

Related Works

2.1 Methods for Data Acquisition

Debugging an integrated circuit requires the analysis of chip's external and internal behavior while a known set of stimuli is being applied. An efficient debug system needs to provide enough execution control and signal visibility. Several Design-for-Debug (DfD) techniques have been proposed(Vermeulen and Goel [2002]). On the matter of data acquisition, there are two main methods: scan-chain and trace-buffer.

Scan chains are sequential data paths between design registers which are commonly used for testing purposes. Regular registers are replaced by a system composed by a selector controlled by a scan enable signal (`SE`) and the register itself, known as scan registers (Figure 2.1). Besides the regular data input (`D`), a scan input (`SI`) is also used.

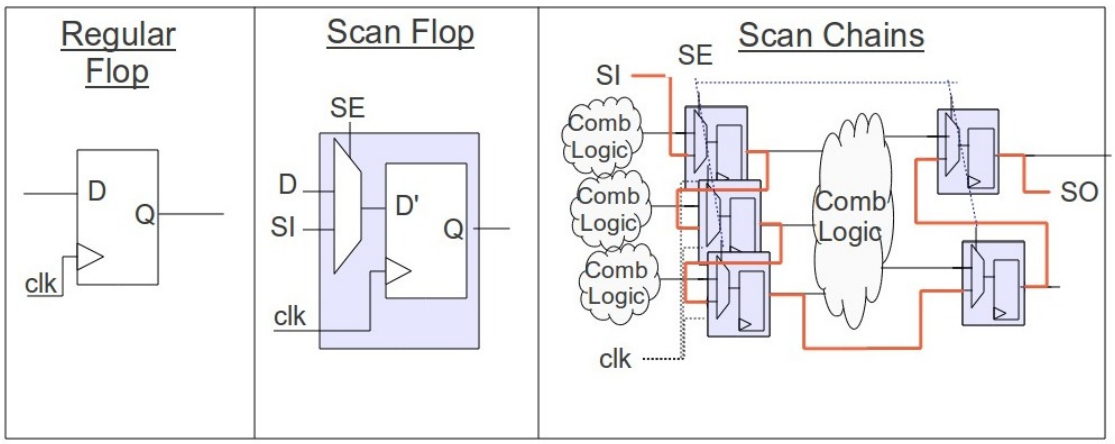

Figure 2.1. Diagrams for a register (a flip-flop D), a scan register and a scan chain, where SE is scan enable, SI is scan input data, SO is scan output data.

The scan registers are then connected sequentially through `SI` ports, creating the scan chain (red path in Figure 2.1). When scan enable is off (logical 0), the scan register works as a regular register and stores `D` value. On the other hand when it is set to on (logical 1), it assumes the `Q` value of the preceding register in the chain.

Although scan chains are actually inserted in the design to increase the observability during manufacturing test, some DfDs reuse them to reveal the chip's state. They first capture most of the internal state elements when a specific triggering event (breakpoint) occurs. In debug mode, the values of scan registers are shifted out through scan chains primary outputs. This can provide very good observability into the state of a chip when it was stopped. Failing elements of a scanned out state can be identified using latch divergence analysis or failure propagation tracing techniques. Although the observability provided by a scan chain is generally very good for a single state, it requires a high time overhead to observe state transitions over multiple cycles.

On the other hand another practical technique consists of a mechanism to access desired signals on-chip and store their history for later read out. Trace-buffer based methods provide real-time visibility to the circuit under debug (CUD) (van Rootselaar and Vermeulen [1999]). In this approach, the signals chosen to be observed are connected to a buffer and their values are captured during chip execution and are retrieved once the execution has finished. This buffer, aka trace-buffer, can be shared with the CUD or used only for debug purposes. Besides the trace buffer, this technique usually requires an embedded controller inside the DfD for sampling internal signal data into on-chip trace buffers (Figure 2.2).

The CUD can be put into operational mode, while the DfD monitors some programmable breakpoints, whereupon data will start to be stored in on-chip trace buffers. After that, the sampled data can be transferred off the chip for further analysis.

The main benefit of this method is that the chip does not necessarily have its behavior altered, and consecutive cycles of data can be captured without slowing the IC down. However, this technique requires a trade-off between the amount of data observed and die area overhead. Consequently, the trace buffer depth will limit the number of cycles that can be stored, while its width will limit the number of signals that can be sampled.

2.2 Signal Selection

In order to handle the lack of visibility, Hsu et al. [2006] presented the first known work that introduced the idea of selecting signals to virtually enhance visibility. It

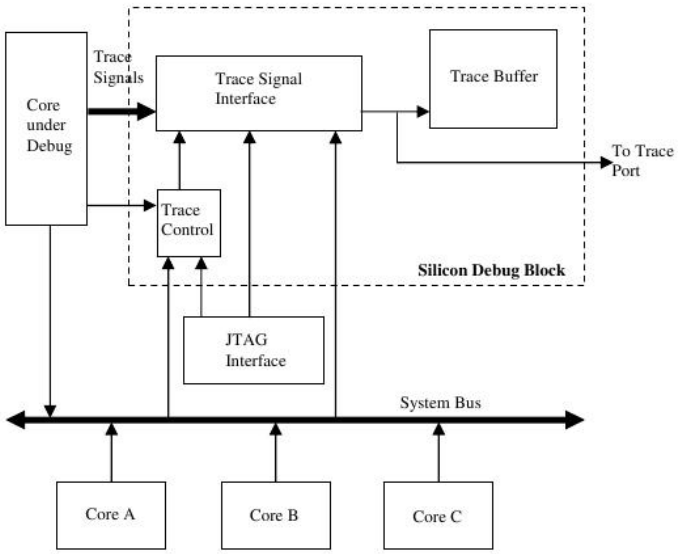

Figure 2.2. Example of a trace buffer based architecture (Prabhakar [2009]). Besides regular connections, one bus is added to connect the monitored signals to the trace buffer or trace port. The JTAG Interface is used to control trace capturing.

introduced some interesting concepts:

Signal Selection: An optimal set of control registers is determined to be monitored based on the capability of data expansion, i.e., all registers necessary to restore the values of the whole signal set are selected.

Data Expansion: Values of non-visible signals are deduced from the captured data based on knowledge of design function.

The proposed technique enables the virtual observation of combinational nodes and has a small computational overhead. Nevertheless, this was just a preliminary study and did not mention the area overhead of the method proposed.

After that, Gao et al. [2009] presented a different algorithm to enhance virtual visibility based on topological relationships. The selection uses an abstract digraph $G = (V, E)$ of the CUD, where the nodes are system registers and the edges connect a pair of nodes if their represented registers are connected through combinational logic. Considering the fan-in set of each node, the problem was reduced to a maximum coverage problem and a greedy approach could be applied. Therefore, the algorithm presented tries to maximize the number of registers that have at least one path to any trace signal. It doesn't evaluate the combinational logics between each pair of registers.

In another approach, trace-signal selection is based on the concept of state restoration (Ko and Nicolici [2009]). The combinational logic is no longer ignored; it is used in an attempt to deduce non-visible registers values based on the type of connection to traced signals. The combinational logic is decomposed into two input primitive gates, so inference functions and restoration probabilities are established for each of them. The restorability of a signal can be made in two ways, backward or forward, which means the deduction can be based on the fan-out or fan-in values.

Once all combinational logics have been evaluated, a greedy algorithm is used to select the signal set that maximizes the circuit restorability. The selection is based on how many other signals will likely be covered after state restoration.

According to Liu and Xu [2009], the previous work mentioned has introduced many relevant concepts. However, its definitions for the gate-level restorability did not have enough theoretical basis. The restorability should be based on conditional probabilities, since it does not represent the chance of a node to assume the values 1 and 0. Liu and Xu [2009] then presented new restorability definitions and formulae. They also presented an algorithm of signal selection similar to the one described by Ko and Nicolici [2009], where the main difference is, in the new algorithm, a signal can only be used in one direction. If the visibility of a gate's output is obtained through forward propagation, it should not be used to further improve the visibility of its inputs through backward justification. Comparing to Ko and Nicolici [2009], the methodology based on conditional probabilities showed a higher restoration ratio.

Other enhancements on the mentioned trace selection algorithms were presented by Shojaei and Davoodi [2010]. The authors proposed a new criteria for the algorithm of signal selection of Liu and Xu [2009]. They also presented a method for determining some critical state elements and created a weight that reflects their reachability. Here, the critical state elements were established to track power-drop issues, though their algorithm does not depend on that kind of critical state elements. This new weight is then used in the algorithm of trace selection combined with the reachability to the rest of the signals. The algorithm generates N trace selection possibilities, where each one provides a unique trade-off between the visibility of critical state elements and the total visibility.

2.3 Use of Formal Methods

Some approaches for post-silicon debug try to overcome the trace depth limit through the use of formal methods. Ho et al. [2009] presented a technique that uses guided

formal verification to build a valid trace that represents a failure scenario of a bug detected during silicon debug. This method is based on the use of user-hypothesized preconditions, called *waypoints* (aka *lighthouses*), of the bug. Once the waypoints are identified and ordered in the expected order of occurrence, the algorithm tries to find a counterexample for a negated waypoint where the start point is the scenario of the previous waypoint. For the first waypoint, the reset state is used. Any model checking heuristic optimized for finding counterexamples can be used to this approach.

One drawback of this technique is that there is no guarantee that the trace to an error signature will pass through any waypoint and it is subject to misleading interpretation of bug the signature. Another limitation is related to larger traces or the lack of identified waypoints. The formal verification tool may fail to find a counter example due to state explosion. One additional limitation of the presented work is that it lies on user-hypothesized preconditions, ignoring any information of the execution that may have triggered the error.

Another proposed method that applies formal verification to assist post-silicon debug, Backspace (De Paula et al. [2008]), enhances the observability of the execution leading up to a detected bug. Its goal is to explain the some buggy state, by creating a “backspace” capability, i.e., it applies formal methods to compute predecessor states and then it generates a trace with them. The resulting trace shall be viewed like a simulation waveform.

The framework proposed consists of adding a DfD with a signature that saves some history information and a programmable breakpoint mechanism that allows the execution of the CUD to be suspended. Essentially, the proposed algorithm "works backwards", as shown in Figure 2.3.

Preliminary results (De Paula et al. [2008]) showed that the framework works and it was able to produce traces with the maximum established length in most cases. Still, this framework needs more improvements so it can be used in the field. Some limitations pointed by Gort [2009] were area overhead, breakpoint bit selection method and coping with non-deterministic executions.

2.4 Embedded Assertions

Assertions are logic statements created by designers that specify expected behaviors of a set of signals. The correctness of the described behavior means that the related assertion should always be evaluated as true. Therefore, in order to check circuit correctness, hardware assertions can be included in the design RTL description. This

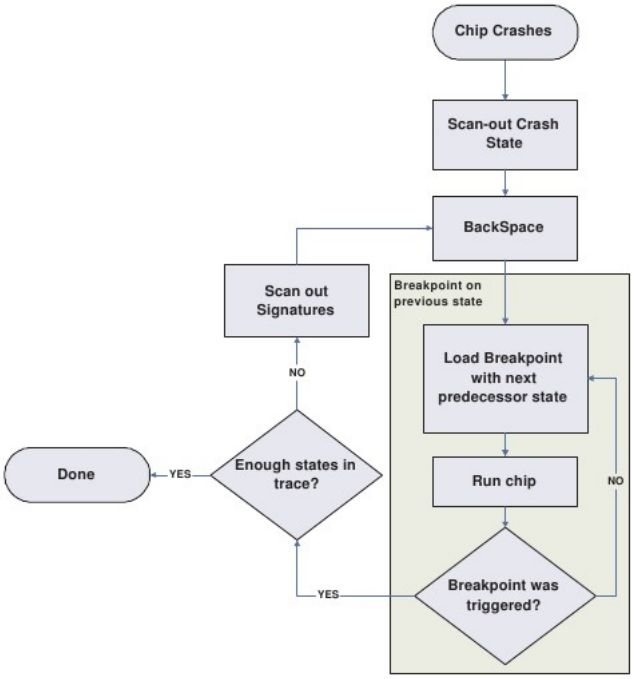

Figure 2.3. Debug Flow using Backspace (Gort [2009])

technique is widely used during pre-silicon validation, by both simulation and formal verification. Because of that, hardware assertions have been incorporated to some verification languages like System Verilog Assertions (SVA) (Acellera Organization [2004]), Open Verilog Library (OVL) by Acellera Organization [2011] and Property Specification Language (PSL) (Accellera Organization [2004]).

On the other hand the use of hardware assertions in post-silicon is still being consolidated. The assertion languages used in pre-silicon validation usually describe circuit behaviors in a higher level abstraction that not always can be easily expressed in a HDL. Due to this difficulty, assertion checkers generators have been developed in the past few years, so the hardware assertions can be synthesized inside the target circuit.

Nacif et al. [2003] presented a library of synthesized checkers based on OVL assertions modified for scan-chain architecture. Boule and Zilic [2007] created a method for generating PSL checker circuits from sequential-extended regular expressions (SEREs) that can be used in hardware emulation and in silicon debug tools. The most widely known commercial tool for usual assertions conversion to their synthesized form is FoCs (short for Formal Checkers) from IBM [2003].

In the matter of post-silicon debug, assertion checkers have been used as breakpoints and for visibility enhancement. According to Geuzebroek and Vermeulen [2008],

the great advantage of assertion checkers over regular breakpoints is that the exact logic is known at design-time, therefore it can be hard-coded to minimize silicon area costs.

An example of DfD based in assertion checkers was described by Abramovici et al. [2006]. It proposes an assertion based debug architecture that uses the assertions firing as a trigger to stop recording of signals in the trace buffer. In this way, the contents of the trace buffer should represent the most recent activity preceding the assertion firing, so it is likely to provide good clues about the misbehavior that made the assertion fail.

In Geuzebroek and Vermeulen [2008], the authors show how to integrate assertion checkers in existing on-chip debug infrastructures. In order to increase visibility, in other words, they present methods to retrieve assertions values during chip execution. Note that assertions in silicon provide observability of internal properties.

For scan-chain based architectures, they suggest a similar architecture from Nacif et al. [2003], although no combinational logic is used. Not only are the assertions used as breakpoints, but they can also can be scanned out once the CUD is stalled through a test access port (TAP). One drawback that Geuzebroek and Vermeulen [2008] mentioned that makes Nacif et al. [2003] more attractive is that a failure can take too long to reach the debugger (Figure 2.4).

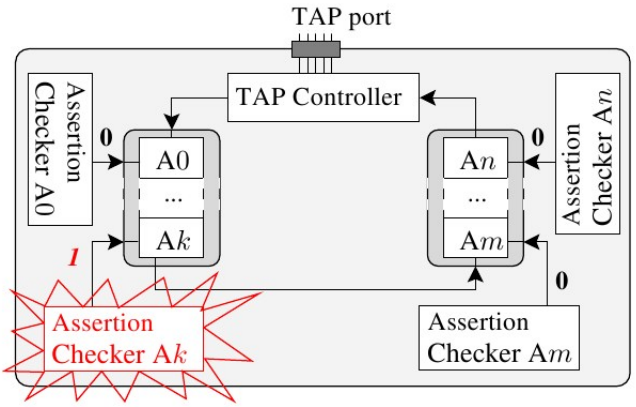

Figure 2.4. Scan based architecture suggested by Geuzebroek and Vermeulen [2008]

Geuzebroek and Vermeulen [2008] also suggested an architecture based in trace buffers, in such a way that violating assertions can be embedded in the trace data. Since the quantity n of assertions can be larger than the trace width w, n assertion violation signals need to be compressed into m bits. Two different approaches were described by the authors. Both of them use a module to dynamically select which data will be sent to the trace buffer. The first one selects the assertion checker to be captured by its `ID`, while the second approach select an entire assertions set to be

captured by the set `ID`.

An improvement to the last architecture was introduced by Neishaburi and Zilic [2010]. In this article, the authors focus on an algorithm to choose the assertions of each set (aka assertion cluster), which tries to group assertions more similar to each other. The method proposed uses the graph partitioning approach, with a digraph $G = (V, E)$ where each vertex, $v \in V$, represents a register or an input / output in the design and each edge, $e \in E$, shows that there is a combinational circuit or wire connecting two registers (Figure 2.5).

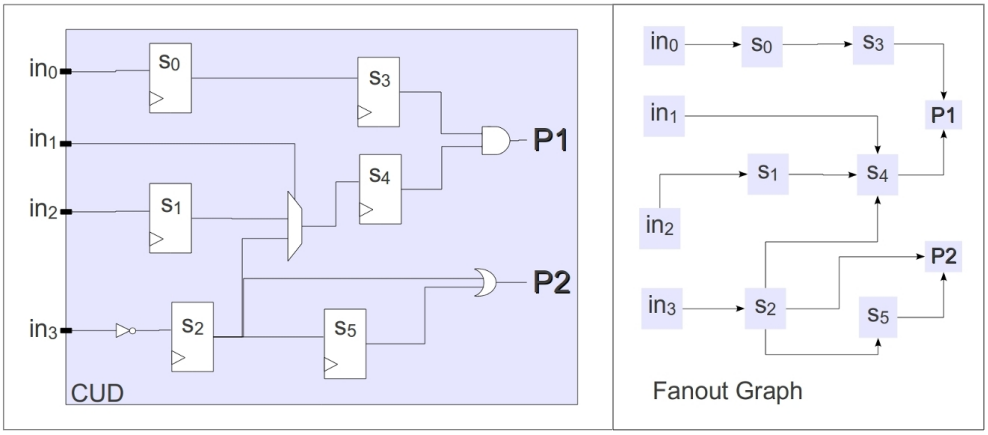

Figure 2.5. Example of how a circuit can be represented in a fanout graph.

For each property, a subgraph of all registers that are connected to an assertion by a sequential or combinational circuit is extracted from G. Then, each iteration of the partitioning algorithm merges the two assertions whose subgraph have more common elements, until a predetermined number of groups is met. Consequently, each assertion of a cluster has similar driving logic.

Chapter 3

System Overview

As already discussed, one of the main limitations of post-silicon debug is the lack of internal visibility of the circuit. Because of that, not only does the detection of the bug become harder, but also the identification of its root cause demands a lot of more effort when compared to pre-silicon techniques. Despite the difficulty faced during post-silicon debug, this validation stage has become essential due to the complexity of integrated circuits (IC) developed nowadays. The more complex the IC gets, the more bugs that pass undetected through pre-silicon validation and the more pressure that is put into post-silicon stage.

In order to increase the visibility of the circuit under debug (CUD), some modules dedicated to debug purposes have been included in silicon chips to retrieve that information when necessary. Two approaches have being widely employed for that purpose, scan-chains and trace-buffers. On the one hand, scan-chains are just reused from silicon test stages and result in a small area overhead, when they don't use redundant registers. But to retrieve the registers' values, this approach requires the CUD execution to be stopped. On the other hand, architectures based in trace-buffers can capture internal signal values without any interruption. However, this requires much more space and the number of signals monitored is limited by the trace width.

Considering the benefits and drawbacks of each technique, we propose a hybrid approach to increase circuit visibility and to detect errors.

3.1 Debug Flow

In spite of its name, the post-silicon debug starts in the early stages of a circuit development, before any chip is in fact manufactured. The implementation of any debug structure is done while the circuit itself is also being developed. In the proposed ap-

proach, this is still the case.

The first step would be integrating the circuit target and the DfD module. Besides, it should be done during the CUD development, and more precisely, in the end of RTL description phase (See Figure 1.1).

First, the validation engineer, a system user, must define which assertions will be synthesized with the design. For example, those that have already been proven true should be removed. However, the definition of which assertions should be embedded are not in the scope of this work and the user must provide a list of the assertions that should be embedded.

Once the RTL description of the design is ready and the assertion list has been established, the connection software can be used to select which signals will be monitored according to each potential assertion failure. During this stage, the user must also choose whether to join the assertions in clusters or not. The user can limit the maximum number of clusters or limit how much alike COIs must be to join the properties. Different configurations can be tested in order to chose the best one, considering the cost benefit between visibility enhancement and area overhead.

After that, new RTL code will be created and ready for synthesis. The complete flow can be visualized in Figure 3.1

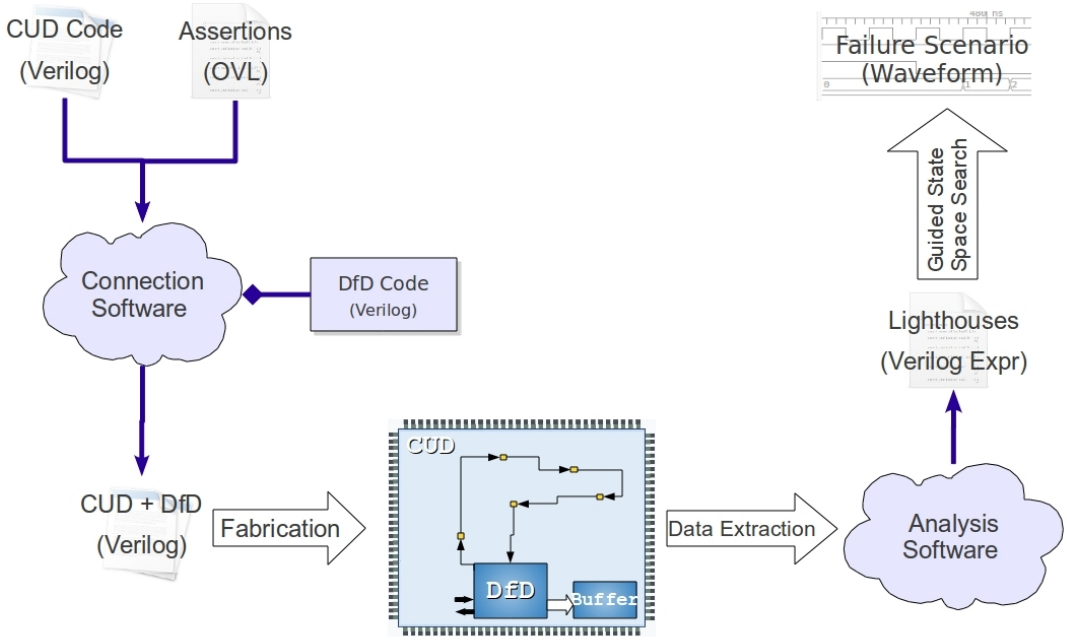

Figure 3.1. The use flow of the proposed system

The next step is during post-silicon validation. The DfD module should be put into monitor state so it can detect any assertion failure and promptly stop CUD execu-

tion. In the end of the monitor stage, an identification number of the failing assertion will be written into the trace buffer, which can be extracted by an off-chip system.

To debug the identified failure, the test that triggered the error must be run again. After the detection of a failure, the DfD module is able to reconfigure itself into a capturing state. Nevertheless, the user can also configure the DfD system execution state and the assertion that should be monitored manually.

During the second execution, the DfD unit will capture the state of some internal signals for a later data exchange. That execution should finish like the previous run, after the failure has been detected and identified. If no failure was identified, the test should be re-run.

After capturing execution, the data stored in trace buffer should be extracted. Finally, the raw data can be interpreted and converted into lighthouses by the analysis software. The lighthouses are used for directed state space search, which can be done using any simulation or verification tool, in order to restore the failure scenario.

Due to the requirement of two executions, the proposed system is especially suitable for debug of functional bugs, which is more likely to be reproducible. Non-deterministic failures are a challenge for most debug systems already developed and so is it for this system.

One way to overcome this problem is to always configure the system to capturing state. If no failure has been identified yet, a group of signals not sensitive to any assertion can be captured and used to reconstruct a valid trace to find the root cause of the error. This would make the system behave like regular capturing devices. However, it would still report which assertion was violated.

3.2 Design for Debug

Assertion checkers may be embedded in the circuit to detect wrong behaviors and invalid states. In cases where they are used, we decided that a scan-chain should be employed to connect all of them sequentially. Since the assertions' values must be extracted only when a checker is triggered, the execution can be stopped without any loss of information of the bug's root cause. Therefore, we minimize the area overhead and we can still monitor all the assertions embedded in the CUD.

Then, a trace-buffer approach is applied to store the values assumed by internal signals. Although a trace-buffer results in a bigger area overhead than scan-chains, its capacity of storing many different cycles during real time execution helps in the reconstruction of the failure scenario. Still, a small number of signals can be monitored

and they should be chosen carefully. The monitored signals must carry some evidence of the bug; otherwise debugging will be inefficient and it will probably fail to identify the bug's cause.

Finally, a design for debug (DfD) module was developed to integrate the identification of the assertion that has failed, acquired by the scan-chain, to a system that chooses which signals must be captured by the trace buffer in the next execution. So, according to the failure identified, a specific group of signals can be captured. The main idea of the debug system is depicted in Figure 3.2.

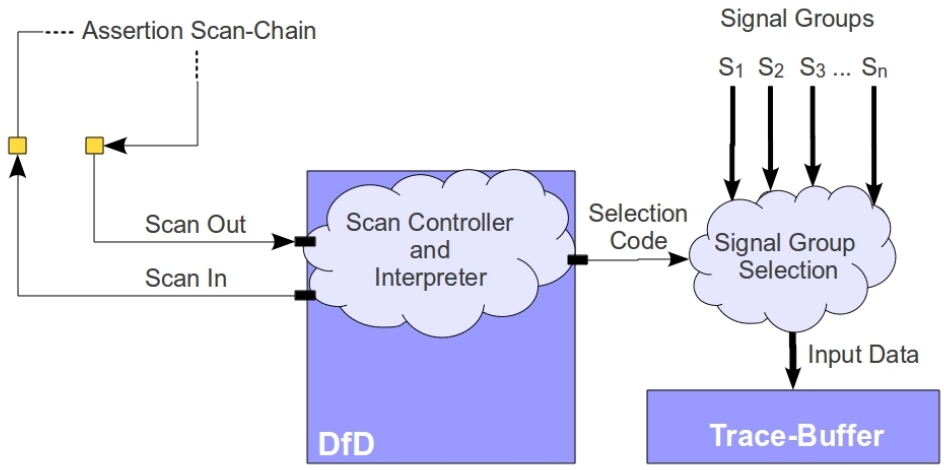

Figure 3.2. Diagram of hybrid approach that exploit both, scan-chains and trace-buffers techniques, and that integrate them to capture more relevant signals to failure analysis.

The design for debug module is also responsible for the communication to an off-chip system, which in turn will be responsible for enabling and configuring the DfD system execution and also for extracting the data stored in the trace buffer.

3.3 System Integration

The efficiency of the proposed debug system relies on the correlation between the failing assertion and the group of signals that will be monitored. The group selected should be the one that best reflects the failure cause and that optimizes the restoration of the failure scenario.

Thus, the proposed work was not limited to the development of an on-chip debug structure capable of enhancing the CUD visibility. It investigates how to establish which signals should be monitored to explain a specific assertion failure, i.e., how a

good set of signals can be selected just by knowing the assertion statement and circuit's logic.

The mechanism proposed was based on the concept of cone of influence (COI) of a property. This concept is commonly applied to reduce the search space during formal verification by considering only the signals that have some potential influence in the property statement. We believe that this reduction of the search space can also be applied in post-silicon debug in order to optimize trace buffer usage. Monitoring one or more signals that has no influence in the property statement would be a waste of space, which is limited to a low number of signals.

Therefore, an algorithm was created to employ cone of influence reduction to choose a set of signals that is more likely to carry an evidence of the root cause of an assertion failure. One distinct group of signals can be selected for each assertion.

Note that the on-chip system was created to select one group of signals to be monitored among all groups created in the earlier stage of development and it is not capable of changing the group composition. Because of that, the selection of signals that compose each group should be done in an early stage of development, before tape-out. This approach also minimize area overhead required to route signals into the signal selector module. More connections would be necessary to support dynamic composition of signal groups, since all signals would need to be connected to the DfD module.

For that purpose, a connection software was developed. This software implements the signal selection based on the cone of influence of the embedded assertions and it is also capable of gathering the assertions in clusters. In addition, the developed software generates new RTL code for the circuit under debug including all DfD logic and connections.

The user can configure the DfD logic through the connection software and the RTL code generated is ready to be used in synthesis.

Additionally, a second software was implemented to translate the data extracted from the trace buffer. It reuses the configuration files created to guide the system connection and it is capable of translating the raw data acquired. This analysis software maps the values captured to the state of the signals monitored and creates one property for each sample that leads the reconstruction of a possible execution to the assertion failure.

Chapter 4

On-Chip Architecture

In this chapter, we present an architecture capable of detecting and identifying assertion failures inside the circuit, and moreover, a system capable of choosing which group of signals to be captured according to the failure detected. Our debug architecture employed a scan-chain to identify errors and a trace-buffer to capture the state assumed by some internal registers during circuit execution. In order to integrate the information retrieved by both, the architecture required the implementation of a design for debug module.

4.1 Design For Debug

The DfD system developed is responsible for extracting relevant data from a target circuit, the circuit under debug. The DfD logic must be included in the same chip as its target and requires a connection to a trace buffer, to the CUD itself and to embedded assertions. An interface through a functional boundary scan (FBS) is also required in order to to configure the DfD module behavior. The DfD module interface is detailed in appendix A.

Functionally, the DfD module can be divided into three functional modules according to main tasks performed:

Assertion Processor: This module manages the signals received from the hardware assertion checkers. Its function is to detect any assertion failure as well as to identify which checker has failed.

Controller: This block is used to configure and control debug execution. It can enable and disable assertion checkers and trace capture. Additionally, the controller

selects which group of signals will be monitored, based on the failure detected by the assertion processor.

Trace Manager: This block is the interface with the trace buffer and controls its access by external user (see appendix A.2). During debug execution, trace manager uses a circular buffer to store each sample of the circuit state. If the number of samples exceeds the buffer's depth, the oldest capture will be replaced. Moreover, the interval between samples can be configured.

Figure 4.1 shows a block diagram of the DfD logic and its interfaces.

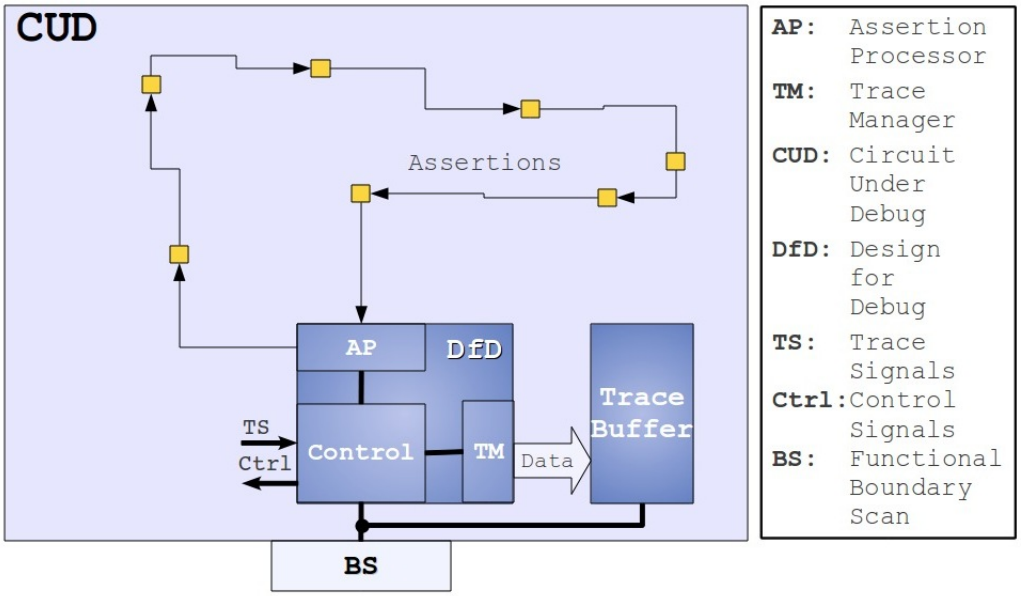

Figure 4.1. Block diagram of the entire design for debug logic and interface.

4.1.1 Execution Modes

The developed DfD system can be set to four different execution states:

Idle: The entire DfD logic and the assertions are disabled, reducing power consumption.

Monitor: The assertions are monitored and any failure is promptly detected and then identified. Only the assertions and its processor are enabled. Note that in this mode, any assertion failure will force the CUD to halt.

Capture: Not only are the assertions monitored, but also the signals are being captured. All DfD modules are enabled in this state.

A regular testing and debug execution flow starts with the configuration of the DfD system to the Monitor state. Once configured, many tests may be executed until any assertion is triggered. The failure will be automatically identified by the assertion processor module and the system will be ready for the re-execution. During the second run, the group of data related to the failing assertion will be stored in trace buffer.

Finally, the data can be extracted from buffer. Along with the captured values, the identification number (`id`)of the monitored assertion and other execution informations (see Appendix A.2) are also written to the trace buffer, in order to help the user interpret the execution result.

The configuration of DfD system execution mode can also be done before any debug session. The trace buffer has one predetermined position for system configuration that must be written with the configuration data (More details in Appendix A.2)

4.1.2 Capturing Configuration

One main limitation of a trace buffer based architecture is that the length of the failure scenario that can be observed is limited by the buffer's depth. A workaround to virtually extend the observability window is to introduce some interval between each sample. After that, some tools for post-processing data (e.g. a Formal Verification tool) can be used to create valid transition scenarios between each sample and, therefore, reconstruct a failure scenario longer than the trace depth.

In that case, there's no guarantee that the trace reconstructed is the same scenario that caused the failure observed, but it still has to be a valid scenario that points to the same failure. One drawback of this technique is that the tool used for post-processing may fail to provide a valid path between the snapshots.

Because of that, the DfD module developed allows the user to choose which strategy he/she prefers to adopt before the debug session. The user can configure the DfD module to store continuous or non-continuous samples (see Figure 4.2). The capturing configurations are classified as:

Continuous Record: A sequence of d cycles is recorded ($C0$). In this case, a post-processing tool must be able to restore the values of signals not captured in the trace based on the verilog/vhdl description.

Lighthouses Record: The DfD module is configured to take snapshots only at the end of an interval i or once a failure was detected. Those snapshots can be used as lighthouses and a failure scenario can be recreated using formal or semi-formal verification tools ($C2$).

Hybrid Record: Initially uses lighthouses record, but in the end a continuous record takes place ($C1$, $C3$). This approach tries to provide information of a longer execution than the continuous record, but at the same time it should provide more details on the moment the failure occurred. This last approach is just an algorithm that will decrease the interval during the execution and it may not coincide with the failure occurrence.

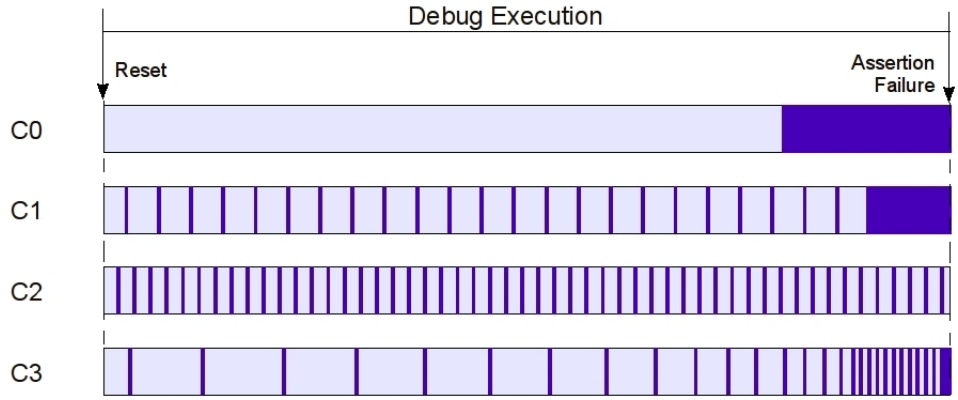

Figure 4.2. Different sample configurations.

Logically, this configuration will only have any effect during the "capture" state, where the interval between the samples depends on system configuration. In spite of its influence, the interval can be configured at any time and it will persist until the system is reset or reconfigured.

4.2 Assertion Checkers

The assertion checkers employed in this work are similar to those presented by Nacif et al. [2003], where all assertions have, in addition to their regular testing interface, a register to store the value of the assertion in the previous cycle, an interface for the scan-chain and one signal for error detection logic.

Table 4.1 contains the description of the signals in the assertion interface.

Figure 2.1 shows how a scan-chain works and the assertion checker does not differ from that. Inside each checker there is a scan-register that can receive the value of the property checker or the scan input, i.e., the value of the preceding register in the chain. Some combinational logic for error detection was also included inside the checker and the signals `EI_N` and `EO_N` represents the input and output of that logic. Therefore,

Signal	Type	Description
CLK	Input	Global clock signal. All registers are sampled on the rising edge of the global clock.
RST_N	Input	Global reset signal. This signal is active low.
TEST_EXPR	Input	The expression logic that must be tested.
ESCEN_N	Input	Error scan enable. This signal is active low.
EI_N	Input	Error input. This signal is active low.
ESCI_N	Input	Error scan input. This signal is active low.
EO_N	Output	Error output. This signal is active low.
ESCO_N	Output	Error scan output. This signal is active low.

Table 4.1. Description of signals from assertion interface.

the chain of assertion checkers are represented in Figure 4.3 as well as the structure of the checker itself.

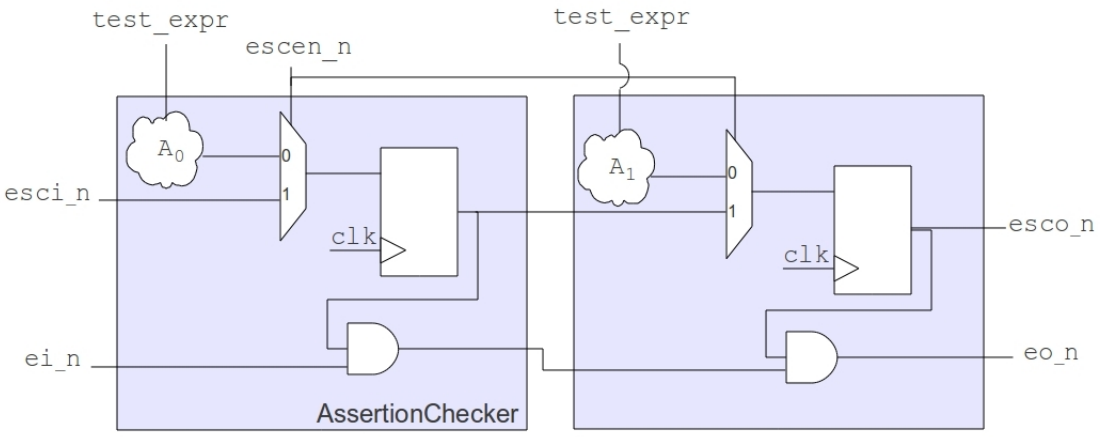

Figure 4.3. Different samples configuration.

Note that the chain of AND gates that compose the combinational logic is in fact optimized during circuit synthesis in order to avoid performance overhead. During the synthesis targeting emulation with a Field-Programmable Gate Array (FPGA), the resulting circuit is composed of a multi-stage AND-tree logic with depth of $\lceil \log_2 n \rceil$ for n assertions.

However, the multi-stage AND-tree may impact the number of allowed combinatorial logic levels for a given target chip clock frequency. In the case of IC synthesis, a different technique based on pseudo-NMOS gate can be used to optimize the presented combinational logic.

A pseudo-NMOS gate consists of an NMOS (N-type metal-oxide-semiconductor) pull-down network for the implementation of the logic function and a simple load

device, while the regular CMOS (complementary metal–oxide–semiconductor) gate has a combination of active pull-down and pull-up technology. A good advantage of a pseudo-NMOS gate is the reduced number of transistors, therefore it has a smaller area overhead. A pseudo-NMOS gate requires $N + 1$ transistors to implement an N-input logic gate, which is almost the half of the $2N$ transistors required by the complementary CMOS. However, a major disadvantage is the dissipation of static power when the output is low (See Rabaey et al. [2003] for more details).

Besides the reduced number of transistors, some pseudo-NMOS gates have a much lower propagation delay and so they are suitable for large fan-in circuits. For example, an pseudo-NMOS NOR gate is composed of one load transistor and one transistor per input disposed in parallel (Figure 4.4).

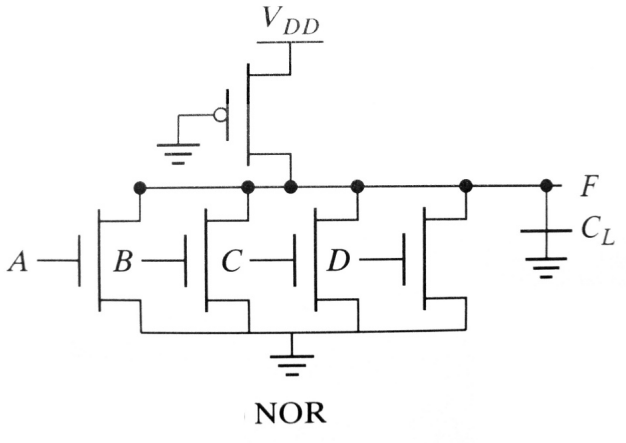

Figure 4.4. Schematic diagram of a four-input (A-D) pseudo-NMOS NOR gate (Rabaey et al. [2003]).

The parallel disposition of the transistors makes the fall time fast, despite the number of inputs, and allows us to implement the AND-tree without impacting on the IC performance.

Therefore, the combinational circuit presented before (Figure 4.3) can be modified in order to use the pseudo-NMOS NOR gate. Figure 4.5 depicts the new combinational logic using the NOR gate.

Note that in our application, although there is static power consumption when one of the transistors is triggered, in practice this static power consumption will only occur in the rare event of an assertion failure.

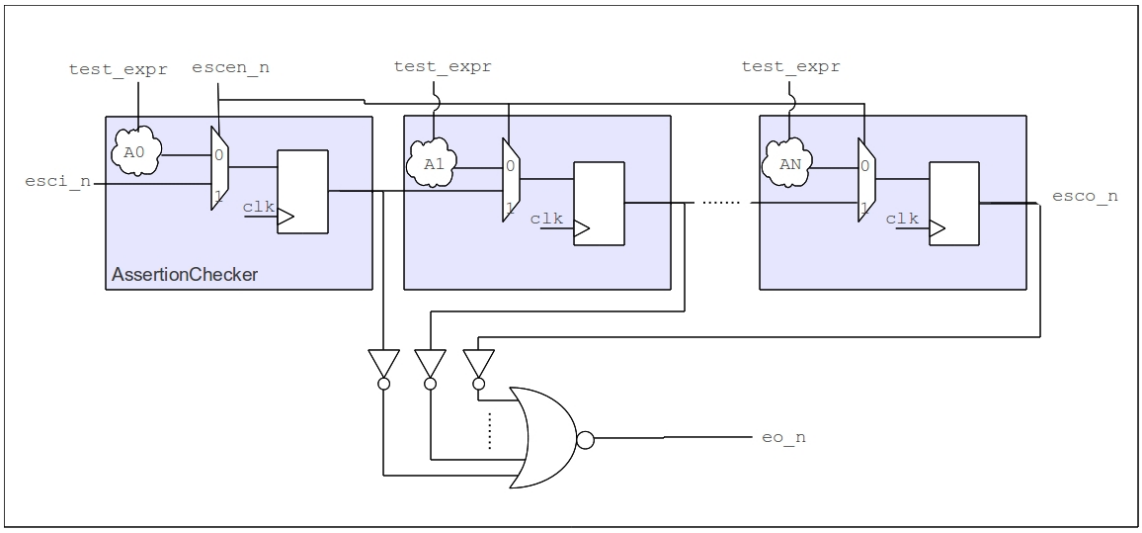

Figure 4.5. The combinational logic that connects the assertion checkers is changed in order to use one pseudo-NMOS NOR gate. The output signal `eo_n` of each checker is negated before the NOR gate.

4.3 Trace Buffer

The trace-buffer is in fact a regular circular buffer controlled by the trace manager module to store the values assumed by some internal signals in real time execution. However, the trace manager module was developed to allow external access to the trace buffer. Consequently, the trace buffer is also a mechanism for allowing the communication with the external controller and the design for debug module.

For the purpose of allowing communication between the external controller and the DfD module, some trace-buffer positions are reserved for communication purposes and they do not keep trace values. The communication protocol together with the definition of reserved positions are detailed in the Appendix A.2.

This communication usage is not transparent to the user when extracting the data from the trace manager. However, an interpretation software was developed for this transparency (Section 5.4).

4.4 System Behavior

The main transactions performed by the DfD module are the detection and identification of a failure, besides the selection and capturing of one group of signals according to an identified failure. The first transaction is mainly performed by the assertion processor, although the controller must stall the CUD so there won't be any interference in the analysis due to invalid circuit state execution. The second one is performed by

the controller and the trace manager.

4.4.1 Failure Identification

Before any information is saved in the trace buffer, the target assertion should be identified in order to optimize which group of signals would better represent the debug scenario. The debug system implements an automatic mechanism to detect and identify any failure that occurs during CUD execution (Figure 4.6).

If any assertion fails (Figure 4.6 (B)), the DfD module sends a stall or halt to the CUD, so it stops its execution. In the next cycle, the assertion processor (AP) starts to scan-out the values of the assertions using a scan-chain mechanism and an internal counter (Figures 4.6 (C) and 4.6(D)). At each cycle, the values of the assertions are moved to the next assertion in the chain and the internal counter is incremented. Once the failure value reaches the scan input of the AP, the scanning process finishes and the value of the internal counter, i.e. the *id* of the failing assertion is sent to the controller and the trace manager (TM).

The TM instance writes in the buffer the captured *id* while the control decodes that information in order to choose which signal group should be monitored in the following execution. During the second execution (Figure 4.6 (E)), a certain signal group is monitored according to the failure detected. The execution will end once the assertion checker is triggered again (Figure 4.6 (F)).

The mechanism used for signal selection will be presented in Section 4.4.2, where the group *id* can be either the assertion *id* or the cluster *id* to which the failing assertion belongs. It will depend on whether the user decided to use assertion clusters or not.

4.4.2 Signal Capturing Sensitive to Assertion Failure

The main idea of this work is to dynamically select which signals should be monitored according to an assertion failure. Note that the definition of the function or map to convert the assertion `id` to the signal group `id` is done in pre-silicon stage (see Section 5). The system is capable of identifying a failure, translating that information and capturing the correct group of signals without any other configuration between executions.

For small number of assertions, a simple and intuitive way to implement the selection of signals according to an assertion failure is by using the assertion `id` to control which group will be monitored (Figure 4.7). Therefore, the group `id` would be equal to the assertion `id`.

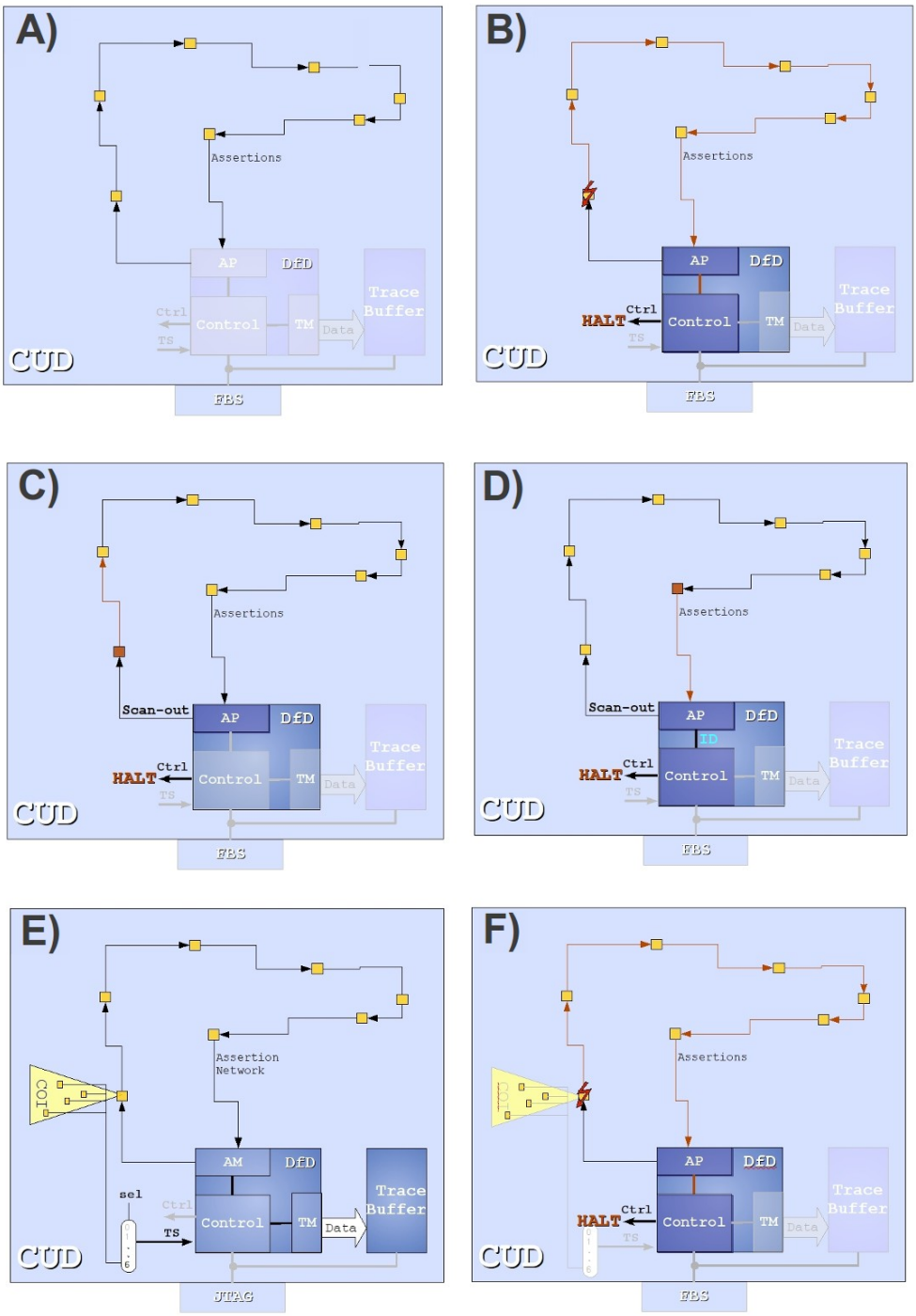

Figure 4.6. Sequence of events starting with error detection, then going through error identification and a new CUD execution, with the right group of signals being captured.

Although intuitive, this would be a naive approach considering the IC complexity nowadays, since the proposed architecture would have a great area overhead imposed

by the on-chip circuitry. Hence, capturing one specific group of signal for each assertion embedded in the circuit is unreasonable for most systems, since they may have hundreds or even thousands of assertions. But this issue can be overcome if the assertions are joined in clusters.

In the case where the concept of assertion clusters are employed, the signal group is selected based on the cluster *id*. So the implemented system can be easily configured to support mapping each assertion *id* to the cluster *id* it belongs to or even support the definition of functions to determine the cluster *id* according to the assertion *id*. Despite the mechanism chosen, the architecture behavior is the same and the controller will translate that information so that the proper group of assertions is monitored (Figure 4.7).

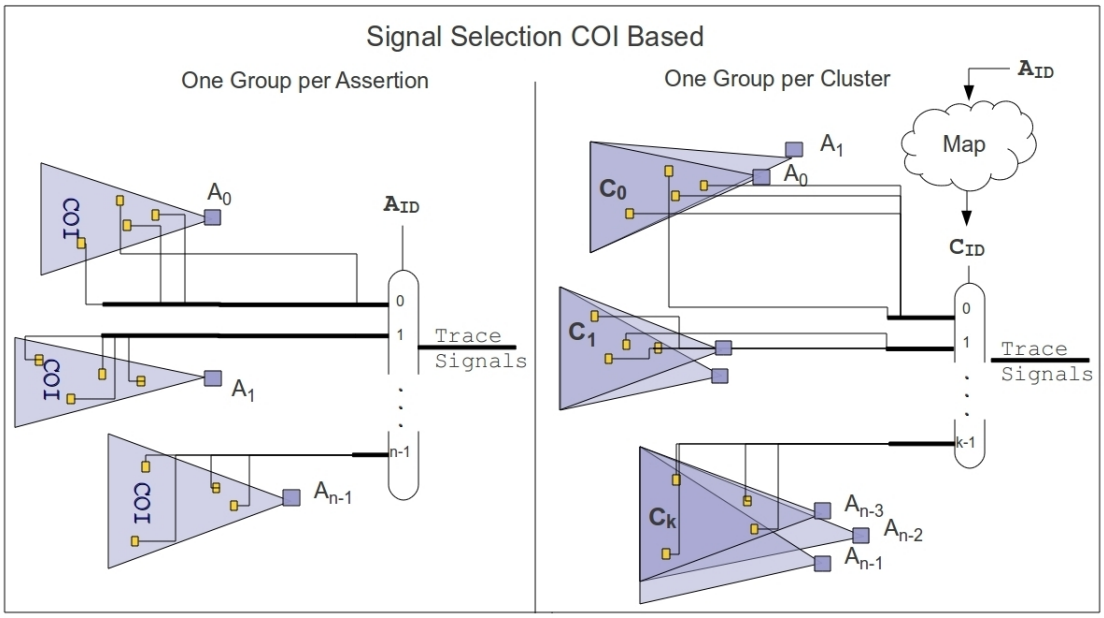

Figure 4.7. Different mechanisms for signal selection with and without clusters. A multiplexer is used to select which group will be monitored. The control is performed by the assertion `id` signal or cluster `id` signal respectively.

The signal capturing transaction starts right after the identification of a failure by the assertion processor(AP). The AP sends the information to the controller module, which translates the information acquired to the `id` of the group signal that must be captured. That information is stored inside the controller so it can be used in the next execution. Once the user restarts the CUD, the controller sends the group `id` (C_id) of the signal that must be captured to the selection structure, either a multiplexer or a bus. The correct data will then be transmitted to the trace manager, which will be responsible for determining when the captures will happen based on the previous inter-

val's configuration. The position of writing will also be determined by trace manager. The entire transaction flow can be visualized in Figure 4.8

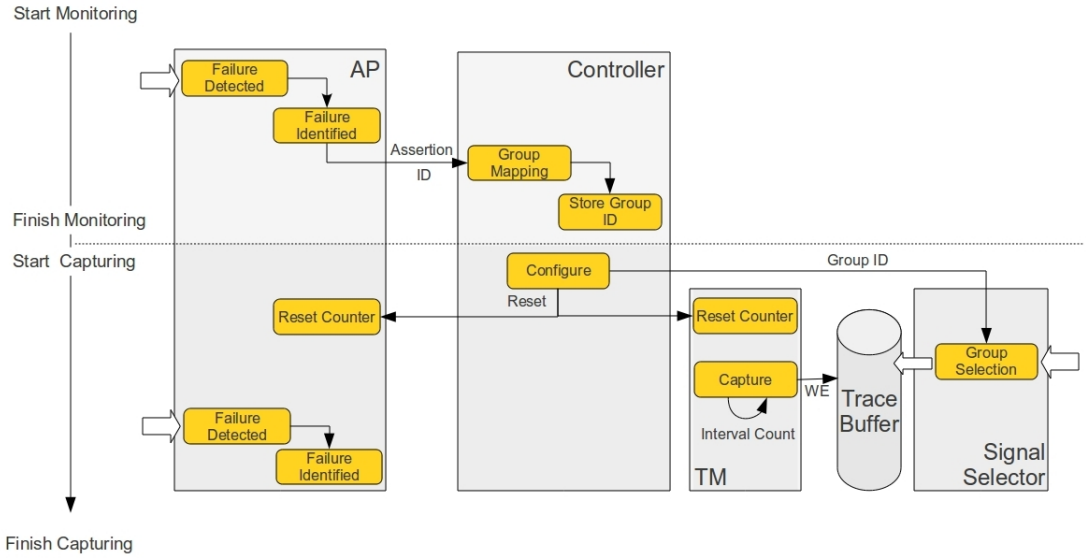

Figure 4.8. Representation of the events that compose the signal capturing transaction.

Although the logic used for signal selection on-chip is multiplexer-based, that approach should only be applied to a small number of signal groups and it was chosen in this work for simplification matter. The architecture proposed here would also work if a unique bus were used, where each signal group has an associated `id`. The group `id` related to the failing assertion would be broadcast by the DfD module and it would determine which group should send data through the bus.

4.5 Synthesis Configuration

The design was projected in such a way that it can have different attributes configurable by the validation engineer. Some of those configurations can be done during the system synthesis via instantiation parameters, such as:

1. Trace width and depth.
2. Number of Assertions.
3. Number of Clusters.

The signal selection and the mapping of assertions to signal group are automatically generated by the integration software. They were implemented in separate

modules to allow manual editing. They can be easily replaced before synthesis according to engineer preference (e.g. the signal selection can be done through a bus instead of a multiplexer structure). The interface of both modules is detailed in Appendix B.

Furthermore, the architecture described in this chapter was developed to be flexible enough to allow its integration with different designs and configurations. However, that also requires a more complex configuration that goes beyond the synthesis parameters and module replacement. Due to this requirement, software was created to configure the other design for debug aspects, such as assertion clusters, correlation between each assertion and the group of signals to be observed and more. A more detailed description of these configurations can be viewed in the following chapter.

Chapter 5

Integration Software

In order to integrate the DfD logic into the design of the circuit under debug, we developed software that receives as an input the RTL description of the circuit under debug and a set of assertions to be monitored already embedded in the CUD code.

Additional software was created to extract relevant information from trace data. This software gathers the values captured in one group for each cycle and maps them to the respective signal monitored and state.

5.1 Connection Software

The connection software was developed to generate a hardware description with the circuit to be debugged augmented with the debug architecture and the design for debug module. The main tasks performed by this software are:

Signal Selection: The selection of which storage elements of the CUD that should be monitored in case some assertion fails. The algorithm of signal selection is sensitive to the context of the embedded assertions and their cone of influence.

Assertion Clustering: The assertions can be gathered in groups according to the similarity of their cone of influence.

DfD Module Instantiation and Connection: The RTL code of the DfD module and the trace buffer must be integrated with the CUD code, in addition to their instantiation. The connection of the selected signal groups is done through a signal selector module controlled by the DfD module. The assertions are also connected to the debug module, employing the connection presented in Section 4.2.

Figure 5.1 shows the connection flow, which involves the analysis of the target design and cone of influence extraction, signal selection, the creation of auxiliary files and other tasks.

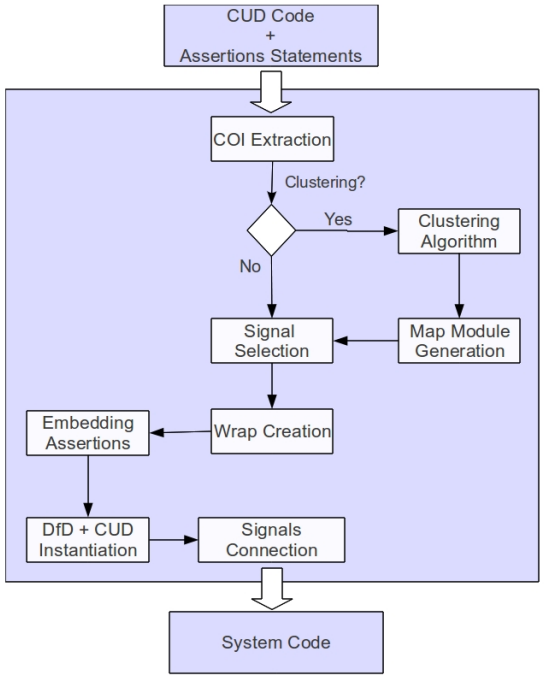

Figure 5.1. Diagram of the execution flow of the connection software. From the CUD's code and the assertions statements, it generates an RTL description of the system integrated and ready to be synthesized.

Note that, the clustering algorithm is ran only if the user choose to join the assertions in groups.

5.2 Signal Selection

One important aspect of the system proposed is how signals are selected to be captured in a specific debug session. Since we are dealing with assertion failure analysis, conceptually, we know that the root cause of any failure must be inside the cone of influence of its assertion.

5.2.1 Cone of Influence Reduction

The concept of cone of influence is widely used in formal verification due to the fact that the main challenge of formal verification is the state explosion, which restricts the complexity of the system to be verified. Thus, to verify more complex designs,

abstraction techniques are employed to build reduced models of those designs. Abstract models are generated removing components that are irrelevant to property analysis.

One common abstraction method implemented under the hood of all commercial formal tools is the cone of influence (COI) reduction. This technique is an effective and safe state reduction that selects the set of signals to be considered during the proof of a property (Perry and Foster [2005]).

In a nutshell, the cone of influence reduction attempts to minimize the size of the state transition graph by eliminating variables that do not influence the variables in the specification of the property. Since this technique maintains all state variables that potentially affect the property, aka cone of influence (Figure 5.2), the reduction will have no influence on the proof result. Thus, it is a safe abstraction.

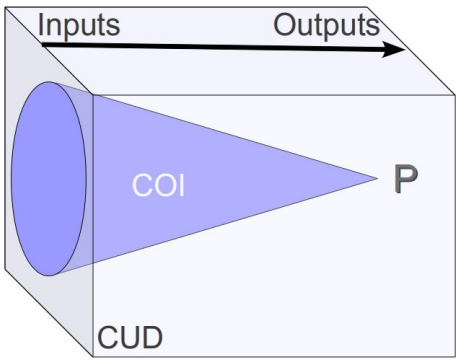

Figure 5.2. Cone of influence representation, where all signals in the blue area compose the COI of the property A which is used by the proof.

The formal definition of cone of influence was presented in Clarke et al. [1999]. Consider that the set of all variables of a given circuit is S and any variable $s_i \in S$ is defined as a boolean function $f_i(S)$, that is:

$$s_i = f_i(S) \tag{5.1}$$

Then the cone of influence of a property A is the minimum subset S' such that:

- $S' \subseteq S$
- if $s_i \in S'$ and it's boolean function f_i depends on s_j, then $s_j \in S'$.

The algorithm to compute the cone of influence is straightforward and can be easily obtained using a fanin graph representation of the original design. The fanin graph is a digraph $G = (V, E)$, where each vertex $s \in S$ represents a variable of the

circuit and any edge $e_{ij} \in E$ stands for a pair of a signal s_i and one of its direct driver s_j (Figure 5.3).

Using the fanin graph representation, the problem of finding the cone of influence S'_i of an assertion A_i can be reduced to the reachability set problem. In other words, the cone of influence is the set of all vertices reachable from any variable s_k used in the assertion specification.

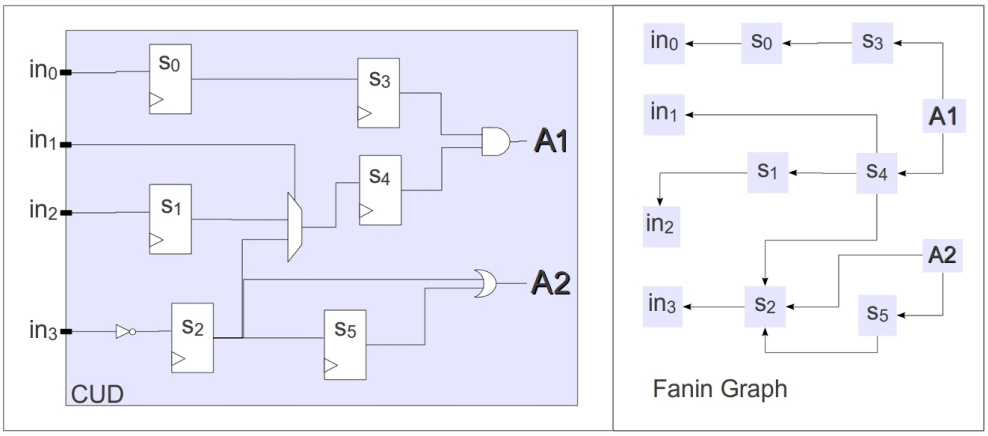

Figure 5.3. Example of how a circuit can be represented in a fanin graph.

If we consider the circuit in Figure 5.3 and its fanin graph representation, the properties A_1 and A_2 will have the following cone of influence:

$$S'_{A_1} = \{s_4, s_3, s_2, s_1, s_0, in_0, in_1, in_2, in_3\}$$

$$S'_{A_2} = \{s_5, s_2, in_3\}$$

5.2.2 COI Based Signal Selection

Different signal selection algorithms have already been proposed to increase observability of a circuit under debug (Section 2.2). All of them try to increase visibility choosing one group of signals T that are going to enhance data expansion, where the size of T is equal to trace width w. Thus, they try to select w signals in the entire design that can be used to increase the number of deducted values of non-visible signals. From now on, we will denote U as the group of all virtually visible signals.

Since the visible signals set may only represent a very small percentage of the signals in the design, the data analysis can fail to reproduce some failing scenarios. The design coverage (C) provided by a set of observed signals is defined by the relative amount of visible signals as stated in formula 5.2.

$$C = \frac{|T| + |U|}{|S|} \tag{5.2}$$

Using the concept of cone of influence, we know that some parts of the design may be irrelevant for the behavior of some assertions. Consequently, it is possible to reduce the search space to S'_i for an assertion A_i. However, the set of visible signals, virtually or not, can also contain irrelevant signals. Therefore, the number of visible signals that are relevant for an assertion A_i will depend on the intersections with S'_i and they may have a cardinality smaller than the original sets.

So the coverage of an assertion A_i provided by the selection of one unique group of signals is given by formula 5.3.

$$C'_i = \frac{|T \cap S'_i| + |U \cap S'_i|}{|S'_i|} \tag{5.3}$$

Although the state space of the analysis decreases according to the size of the cone of influence of the failing assertion, the visibility enhancement proportioned by T can also diminish.

Because of that, we propose an approach capable of taking advantage of the reduced size of the cone of influence and the entire width of the trace in order to increase the coverage of all assertions.

Briefly, the proposal is to instead of selecting one group of signals T to be captured, we define one group T_i for each assertion A_i.

The definition of each T_i is done using any of the algorithms found in the literature, where instead of analyzing the entire circuit logic, only the logic inside the cone of influence of A_i is considered. Then, the size of T_i is equal trace width, w, and the new coverage obtained for the assertion A_i is optimized to:

$$C''_i = \frac{w + |U_i \cap S'_i|}{|S'_i|} \tag{5.4}$$

where U_i is the set of non-visible signals whose values can be deducted from the signals in T_i.

Furthermore, since the algorithm used for signal selection is the same, the size of the set of virtually visible signals shall be greater than or equal to the set previously found, which can be easily ensured.

Consequently, by choosing the group of signals to be captured using only those inside the failing assertion cone of influence, we are maximizing both, the number of relevant signals being captured and the number of signals virtually visible.

Since the focus of the present work is not to evaluate signal selection algorithm itself, the method implemented was similar to the one presented by Gao et al. [2009]. In our algorithm, the combinational logic between each register is also ignored. But the bigger the fanout of a register is, the better candidate it is.

Therefore, for each embedded assertion, we run the signal selection algorithm but consider only the cone of influence of the current property. In the end, $|A|$ groups of signals are created. where $|A|$ is the number of assertions in the design.

5.2.3 Signal Selection Based on COI Hierarchy

A second approach was established considering the module hierarchy of the design. This approach also derives from formal verification techniques. When a design is too complex for entire analysis, a verification engineer applies his/her knowledge of the design and the property statement, as well as his/her knowledge about the development and testing procedures, to choose which areas are more likely to have a bug that would affect some assertion. We decided to also use that knowledge, aka hierarchical verification (Wong [1985]), for post-silicon debug.

The hierarchical design methodology is widely used in designing VLSI circuits. Larger circuits are divided into functionally independent modules that contain less components and thus have lower complexity. And depending on the complexity of these modules that rule can be applied recursively until functionally atomic modules are obtained.

These functional modules are like sub-circuits and they are exhaustively tested during unit validation. However the same does not happen for bigger functional modules created by their junction due to the increasing complexity. Therefore, the iteration between those atomic modules are critical and can be used to isolate an error. The error can be a consequence of a wrong behavior of the unit where the assertion is embedded, but it is more likely to be a consequence of a misleading communication or even an unexpected request.

Moreover, hierarchical information can be retrieved from each assertion statement. That allowed us to choose some areas inside a circuit that should be monitored in order to explain a certain failure. Hence, this second approach uses the assertion locality and the design hierarchy to establish which signals should be monitored.

Therefore, the goal of this second technique is to allow the user to focus on some

specific area of the circuit that is more likely to carry evidence of an assertion failure. Considering both aspects, signal selection focuses on the unit where the assertion is embedded, as well as its closest neighbors. The algorithm created can be divided into three main steps: definition of the visible modules, definition of the hierarchical cone of influence and signal selection.

The first step is to select which modules should be monitored considering an embedded assertion. The goal is to preserve the signals in the maximum distance of n levels. For example, for a distance equal to 2 levels, a module is considered visible if it fits in one of the following groups defined by their position relative to the one where the assertion was embedded (M_A):

1. Modules instantiated inside the M_A.

2. Modules instantiated in the same place where the M_A is.

3. Module in which the M_A is instantiated.

4. Module where the last one is instantiated.

The second step is to establish which signals will be considered for signal selection. Besides registers, signals in the boundary of the visible units are taken into account. The same logic used to define the COI of a property can be used to define the hierarchical cone of influence (hCOI). Instead of considering the entire set of signals inside the circuit, S, the hCOI is determined over the set of signals inside the visible area S_h. I.e., the hCOI is a set of signals defined using the same rules of the COI definition, plus one restriction:

- There must be a path between the property and the signal that only goes through the visible area.

Once the set of signals inside the cone of influence until is defined, the signal selection algorithm is performed over it.

One limitation of both approaches is the area consumption to route all possibly monitored data into the trace buffer, as already explained in section 4.4.2. Nowadays, integrated circuits can have hundreds or even thousands of embedded assertions and the approach presented here would not be feasible. In order to overcome this issue, we also propose an algorithm to join the assertions in clusters.

5.2.4 Clustering Assertions

Although the COI is a subset specific for each assertion, two or more assertions can have similar COIs. For the method of COI based signal selection, these assertions can be grouped without any loss of information or with an acceptable loss.

Assertion clusters are then united by the same principle presented by Neishaburi and Zilic [2010], i.e., the closeness of two assertions is determined by the intersection of their cone of influence (COI). The more common elements their COIs have, the bigger the chance is for them to be joined in one cluster.

For the assertion clustering, two different algorithms were implemented. The first one is just like the one presented by Neishaburi and Zilic [2010], where the number of cluster c is previously set. In this algorithm, the pairs of properties are sorted by the size of their intersection set. At every iteration, the pair with the largest intersection is chosen to be merged into one cluster.

Although the purpose of the clustering algorithm in the previous work was different, the maximum number of assertions per cluster was also used here. The reason is simple: to avoid one big assertion cluster being created due to one property with a big COI. For example, if a property A_i has a COI equal to the set of signals inside the circuit, $S'_i = S$, then one cluster with the properties with the biggest COIs would all be gathered together.

Because of that limitation, a second algorithm was created to only join those assertions that have a similarity factor of at least σ, where σ is the lower threshold determined by the user. The similarity factor is calculated using the percentage of intersection of the properties compared, and it is the minimum percentage obtained:

$$s_{ij} = \min(p_{ij}, p_{ji}) \tag{5.5}$$

Where the percentage computation is based in the intersection's size between two properties COIs. For properties A_i and A_j, the computation of the percentage is:

$$p_{ij} = \frac{|S_i \cap S_j|}{|S_i|} \tag{5.6}$$

Therefore, the variable p_{ij} indicates the percentage of signals from S_i that are also included in S_j.

The same formula is used for comparing the similarity of one property and a cluster of properties C_k:

$$s_{iC_k} = \min(s_{ij}), \qquad \forall A_j \in C_k \tag{5.7}$$

To gather the assertions in clusters based on their similarity, we propose an interactive and greedy algorithm (Algorithm 1). On each iteration, the algorithm chooses the highest similarity between properties and property clusters already created and check whether it is bigger than the inferior threshold σ.

Algorithm 1 Similarity based assertion clustering

```
Input: List of assertions A, similarity threshold σ
Output: Set of clusters C
C ←{}
repeat
  max_sim ←0
  //Find a pair assertion / cluster with maximum similarity
  for i ←0 toC.size do
    prop ←getPropMaxSimilarity (C[i], A)
    psim ←getSimilarity (C[i], prop)
    if psim > max_sim then
      max_sim ←psim
      save_prop ←prop
      save_cluster ←i
    end if
  end for
  //Add the property to the cluster if similarity ≥ threshold
  if max_sim ≥ σ then
    AddProperty (C, save_cluster, save_prop)
    RemoveProperty (A, save_prop)
  else
    //Find pair of properties with maximum similarity
    for p1 ←0 toA.size - 1 do
      p2 ←getPropMaxSimilarity (p1, A)
      psim ←getSimilarity (p1, p2)
      if psim > max_sim then
        max_sim ←psim
        save_prop1 ←p1
        save_prop2 ←p2
      end if
    end for
    //Create a new cluster if similarity ≥ threshold
    if max_sim ≥ σ then
      JoinProperties (C, save_prop1, save_prop2)
      RemoveProperties (A, save_prop1, save_prop2)
    end if
  end if
until max_sim < σ
return C
```

An example of the algorithm execution for a threshold of 85% is depicted in

Figure 5.4. The algorithm starts by creating a first cluster with the pair of properties with the highest similarity factor (Figure 5.4.1). In the following interaction, it fails to find a property that can be joined into this new cluster (Fig 5.4.2), so it creates a new cluster (Fig 5.4.3). Once all similarities are below the established threshold (Figure 5.4.4), the algorithm halts.

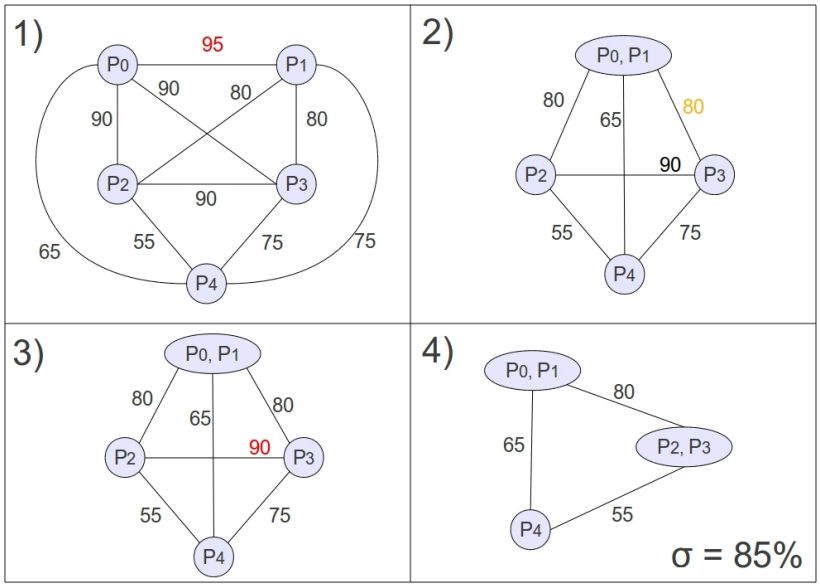

Figure 5.4. Example of the algorithm for clustering assertions based on similarity.

On the one hand, the first algorithm focus on a limited hardware overhead, where only n groups of signals can potentially be captured. On the other hand the main concern of the second algorithm is to group only assertions with similar COIs, hence, optimizing the quality of data captured.

Once the assertion clusters are created, the selection algorithm will run based on a set of signals S'_{C_k}, where:

$$S'_{C_k} = S'_i \cup S'_j, \qquad \forall A_i, A_j \in C_k \tag{5.8}$$

Any compensation mechanism can be adopted to prioritize the signals that are common to more assertions, although it might not be necessary if the cones of influence are alike. Because of that, we did not include any compensation in our algorithm for signal selection when using it with clusters.

Using clusters, the algorithm is performed in the same way, but instead of finding one group of signals for each assertion, it chooses one group for each cluster. The

set of signals considered in the signal selection algorithm is the union of all cones of influence of all assertions inside the given cluster. In the end, the quantity of signal groups established will be equal to the number of clusters, $|C|$.

5.3 Automatic Connection

The instantiation and connection of the design for debug system to the target circuit is done using a wrapper, which is a module automatically generated that instantiates both the CUD and the DfD module, plus a signal selector module. That approach is cleaner and avoids unnecessary changes in CUD code, and furthermore, it has no effect in the design synthesis. From a file hierarchy point of view, the system architecture is represented in Figure 5.5.

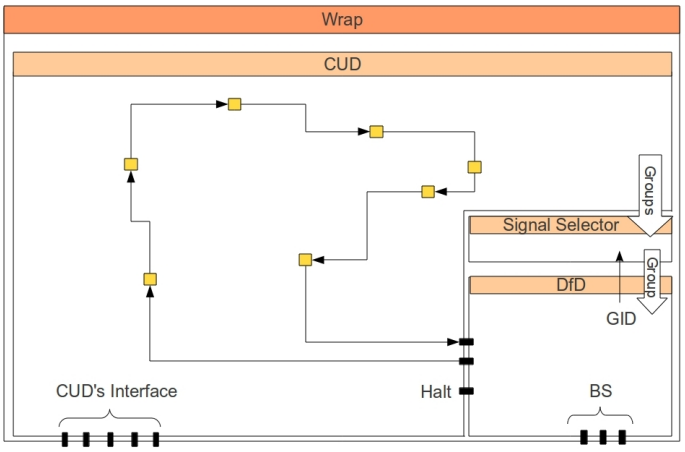

Figure 5.5. RTL instantiation using a wrap module. Signal selection is done through an intermediate module and the wrap's interface is composed by the CUD's old interface plus the boundary scan.

In order to instantiate the circuit under debug, the user must define its interface for fastest clock, reset and halt, in addition to any instantiation parameter that differs from its default value. The reset is optional, since some circuits may require a more complex reset sequence and cannot be restarted automatically.

On the other hand the instantiation of the DfD module and its connection is done automatically using metrics extracted from the circuit RTL and files generated by the signal selection and cluster algorithms. The only parameter the user should configure in this stage is the trace buffer's depth.

A third module is instantiated in the wrap file, the signal selector, and it implements the selection of the signal group that will be monitored. This module is

controlled by the DfD unit and its purpose is also code readability and flexibility, since it is also generated automatically during this stage.

One important goal of the technique implemented is to minimize the required modifications in the original circuit RTL. Thus, assuming that the assertions were already embedded due to post-silicon testing, the only changes required are:

- Adding a connection to the monitored signals to module interface
- Connecting the assertions in a chain and add control signals to scan their values

Once this software is run, new RTL files are generated and ready for synthesis.

5.4 Data Retrieval and Interpretation

The data extracted from the trace buffer can be written to some file and a simple software was created to interpret it. The capturing samples are translated to "lighthouses", which are properties that act as guides in a directed state space search (Yalagandula et al. [2000]).

The first step for lighthouse creation is to map the data to the captured signals' state. Each bit of each sample corresponds to the state assumed by one signal in the moment it was captured. For example, a trace signal s_i is mapped to the variable v_i if the bit i of the sample is equal to 1. If that same bit had the value 0, the variable would be negated ($\neg v_i$). After all bits of a sample have been translated, the lighthouse is generated by the conjunction of all mapped variables.

As the result of this interpretation, a new file is created with the lighthouse expressions. For each sample of signals one lighthouse is created. Moreover, the lighthouses are also in a temporal sequence equal to the sequence of events captured.

In addition to the lighthouses, the interpretation software extracts the following informations from data read:

Error Occurrence: The first piece of information that must be read is whether there was an error or not during the last execution.

Failing Assertion: If there was an error, the script identifies which assertion has failed.

Trace Usage: The length of data captured can differ according to the execution length and can also be determined from the data read. This information is only kept in the trace buffer when the DfD module was executed in Capture state.

In addition to the data extracted, the software also requires some information generated in pre-silicon stage by the connection software. The list of signals inside each group and the cluster definition must be provided.

Chapter 6

Results

In order to evaluate the technique proposed and assess its viability, three different tests were developed. In this chapter, we present the result of each one. Moreover, we also present the analysis of the results found.

As a proof-of-concept of the entire system described in Chapters 3, 4 and 5, a study case employing the debug system was performed. This first test was used to evaluate the entire debug flow, starting with the DfD system integration until the data extraction and interpretation.

A second test was developed to assess the signal selection method employing the concept of cone of influence together with the clustering algorithm, both presented in Chapter 5.

The last test focused on the impact caused by the insertion of the design for debug logic and all the connections necessary for its use. The area and performance overhead were evaluated using different system configuration.

6.1 Case Study

In this thesis, we propose a novel technique for post-silicon debug. Consequently, the first test was performed to assess its correct functioning and also its feasibility of choosing the correct group of signals according to the failure detected.

Although this study case was developed with a small design, conceptually it shows how this architecture can be employed to debug more complex designs. It illustrates the debug flow described in section 3.1 which includes the automatic connection of the DfD system, error detection and identification on-chip, data extraction and trace recovery.

6.1.1 Target Design

The circuit to be debugged is a simplified version of the MIPS32 processor implemented in Verilog. This design was chosen due to its simplicity and vast documentation available (MIPS Technologies, Inc [2001]).

The implemented version includes one memory and two caches (instruction and data cache), a module of traps, forwarding mechanism and pipeline interlock provided by hardware. This version has a five depth pipeline:

IF: Instruction fetch, the instruction indicated by the program counter is read into the instruction register.

ID: Instruction decode, the fetched instruction is decoded, processor flags are set.

EX: Execution, arithmetic logic unit (ALU) stage.

MEM: Memory, data can be read or written in memory.

WB: Write-back, the result can be written in a register.

A total of 38 assertions were embedded in the processor to check whether the design was behaving correctly. The assertions were all described in OVL and their checkers had the same structure described in section 4.2. The processor modules were modified so it was possible to add assertion wires through their interfaces. Figure 6.1 shows the processor's module hierarchy and the order in which the assertions were routed.

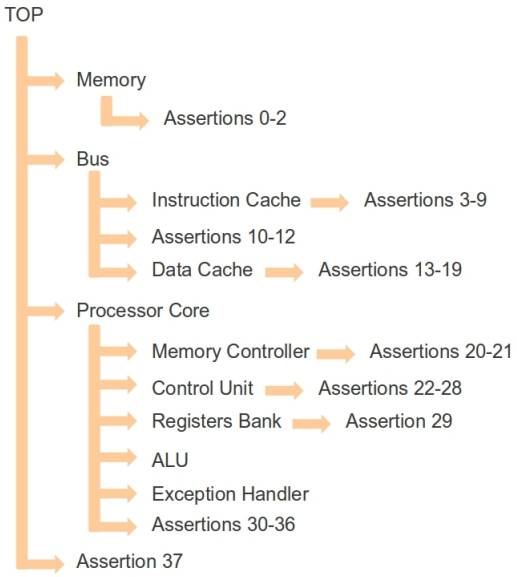

Figure 6.1. Processor modules hierarchy.

6.1.2 DfD System Integration

The proposed debug flow starts during the stage of RTL description in the pre-silicon development phase. Before tape-out, the design for debug module must be integrated to the circuit to be debugged.

Therefore, given the RTL description of the circuit to be debugged and the properties specifications, the first step was to use the connection software to integrate the DfD module.

As an initial procedure, the connection software extracted relevant information from the design RTL, such as:

Internal Signals: All candidates to be monitored were extracted to an auxiliary file, the candidates include module inputs and registers.

Cone of Influence: For each property in the design, a file was generated with the configuration of its cone of influence. Besides the candidates that compose the COI, these files contained each path that connects the subsequent pair of signals.

The second procedure was the assertion clustering and signal selection. The algorithm chosen for clustering in this test was the one with a predefined number of clusters, which was set to 4 clusters. The assertions were distributed as shown in Table 6.1 and the signal selection followed that distribution. For each cluster, a group of 32 signals was defined using the union of the COI of all properties in it.

Cluster ID	Assertions ID
0	7, 8, 9, 10, 12, 13, 14, 15, 16, 29, 30, 33
1	0, 1, 2, 3, 4, 5, 6, 11, 17, 25, 26, 35
2	18, 19, 20, 21, 22, 23, 24, 27, 28, 31, 32, 34
3	36, 37

Table 6.1. Assertion distribution into 4 clusters.

Finally, the connection software generated the RTL code of the debug system integrated to the circuit to be debugged. For the group selection, a module implementing a multiplexed selection was automatically generated, and for the mapping between assertion `id` to the cluster `id`, a second module was generated and inserted inside the DfD control unit.

6.1.3 System Operation

The new RTL generated by the connection software was ready to be synthesized without any modification. It contained the description of the debugging system fully integrated, composed of the DfD module, the embedded assertions, the MIPS32 based processor augmented with the debug architecture, plus the communication interface between the DfD module and the debug architecture.

The debugging system was then synthesized using Xilinx® ISE Design Suite and simulated using the Xilinx® iSim. In order to simulate the system, a verilog test fixture was implemented to simulate an external stimulus necessary for system initialization and also for the DfD module configuration. Besides that, the test fixture was capable of resetting the CUD to rerun in case an error had been detected. Test vectors were uploaded to CUD memory before its execution and represented MIPS instructions of different compiled softwares.

Once the vectors were uploaded, the simulation was started by reseting the CUD and configuring the DfD module to the monitor state. The DfD module configuration was done through a trace buffer reserved position, which was written by the external controller (Figure 6.2). The circuit was then put into its regular execution and it was only interrupted if any failure was detected; otherwise, it would stop when a halt instruction was read.

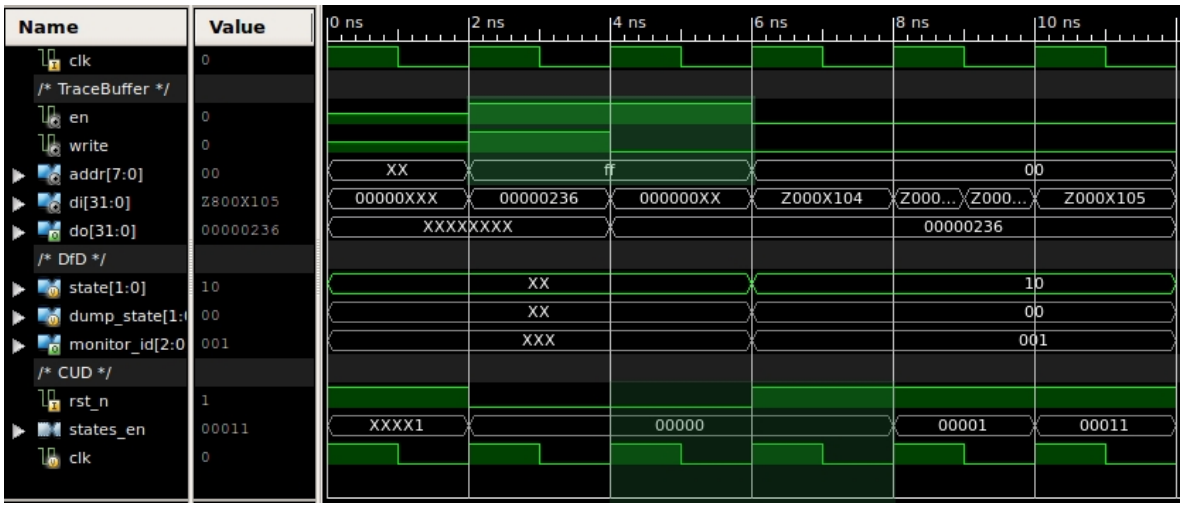

Figure 6.2. During the configuration transaction, the CUD was reset while the DfD module was configured via a trace buffer. After the configuration data had been written to the buffer, the DfD module used it to set its internal state signals.

In the case of a detected failure, the DfD control unit immediately stalled the processor execution (Figure 6.3) and started to scan-out the values stored in the as-

sertions. The assertion processor was able to correctly identify the assertion that had failed by incrementing an internal counter until the error reached its interface (Figure 6.4).

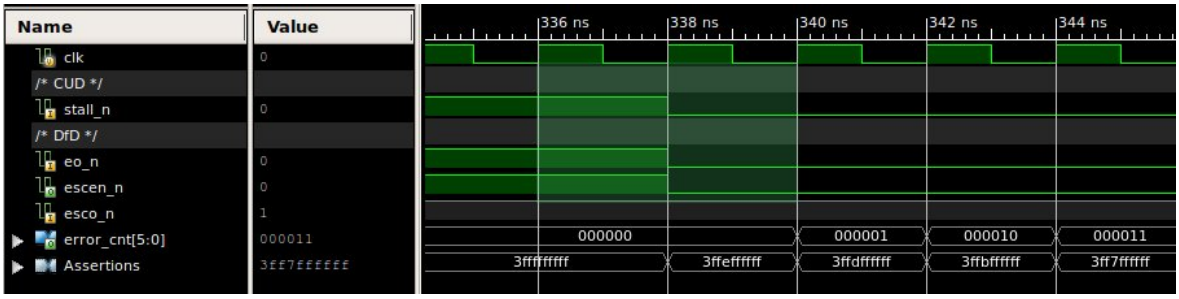

Figure 6.3. Once an error had been detected, the CUD was stalled and the error started to be scanned.

Finally, the error information was written in the trace buffer and the system was halted by the DfD control unit.

After the system had been halted, the data was extracted from the trace buffer to check whether the execution finished normally or an error was detected (See appendix A.2 for more details). If the CUD was halted due to a failure, the system was reset

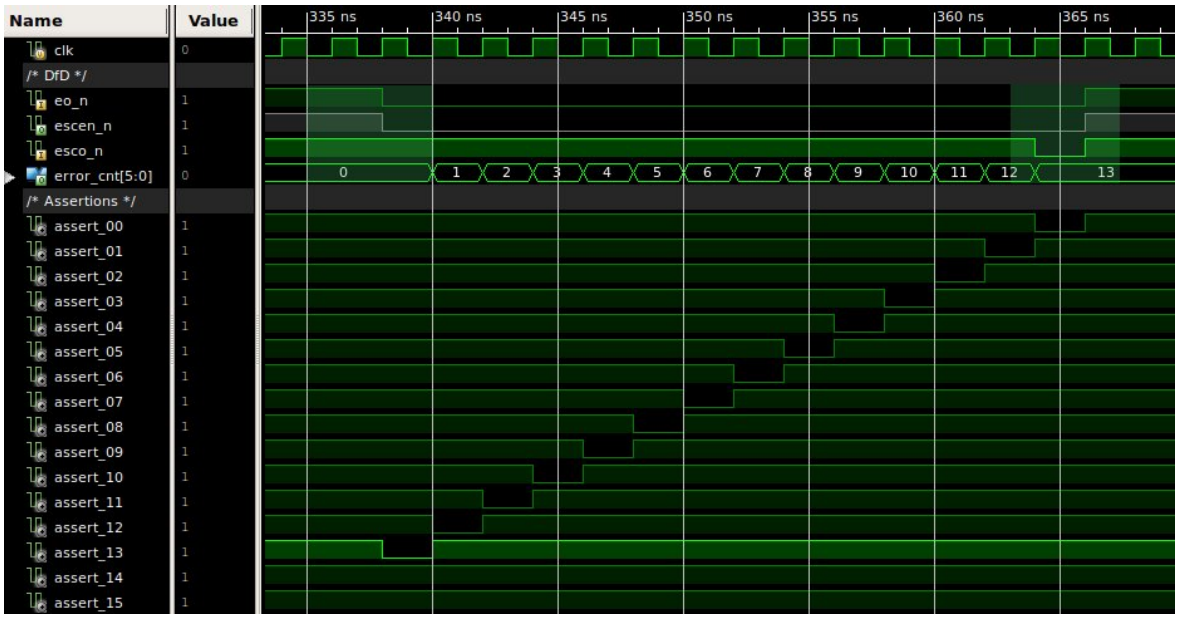

Figure 6.4. The error identification was done through the assertion scan-chain. Once scanning had been enabled, the values of the assertions were transmitted to the assertion processor. It took 13 cycles to reach the processor corresponding to the `id` of the failing assertion.

and the same test was re-executed. The DfD module didn't require any external configuration - it automatically changed to capturing state and started monitoring the correct signal group (Figure 6.5).

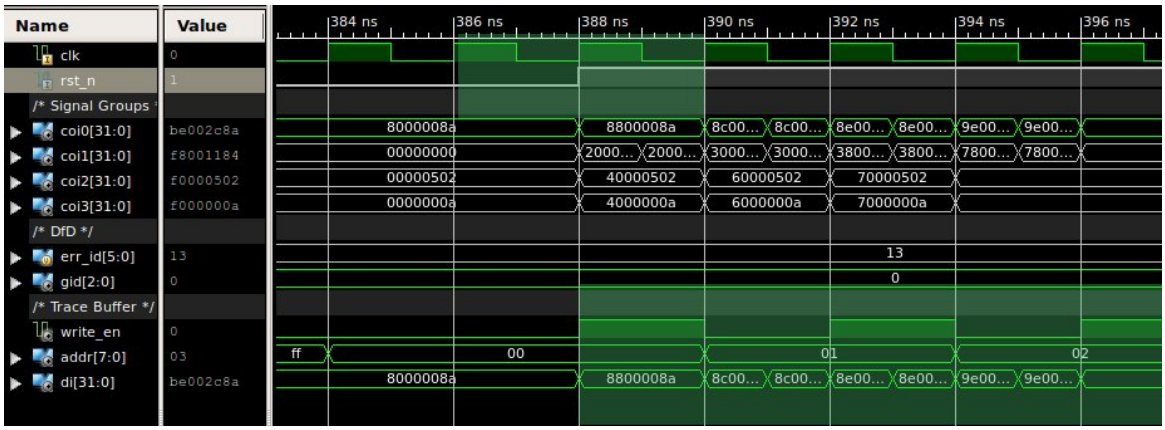

Figure 6.5. This trace exemplifies error capturing after the CUD has been reset. The group captured was group 0 due to a failure in assertion 13. The interval used was of 1 cycle.

During the second execution, the group of internal signals was captured according to the failure detected until the error was again detected. After the second execution, the data was extracted from the trace buffer into an external file for further analysis.

Of the six test vectors created, only one triggered an assertion failure inside the circuit under debug, which was promptly detected and identified. In the other five executions, the DfD system had no interference in the circuit execution, which ended as expected.

6.1.3.1 System Emulation

The same test vectors were used in system emulation using a Field-Programmable Gate Array (FPGA), a Virtex 5 model. The purpose of the emulation was to validate the simulated behavior, because its execution environment is more like the silicon circuit and it is more accessible.

The communication to the debug core was done using a larger hardware system design controlled by a MicroBlaze processor. This new system replaced the test fixture and it was responsible for CUD initialization, DfD module configuration and data extraction. The communication system was developed by Abner Marciano and I and it will likely be addressed in his undergrad thesis. A summary of the emulation architecture can be visualized in Appendix C.

As expected, in the FPGA environment, the debug system behavior was similar to its simulated behavior. The DfD system efficiently detected and identified an assertion failure. Moreover, it captured the values assumed by the group of signals chosen according to the debug failure. In the end of the debug session, those values were transfered to an external computer and saved to a raw file.

6.1.4 Data Extraction and Interpretation

In order to identify the root cause of the bug, the data extracted from the chip during debug session was used to restore a possible failing scenario.

First, the trace buffer was dumped and the data extracted was saved to a file in an external computer. Then, the analysis software presented in section 5.4 was used to retrieve only the relevant data from this file. Each data sample was encoded into a boolean statement using verilog syntax and used as lighthouses.

After that, the method of guided formal verification was used to recreate a valid sequence of events that led the chip to the assertion failure, starting in the reset state and passing through the lighthouses.

In the end, the system was useful to extract relevant data from the circuit execution and we were able to successfully rebuild a possible failure scenario using a formal verification tool.

6.1.5 Third-Party Software

In order to test the proposed system and also to simulate the entire developing and debug process, two third-party software were used.

For hardware synthesis and simulation, the Xilinx® ISE Design Suite Xilinx [2011a] was used targeting the FPGA Virtex 5 (XUPV5-LX110T). And to restore the failure scenario, the formal tool Jaspergold® Automation [2011] was used to search for a possible scenario between the lighthouses created and the failure detected.

6.2 Accuracy of Signal Selection

To measure the accuracy of signal selection COI based with and without assertion clustering, we implemented an algorithm similar to the one presented by Gao et al. [2009]. The choice was due to its simplicity and independence of data analysis tool.

We used three different designs for this part. Besides the MIPS design used in the previous test, an opensparc processor and a bus controller with 4 distinct access

were analyzed.

The first part of this test consisted of the analysis of the clustering algorithm and similarity between the COIs of each property in a design. The result of the clustering algorithm pointed out an interesting aspect of the selected designs. All properties in the design had similar COI. Despite the checkers logic, almost all registers of the design were inside the cone of influence of each property. For the MIPS design, all the assertions were included in a single cluster for similarities up to 99%.

Table 6.2 shows the the maximum similarity between the properties of each design tested, i.e., the maximum similarity factor that can't distinguish properties between more that one cluster.

Design	**#Assertions**	**Max Similarity**
MIPS 32	38	99%
OpenSparc	66	97%
Bus Controller	64	98%

Table 6.2. Maximum similarity found for each design.

All designs tested had the same aspect that led all cones of influence to be similar, probably, due to the number of interconnections existing in the design, cross dependencies and feedback mechanisms. However, we had access to just a few designs, so we cannot conclude that this is a generic characteristic for all modern designs, although there seems to be a tendency for that. A new and wider study should be done in order to correctly analyze this aspect and try to identify which kinds of designs are suitable for COI based signal selection.

In the next part of this test, we tested the clustering algorithm using the hCOI based signal selection. The same designs used in the previous analysis were analyzed using this second approach. The hierarchical cone of influence was no longer equal for all properties and the signal selection algorithm was capable of selecting distinguished sets of signals. Here we will present the result for the MIPS32 based processor.

The clustering algorithm based on the similarity of the hierarchical bounded COI (hCOI) was tested for similarity of 80%, 90% and 95%. The result of each test can be seen in Table 6.3.

On the other hand the clustering algorithm presented by Neishaburi and Zilic [2010] was tested for 4, 8 and 12 clusters, with a maximum number of 5, 8 and 10 assertions per cluster. The result of each test can be seen in Table 6.4.

Tables 6.5 and 6.6 show the comparison between the set of signals chosen individually for each assertion and its cluster signal selection for each algorithm implemented. Both tables are for 32 bits selection.

Minimum Similarity (%)	Number of Clusters	Size of Each Cluster
95	14	9, 9, 4, 3, 3, 2, 8 x 1
90	13	10, 9, 4, 3, 3, 2, 7 x 1
80	11	11, 9, 4, 3, 3, 3, 5 x 1

Table 6.3. Clusters created by the algorithm based in the similarity of the hCOI for the MIPS design.

	Size of Each Cluster per Boundary		
Max Number of Clusters	5	8	10
12	7 x 5, 3	3 x 8, 4, 2	3 x 10, 6, 2
8	6 x 5, 2 x 4	3 x 8, 4, 2	3x 10, 6, 2
4	-	-	3 x 10, 8

Table 6.4. Clusters created by the algorithm presented by Neishaburi and Zilic [2010] restricted to the hCOI for the MIPS design.

	Cluster Configuration (# clusters x max size)			
Intersection Signals(%)	8 x 5	8 x 8	8 x 10	4 x 10
90-100	12	4	5	6
80-90	14	12	18	12
70-80	6	15	6	9
60-70	1	4	2	3
up to 60	5	3	7	8

Table 6.5. Distribution of signal selection intersection comparing hCOI based signal selection and clustering based for different c assertion clusters.

The distribution of this intersection can also be visualized in Figure 6.6.

Comparing the algorithms implemented, the similarity algorithm was more sensitive and produced signal selections closer to the assertion based selection. However, it produced more clusters, and half of the clusters created had only one assertion. On the other hand, the intersection algorithm generated fewer clusters, but it showed a tendency of gathering assertions with hCOI not so similar and resulted in a signal

	Minimum Similarity		
Intersection Signals(%)	95%	90%	80%
95-100	16	15	11
90-95	11	11	6
80-90	10	11	16
70-80	1	1	5

Table 6.6. Distribution of signal selection intersection comparing hCOI based signal selection and clustering based, grouped by their similarity.

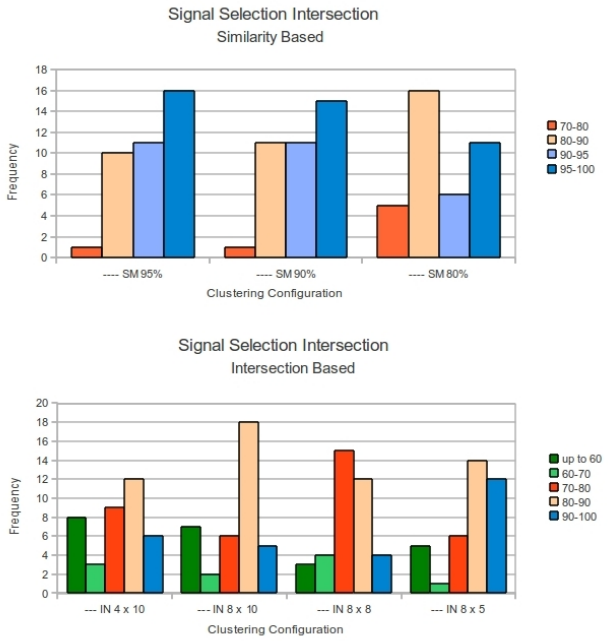

Figure 6.6. Graphs with the distribution of signal selection intersection comparing signal selection hCOI based and clustering based for both clustering algorithms.

selection more different when compared to the assertion based selection.

Therefore, the similarity algorithm proved to be more sensitive and produce a better signal selection, however, it may create too many clusters. In cases where the area overhead is critical, a modified version of the algorithm could apply an upper limit of clusters, instead of using a minimum similarity threshold.

6.3 Area and Performance Overhead

The MIPS32 based processor was again used to evaluate the area and performance overhead of the insertion of the design for debug logic with a trace buffer of size 256 x 32 bits, depth and width respectively. In this case, the method for signal selection was a multiplexer.

Since the synthesis was done targeting an FPGA, the area overhead was measured in the number of registers and lookup tables (LUTs).

Table 6.7 represents the area overhead due to the inclusion of the assertion checkers and their connection, which was 144 registers plus 266 lookup tables. Since there were 38 embedded assertions, the overhead was 38 registers to form the scan chain and 104 registers required to check the property behavior, in cases where the property

represented a temporal expression. The overhead of LUTs was due to the routing of the scan chain and the combinational logic of the checker.

	Unit Measured	
Components	# Slice Registers	# Slice LUTs
Processor	1789	3147
Processor + Assertions	1933 (+8.0%)	3413 (+8.5%)

Table 6.7. Area overhead due to the inclusion of the assertion checkers.

The area overhead of the components of the on-chip debug system is shown in Table 6.8. The impact of the DfD module insertion is analyzed separately from the connection of all signals to the signal selector module.

	Unit Measured	
Components	# Slice Registers	# Slice LUTs
Processor + Assertions	1933	3413
+ DfD Module	1976 (+2.2%)	3454 (+1.2%)
+ Signal Connection		
- Assertion Based	1976 (+2.2%)	3720 (+9.0%)
- Cluster Based	1976 (+2.2%)	3693 (+8.2%)

Table 6.8. Area overhead due to the addition of each system component, i.e., the analysis of one row is relative to the preceding one. The percentages shown in the columns are relative to the area overhead calculated using the CUD + Assertions.

The overhead caused by the DfD module was only 2.2% of registers, most of it due to the two counters inside the trace manager and assertion processor, plus 1.2% of LUTs for all the control and interpretation logic.

The signal routing was the more expensive structure in order of occupied LUTs. Monitoring one specific group of signals for each property had a bigger impact when compared to the clustering approach. In spite of the difference for routing 4 groups instead of 38, the final difference was only 0.8%, since the clustering approach required a translation mechanism that also consumed lookup tables.

The area overhead found for the entire system was significant and should be improved. In particular, the overhead due to signal routing and assertion insertion should be reduced.

On the other hand, the overhead caused by the DfD logic was small and it was shown to be a great alternative in cases where the assertions were already embedded for validation purpose. It provides internal control and visibility of the assertion that failed and allows signal selection sensitive to the failure. It also can be configured by the user to monitor any other group of signals.

There was no significant impact over the performance of the circuit.

Chapter 7

Conclusion

Post-silicon debug has two main limitations, lack of visibility and low controllability. Because of that, it requires a lot of effort from a validation engineer to find bugs and identify their root cause. Some solutions had been proposed in the literature to provide more visibility during this stage; however, all the feasible approaches presented a small and limited enhancement or required the circuit to be stopped.

Considering the limited space, we presented a novel technique that concentrates the visibility enhancement in some areas of the circuit. As far as we know, this is the first solution that employs the concept of isolating an area in the circuit to be monitored. Moreover, we implemented an automatic method to choose which area should be monitored according to an identified assertion failure during real time executions.

Therefore, the main goals of this thesis were to introduce this new solution for post-silicon debug, as well as to identify the positive and negative aspects of this idea and to determine which ones require improvements so this technique can be applied in the industry.

7.1 Contributions

This section outlines the main contributions of this thesis to be applied during post silicon debug.

7.1.1 Embedded Assertions

The architecture presented in the present work employs assertion checkers to improve error detection as well as to enhance the quality of the signals monitored. The design for debug system developed is capable of detecting and identifying any failure during

real time execution. It also can be used to disable the assertions to reduce the power consumption.

The assertion checkers used allow the error to be promptly identified during a CUD execution, even when there is no evidence of the error in the chip outputs. Thus, the circuit can be halted right after a failure detection, and since the assertion checkers are usually triggered in an early stage of error propagation, the distance between the root cause of the bug and the last signal capture is diminished.

7.1.2 Signal Group Selection

For internal visibility enhancement, trace-buffer architecture has become one main approach utilized by debug engineers. This architecture is capable of saving the values assumed by a limited group of signals during CUD execution for a late recovery. On the one hand, this approach can store signal values without any interference on execution flow. But on the other, the number of signals that can be monitored is very low and it is limited by the trace width.

To overcome this limited access, we proposed an optimization of signal selection algorithms sensitive to assertion failure analysis. The root cause of a failure can be inside any place of the circuit that belongs to the cone of influence and this can be a good boundary to search for the it. There's no need to monitor signals outside the COI, since they don't carry any evidence of the error.

However, depending on the system, the cone of influence of each assertion may contain almost all logic of a circuit. For those cases, we also proposed an approach based on the hierarchy of the circuit. Thus, the restriction of the area to be monitored is done according to the module hierarchy and the position of the assertion. Although this approach doesn't guarantee that the monitored area will contain the logic that caused the failure, it will likely carry evidence of the bug and it may be used to isolate the error.

7.1.3 Clustering Algorithms

Although selecting one group of signals for each potential assertion failure may seem an intuitive approach that guarantees the maximum coverage for all of them, it may be considered a naive approach since hundreds or even thousands of assertions may be embedded in one circuit. Because of that, two algorithms were presented to gather these assertions in clusters, thus reducing the number of groups that can potentially be observed.

The first algorithm prioritizes the control of the number of clusters to be generated and it was based on a existing algorithm for the same purpose. However, this approach seemed vulnerable, since a property with a large cone of influence would cause all the assertions with bigger COIs to be gathered in the same cluster.

A second algorithm was then proposed to consider the similarity between the set of signals that compose the cone of influence instead of only ordering the size of their intersection.

7.1.4 System Integration

Two software were developed to help system integration (i.e., the insertion of the debug architecture and its connections along with data interpretation). The connection software developed allows the user to easily configure the attributes of the design for debug module to be embedded. Moreover, it allows the user to choose how the signal groups and clustering should be configured.

One important aspect of this software is to allow the user to set different configurations and to compare their impact.

7.1.5 Employment of the Debug Architecture

The debug system implemented was able to capture a specific signal group according to a detected failure. This was an automatic process that required minimal configuration before the circuit execution. It was able to detect and identify the failure, and in addition, it controlled the DfD module reconfiguration and signal capturing. After the debug execution, all data captured plus information about the error identified was stored in the trace buffer for later recovery.

Through the application of the architecture proposed in a study case, we could evaluate its impact and conclude that it is a feasible technique, although it still requires some enhancements. Considering that the embedded assertions can be embedded for validation purpose, the area overhead for the DfD module insertion and signal connection was only 43 registers and 307 LUTs, which is reasonable considering its benefit. The overhead of the trace buffer will depend only on the storage space determined during the synthesis.

Additionally, there was no significant performance overhead due to the addition of the debug circuitry.

7.2 Limitations and Future Work

The purpose of the presented work was to introduce one novel approach capable of increasing the visibility of relevant signals for assertions embedded in circuits under debug. Although this architecture looks quite promising, more investigations should be done to optimize its use in the field.

7.2.1 Signal Set Selection Based on Hierarchically Bounded COI

More research is necessary to improve the hCOI based signal selection - specially to refine the definition of hierarchical instances interesting to the analysis of any assertion. The proposed approach is a heuristic that selects signals from instances hierarchically close to the property checker. However, its efficiency still relies on validation engineer judgment to refine the result and determine which levels should be considered. The limit of module hierarchy that should be used can depend on the unit test and how the chip was developed. A rule of thumb is to include information that comes from modules developed separately, since they probably haven't been exhaustively tested together.

7.2.2 Non-determinism and Controllability

A second improvement can be investigated in order to support debugging of non-deterministic errors. The system proposed requires a second execution of the same test that triggered the assertion so it can capture the correct group of signals. Because of that, it requires good controllability of the error, which means it must be easily reproducible. Although the proposed system is suitable to debug most of the functional bugs discovered, non-deterministic failures are still harder to debug in this case.

7.2.3 Area Overhead

The overhead of area required to connect the group of signals to the signal selector should also be enhanced to minimize the impact of this approach. The use of a bus should be investigated for that purpose.

7.2.4 Similarity Between Cones of Influence

Furthermore, one good aspect that should be investigated is the cone of influence similarity, in order to identify which types of designs have properties with similar cones of influence and whether this is a common characteristic in circuits nowadays. Designs with segregated logic that result in distinct cones of influence would probably benefit greatly from the proposed debug mechanism.

7.2.5 Capturing Configuration

Another factor that can be better explored is the lighthouse construction. Although it has already been implemented in the current system, it was not on the scope of this first work to analyze the best approach of capturing configuration or to establish a method to determine the interval to be applied.

It is known that different configurations allow trace restoration to be performed using different algorithms. And the less frequent the snapshots are, the bigger the time coverage will be, although the harder it may get to restore a valid failure scenario. Nevertheless, more studies are required to find which aspects should be considered and how they influence that decision.

Bibliography

Abramovici, M., Bradley, P., Dwarakanath, K., Levin, P., Memmi, G., and Miller, D. (2006). A reconfigurable design-for-debug infrastructure for socs. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 7--12, New York, NY, USA. ACM Press.

Accellera Organization, I. (2004). Property specification language reference manual. www.eda.org/vfv/docs/PSL-v1.1.pdf.

Acellera Organization, I. (2004). Systemverilog 3.1a language reference manual. http://www.eda.org/sv/.

Acellera Organization, I. (2011). Open verification library. http://www.accellera.org/activities/ovl/.

Automation, J. D. (2011). Jaspergold and jaspercore: Advanced formal property verification. Available at: `http://www.jasper-da.com/products/jaspergold.htm`.

Beizer, B. (1995). The pentium bug - an industry watershed. In *Testing Techniques Newsletter, TTN Online Edition*.

Boule, M. and Zilic, Z. (2007). Efficient automata-based assertion-checker synthesis of seres for hardware emulation. In *ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific design automation*, pages 324--329, Washington, DC, USA. IEEE Computer Society.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press, Cambridge, MA.

Constantinides, K., Mutlu, O., and Austin, T. (2008). Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 282--293, Washington, DC, USA. IEEE Computer Society.

De Paula, F. M., Gort, M., Hu, A. J., Wilton, S. J. E., and Yang, J. (2008). Backspace: formal analysis for post-silicon debug. In *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, pages 1--10, Piscataway, NJ, USA. IEEE Press.

Gao, J., Han, Y., and Li, X. (2009). A new post-silicon debug approach based on suspect window. In *VTS '09: Proceedings of the 2009 27th IEEE VLSI Test Symposium*, pages 85--90, Washington, DC, USA. IEEE Computer Society.

Geuzebroek, J. and Vermeulen, B. (2008). Integration of hardware assertions in systems-on-chip. In *Test Conference, 2008. ITC 2008. IEEE International*, pages 1--10.

Gort, M. (2009). Practical considerations for post-silicon debug using backspace. Master's thesis, University of British Columbia.

Ho, C. R., Theobald, M., Batson, B., Grossman, J., Wang, S. C., Gagliardo, J., Deneroff, M. M., Dror, R. O., and Shaw, D. E. (2009). Post-silicon debug using formal verification waypoints. In *Proceedings of the Design and Verification Conference and Exhibition (DVCon '09)*, pages 85--90.

Hsu, Y.-C., Tsai, F., Jong, W., and Chang, Y.-T. (2006). Visibility enhancement for silicon debug. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 13--18, New York, NY, USA. ACM.

IBM (2003). *FoCs: Formal Checkers - a Productivity Tool.* IBM Research Lab in Haifa.

Keshava, J., Hakim, N., and Prudvi, C. (2010). Post-silicon validation challenges: how eda and academia can help. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 3--7, New York, NY, USA. ACM.

Ko, H. F. and Nicolici, N. (2009). Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(2):285--297.

Liu, X. and Xu, Q. (2009). Trace signal selection for visibility enhancement in post-silicon validation. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09*, pages 1338--1343.

MIPS Technologies, Inc (2001). *MIPS32 Architecture For Programmers Volume I: Introduction to the MIPS32 Architecture.*

Nacif, J. A. M., de Paula, F. M., Foster, H., Jr., C. J. N. C., and Fernandes, A. O. (2003). The chip is ready. am i done? on-chip verification using assertion processors. In *12th International Conference on Very Large Scale Interation - System-on-Chip (IFIP VLSI-SoC)*.

Neishaburi, M. H. and Zilic, Z. (2010). Enabling efficient post-silicon debug by clustering of hardware-assertions. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 985--988.

Perry, D. L. and Foster, H. D. (2005). *Applied Formal Verification: For Digital Circuit Design*. McGraw-Hill, New York, USA, 1st edition.

Prabhakar, S. (2009). Algorithms and low cost architectures for trace buffer-based silicon debug. Master's thesis, Virginia Polytechnic Institute and State University.

Rabaey, J. M., Chandrakasan, A. P., and Nikolic, B. (2003). *Digital integrated circuits : a design perspective*. Prentice Hall electronics and VLSI series. Pearson Education, 2 edition.

Shojaei, H. and Davoodi, A. (2010). Trace signal selection to enhance timing and logic visibility in post-silicon validation. In *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, pages 168--172.

van Rootselaar, G. J. and Vermeulen, B. (1999). Silicon debug: Scan chains alone are not enough. *Test Conference, International*, 0:892.

Vermeulen, B. and Goel, S. K. (2002). Design for debug: Catching design errors in digital chips. *IEEE Des. Test*, 19(3):37--45.

Wong, Y. (1985). Hierarchical circuit verification. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, DAC '85, pages 695--701, New York, NY, USA. ACM.

Xilinx (2011a). Embedded development kit edk 12.2. Available at: `http://www.xilinx.com/products/design-tools/ise-design-suite`.

Xilinx (2011b). Logicore ip processor local bus (plb) v4.6 (v1.05a). Available at: `http://www.xilinx.com/support/documentation/ip_documentation`.

Xilinx (2011c). Microblaze processor reference guide (embedded development kit edk 13.1). Available at: `http://www.xilinx.com/support/documentation/sw_manuals`.

Yalagandula, P., Singhal, V., and Aziz, A. (2000). Automatic lighthouse generation for directed state space search. In *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pages 237 –242.

Appendix A

DfD Module Interface

In this appendix, the definition of the DfD module interface and also a description of memory communication will be detailed. Although bus width and error ID width are implementation-specific, the following tables show a 32-bit data bus and six-bit ID fields. The trace buffer used had a depth of 1024 positions.

A.1 Interface Description

Table A.1 lists the global DfD module signals.

Signal	Type	Description
CLK	Input	Global clock signal. All signals are sampled on the rising edge of the global clock.
RST_N	Input	Global reset signal. This signal is active LOW.

Table A.1. Global signals

In the table A.2, there are more detailed information about the signals that can be used to access the trace buffer.

Signal	Type	Description
TB_WE	Input	Write enable.
TB_RE	Input	Read enable.
TB_ADDR[9:0]	Input	Address to be written in or read from.
TB_DI[31:0]	Input	Data to be written in the trace buffer.
TB_DO[31:0]	Output	Data read from trace buffer.

Table A.2. Interface used to access the trace buffer.

Furthermore, two signals compose the functional boundary which allow the communication with any external controller (Table A.3).

Signal	Type	Description
FBS_VALID	Input	Indicates when configuration data has been written in the trace buffer. When high, it triggers DfD module configuration.
FBS_DONE	Output	This signal indicates when there are valid data in the trace buffer and it is ready to be read.

Table A.3. Functional Boundary Interface.

All signals described so far have external access. The following interfaces are internal to the circuit and are used for data retrieval and execution control.

Table A.4 shows the interface with the circuit under debug.

Signal	Type	Description
CUD_DATA[31:0]	Input	Sample data. Each sample data bus must have the same width as the trace buffer.
STALL	Output	Flag that indicates whether the CUD execution must be paused

Table A.4. Interface with circuit under debug

Table (A.5) describes the signals used in the control and access to the assertion checkers.

Signal	Type	Description
AP_EO_N	Input	Error input. This signal is active low.
AP_ESCO_N	Input	Error scan input. This signal is active low.
AP_ESCEN_N	Output	Error scan enable. This signal is active low.

Table A.5. Description of signals of the interface with the assertion chain.

A.2 Memory Communication

The communication adopted between the end user and the design for debug is established through a shared memory (the trace buffer itself) and signals `fbs_vconf` and `fbs_valid`. It should be done only when the CUD and DfD module are stalled.

The user should only write in the last position of the shared memory with data to be used for DfD system configuration. This data will be interpreted by the DfD Controller as follows:

```
DATA_WD              WIDTH+3         3      2       0
  |       interval      |      id       |id_vd| state |
```

Where the `state` field should be set to 00 to disable the DfD module, 10 to the monitor state and 11 to the capture state. The `id_vd` indicates whether monitored assertion should be set to `id`. Otherwise, the DfD module will monitor the last assertion that was triggered. And the `interval` variable configures the distance between each snapshot. Note that the width of `id` will depend on the number n of assertions embedded in the design, i.e.:

$$WIDTH = \lceil \log_2 n \rceil$$

On the other hand, the user can read the entire cache where most of the data will be raw snapshots. Since the buffer is used as a circular buffer, there is no guarantee that a snapshot with address i happened before one with address $i + k$, for $k > 0$. There is also no guarantee that all positions in fact store valid snapshots. That is, if trace depth times interval is greater than the number of execution cycles, some trace positions will not be used.

Because of that, two cache positions are reserved for dumping execution detail data, which are the two positions before the last. In other words, in a trace of depth d, the position $d-1$, $d-2$ and $d-3$ are reserved for DfD module configuration, error detail data and trace occupancy.

The error detail data has the ID of the triggered assertion, while the trace occupancy has the write pointer and a bit that indicates if trace-buffer is full or not.

```
DATA_WD          ADDR_WD+1                1      0
  |       xxxxx       |      write_ptr     | full |
```

Note that, if `full` is set to 1 the `write_ptr` indicates the position with the older snapshot; if set to 0, it indicates the first invalid data and the older snapshot will be the first position of the buffer.

Appendix B

Replaceable Modules Interface

Like appendix A, the interfaces described in the following sections are implementation-specific. They depend on the number of cluster and the number of assertions. For simplification purpose, we will consider the configuration relative to 4 clusters and 16 assertions, in addition to the same width of 32 bits for the trace buffer.

B.1 Signal Selector

The signal selector is originally implemented by a multiplexer, but it can be replaced by different logics. The interface is similar to any MUX and it is composed of:

Signal	Type	Description
`GID[1:0]`	Input	ID of the group of signals that shall be selected.
`COI_0[31:0]`	Input	Bus composed of the signals chosen to be monitored in case the group id is 0.
`COI_1[31:0]`	Input	Bus composed of the signals chosen to be monitored in case the group id is 1.
`COI_2[31:0]`	Input	Bus composed of the signals chosen to be monitored in case the group id is 2.
`COI_3[31:0]`	Input	Bus composed of the signals chosen to be monitored in case the group id is 3.
`OUT[31:0]`	Output	Group of signals to be captured.

Table B.1. Description of signals of the signal selector interface.

B.2 Assertion - Signal Group Mapping

The signal mapping represents a function to convert the error id into the group id. It can be implemented as a ROM, a multiplexer or an arithmetic function. Independently, the interface is very simple, as can be seen in table B.2.

Signal	Type	Description
`ERROR[3:0]`	Input	ID of the assertion that has failed.
`GID[1:0]`	Output	ID of the group of signals that should be monitored.

Table B.2. Description of signals of the signal selector interface.

Appendix C

System Emulation

In order to emulate the debugging system in the FPGA, we incorporated the same design used during emulation tests (Section 6.1) into a larger hardware system.

This system included a Xilinx MicroBlaze Xilinx [2011c] softcore processor responsible for providing circuit stimulus and communicating with the DfD system via shared access to the trace buffer. The MicroBlaze was connected to a Processor Local Bus (PLB) Xilinx [2011b] as a master and communicated to its slaves. A slave PLB interface converter was created to interface with the debugging system.

Additionally, a UART module provided by Xilinx was also attached to the PLB as a slave. The UART was used to connect the FPGA system to an external computer through a serial port. The entire architecture can be visualized in Figure C.1

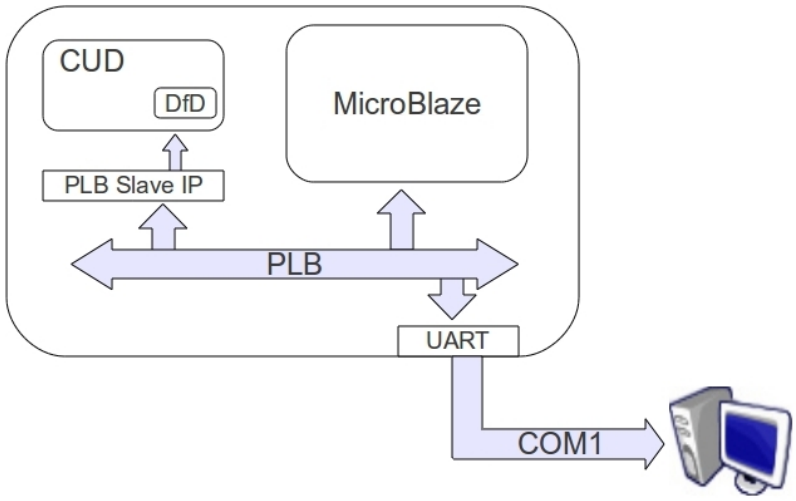

Figure C.1. Architecture developed for the debugging system emulation in an FPGA.

The communication was set to trigger an interruption when an execution has finished, the interruption being dispatched by the APM PLB Slave module. Then, the softcore started to dump all data stored inside the trace buffer. All data read was transmitted to an external computer, which stored it in a raw file.