A Methodological Framework for Automation of Hardware Software Co-Validation of an IP

Abhinav Jain

ECE

IIIT-Delhi

A thesis submitted for the degree of

Master of Technology

2015 August

1. Reviewer: Name

2. Reviewer:

Day of the defense:

Signature from head committee:

Abstract

The VLSI industry is witnessing rapid changes driven by the growing market requirements. In order to maintain their competitive quotient, reducing the time to market of their IC products has become a prime concern of the semiconductor companies. The reason is the direct impact of the time to market on the profit margins of a company. The product development cycle includes formation of the product specifications, design implementation and design verification. The process of verifying a new product design consumes a significant time portion of its development cycle. The verifying process starts after the product specifications are formed, continuing through the various later phases. One of the phases is after the first silicon prototype of the product is fabricated, and is known as the post silicon validation (PSV). Although considered as a bottleneck in the product development cycle, it is an absolute requirement being the final step towards mass manufacturing of the product. While standardized automation tools exists for other phases like verification, the PSV phase has remained void of such tools due to the varying properties of IC products. Traditional PSV methods have failed to keep pace with the VLSI advancements. Creative strategies are required by the semiconductor industry to reduce the product cycle time and address the new requirements of the PSV process. This work presents one such approach which allows introduction of automation in PSV to reduce the time to market of the IC products along with other requirements which are at-speed testing, hardware and software co-validation and reduction in the resource investment in PSV. Using this approach, the average validation time of a PSV for NAND and an SPI NOR flash controller IPs are reduced by 76% and 61% respectively compared to the times required, if the automation framework is not utilized.

Acknowledgements

I express my gratitude to all my teachers, colleagues and friends who helped me during the course my work. Firstly I give a special thanks to Dr. Alexander Fell of IIIT Delhi who consistently reviewed my work and guided me through to its completion. I thank Mr. Amit Goel of ST Microelectronics who motivated me for this work and showed me the big picture related to it, Mr. Anoop Kumar of ST Microelectronics to help me understand the conventional validation scheme and sharing his experience and Mr. Manish Agrawal of ST Microelectronics for his valuable inputs.

Contents

Glossary **v**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

2 Literature Review - Opportunities in Post Silicon Validation **5**

3 Motivation **9**

4 Post Silicon Validation **11**

5 Proposed Methodology **15**

6 Automation Framework **17**

6.1 Linux 17

6.2 Sysfs 18

6.3 Expect 19

6.4 Architecture of framework 19

6.4.1 Support for validation in the device driver 21

6.4.2 Test database 21

6.4.3 Test program 22

6.4.4 Log monitor program 23

6.4.5 Set up controlling program 23

6.5 Work flow 23

6.6 Example 24

6.6.1 Sample Linux device driver 24
6.6.2 Support in device driver 25
6.6.3 Sample test case 26
6.6.4 Sample log monitor program 28
6.6.5 Sample set-up controlling program 29

7 Results **31**

7.1 Conventional procedure 31
7.2 NAND controller 31
7.3 SPI NOR controller 32

8 Conclusion **39**

References **41**

Glossary

BCH Bose Choudhary Hocquenghem; A coding scheme for error detection and correction

CPU Central Processing Unit

DFD Design For Debug; A VLSI design paradigm to assist debugging the integrated chip

DFT Design For Test; A VLSI design paradigm to assist testing the integrated chip

DMA Direct Memory Access; A feature for allowing the peripherals to access the main memory without the help of processor

ECC Error Correcting Codes; Codes for detecting and correcting errors introduced in data during transfer

ELA Embedded Logic Analyzer; A structure for assisting on-chip debug

IC Integrated Chip; An electronic circuit fabricated on silicon

IP Intellectual Property

JTAG Joint Test Action Group; An industry standard for testing by boundary scan

NAND A logic gate; NAND flash is the memory using NAND structure for bit storage

NOR A logic gate; NOR flash is the memory using NOR structure for bit storage

PC Personal Computer

PSV Post Silicon Validation; A phase in semiconductor product cycle

RTL Register Transfer Level; Hardware synthesizable code in a hardware description language

SoC System on Chip; A complete electronic system fabricated on a single silicon die

SPI Serial Peripheral Interface; A serial communication protocol

UART Universal Asynchronous Receiver and Transmitter; A serial communication protocol

VLSI Very Large Scale Integration; A process of designing electronic systems by combining millions of transistors on a single silicon die

GLOSSARY

List of Figures

1.1 Architecture of modern SoC 2

4.1 Set-up for validation 12

6.1 Architecture and work flow of the automation framework 20

6.2 Sample Linux device driver 25

6.3 Sample show function for attribute id 26

6.4 Sample test case 27

6.5 Output of the test case from test program 28

6.6 Sample log monitor program 28

6.7 Sample set-up controlling program 29

7.1 Time for validation with a single NAND flash normalized with respect to time taken for the same using the automation framework 33

7.2 Time for validation with a single SPI NOR flash normalized with respect to time taken for single NAND flash validation using the automation framework by engineer 4 33

7.3 Timeline for validation with multiple flashes using the same set-up . . . 36

7.4 Validation times of different number of flashes using the automation framework normalized with respect to time taken for single NAND flash validation using the automation framework by engineer 4 37

List of Tables

7.1 Description of engineers performing validation 32
7.2 Description of the validation set-ups 34
7.3 Validation times of NAND controller IP on different set-ups using the automation framework normalized with respect to time taken for single NAND flash validation using the automation framework by engineer 4 . 35
7.4 Validation times of SPI NOR controller IP on different set-ups using the automation framework normalized with respect to time taken for single NAND flash validation using the automation framework by engineer 4 . 35

LIST OF TABLES

Chapter 1

Introduction

Rapid advancements in the field of very large scale integration (VLSI) technology has pushed the technology node into deep sub-micron figures allowing the industry to observe a shift in the trend from system on board to system on chip (SoC) in which all the components of a system that were earlier on a big printed circuit board, have now been pushed to within the boundaries of a single integrated circuit (IC). The die size on the other hand has remained the same. Consequently more logic devices are accommodated in the same die area allowing most of the required functionality to be present on the chip itself. This has resulted in high performance systems at the cost of an increase in complexity. Modern paradigm for the semiconductor product design is a central processing unit (CPU) core based SoC in which additional functionality is provided by intellectual property (IP) cores connected to the CPU core (figure 1.1). In every new generation of the SoC, more functionalities are added. In other words, it has a higher number of IPs compared to the previous one, increasing its complexity. The time required to confirm that the product behavior is in accordance with the specifications and/or datasheets also increases with complexity.

The activity of verifying the design is a crucial phase in the product development cycle. It starts as soon as the design specifications are completed and runs in parallel with the design implementation activity. The verification phase evaluates the correctness of the design implementation to meet the specifications. Simulation checks and formal methods are commonly used for verification focused on uncovering the register transfer level (RTL) design errors for quality control. Verification does not cater to all the functional bugs as simulation is very slow and formal verification methods face

scalability problems (1). Hence another evaluation process, known as validation, is employed in the product development cycle.

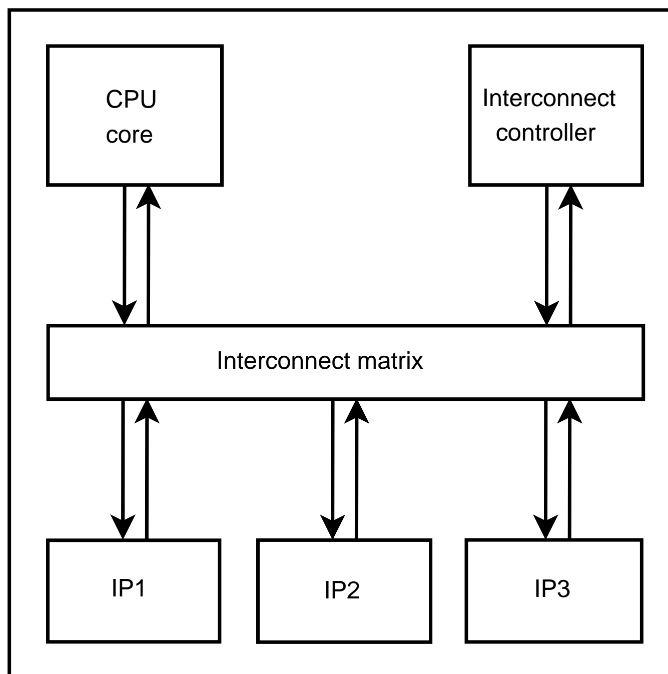

Figure 1.1: Architecture of modern SoC

Validation is a quality assurance phase which begins when the design is implemented, and it tests the design to meet the desired operational requirements. The validation process aims to establish evidence that the design is fit for the intended purpose. It involves the activities of test planning, test execution and response analysis. In test planning, the functionalities to be tested and the stimuli for testing them, are determined. In test execution, the test stimuli are applied to the device under validation. The responses obtained by the application of tests, are analyzed for pass/fail results. Modern SoCs contain multiple IPs with some of them working with external devices. Each IP is individually validated and the ones working with external devices are validated multiple times against different external devices manufactured by different vendors. For instance an IP of a flash memory controller works with memory devices that may be externally connected and would be validated multiple times with different flash memory devices.

Validation has two phases to prove the correctness of all the behaviors of different

components in an SoC (2). The first phase, known as pre-silicon validation, starts much before the SoC tape out using its emulation platform and software models of associated hardware to run simulations. The stimulus is applied by test benches written in C, System Verilog or other supporting languages. The response is in terms of waveforms which are analyzed for functional bugs. This phase is slow but has very high observability and controllability. The second phase is post silicon validation (PSV). It is scheduled after the tape out of the SoC on actual hardware and therefore is capable of catching bugs arising due to lack of maturity of the manufacturing process in addition to the functional ones (3). This phase is much faster but has less observability and controllability compared to the pre-silicon phase. The application of stimuli and capturing of responses is traditionally achieved using hardware debuggers.

PSV is the final phase in the semiconductor product development cycle. Its importance has grown with the increasing complexity of the SoCs as more bugs tend to escape the pre-silicon validation phase (4) and the product cannot be released into the market before its successful validation. The disciplines included in PSV are functional validation, electrical validation and compatibility validation. Testing the behavior of various parts of the SoC in terms of hardware responses when a stimulus is applied, is functional validation. Electrical validation is ensuring correct electrical levels and involves measurement of various quantities using oscilloscopes, logic analyzers and other equipment. An SoC does not work as an independent entity rather as an integrated system in which the hardware interacts with software running on it to work as required. In order to employ the SoC in multiple domains, it is made programmable. Different functionalities are obtained by modifying the embedded software programmed in it. The complexity of SoCs has increased not only in terms of hardware, but also in terms of software that runs on it. As a result validation is not only limited to hardware, but extends into the embedded software part as well.

Compatibility validation is the discipline in which the externally connected hardware and software like operating systems and device drivers for various IPs, are validated for the new silicon (5, 6, 7). Therefore, PSV can detect logical design bugs, fabrication defects, loss of signal integrity, mismatches in thermal and power levels as well as software bugs. With the rising SoC complexity, the resources required for its validation as a complete system has also increased making it a bottleneck in the product

cycle (3). The increasing investment in validation and shortening time to market windows have forced the industry to look into new ways for managing the resources more efficiently because conventionally, hardware and software validations are carried out separately by different teams at different stages of the product development cycle (8).

Chapter 2

Literature Review - Opportunities in Post Silicon Validation

PSV involves significant capital and time investment due to the unavailability of global standards and procedures for PSV, limited access to the internal circuitry and lack of automation tools. Efforts are made to establish standard coverage metric for PSV, enhance the observability of an SoC and reduce the resource investment in PSV.

Kai Cong et al. (9) have used a virtual prototype or simulator of an SoC to evaluate the coverage of the PSV tests. It is the software model of a hardware which allows the unmodified execution of the programs, written for that hardware. The software model is enhanced to capture the execution data whenever a program runs on it. The execution data show which code sections of the software model are executed allowing the computation of the traditional code coverage metric like statement, branch, function and block coverage of a program. Two new hardware specific coverages, register coverage and transaction coverage, are defined which are also computed using the execution data. The register coverage metric is the number of times a register is accessed in a test and the transaction coverage metric is the number of times different transactions or events take place in the virtual device during the test. In PSV, the same tests are executed on the hardware prototype and the device state is recorded at different events. The recorded data from the test execution on the virtual device and the actual device is then compared to discover inconsistencies.

A similar approach is discussed by Mehdi Karimibiuki et al. in (4). They have used the field programmable gate array (FPGA) prototype of an SoC to determine the RTL code coverage of the validation tests instead of the software code coverage of virtual prototypes in (9). Modifications to the RTL code of the SoC capture the execution of the different sections of the RTL code in order to compute the statement and branch coverage when the validation tests run on the FPGA prototype. These approaches try to establish coverage metric for the PSV tests and allow good quality tests to be ready before the arrival of the silicon prototype.

To enhance the observability and controllability during PSV, debug assisting structures must be added to the SoC in the design phase. Embedded logic analyzer (ELA), discussed by Ho Fai Ko et al. in (10), is such a design for debug (DFD) structure consisting of embedded memories and trigger units. The ELAs are distributed across the SoC to tap important on-chip signals. The trigger unit monitors a set of signals to detect a pre-defined event which is followed by sampling of the tapped signals and then storing them in the embedded memories or trace buffers. The tapping is achieved by using the testing wrappers around the concerned blocks. Miron Abramovici in (11) proposes the addition of reconfigurable structures and supporting logic structures into the SoC for in-system silicon validation. The reconfigurable structures namely programmable trigger engine and reconfigurable logic engine, are interfaced to the standard joint test action group (JTAG) port through a primary controller allowing its reconfiguration according to the validation requirements. A tracer captures the tapped system signals which are selected by a signal probe network. This system can be configured for signal tracing and analyzing the logic, obtaining scan dumps and for performing on-chip functional testing of a block. When the trace buffer overflows, its content is offloaded over the JTAG port. The amount of trace data is limited by the width and depth of the trace buffer which are kept small to minimize the area overhead.

As the amount of the trace buffer is limited, less number of signals and their states can be stored. To increase the observability of the circuit, the non-traced values must be computed from the traced signals and the input test vector using restoration algorithms. Selection of the signals in such cases play a significant role as the restoring capability of some set of signals may be higher than the others. The possibility of a bug capture depends on the extent of signal restoration. In (12), partial restoration calculations are performed for various signals and based on them the signals to be traced are

selected. In (13), Kanad Basu and Prabhat Mishra have presented a method for signal selection based on total restorability. The signal paths are classified into dependent and independent paths. Then, using the probabilities of the different input signals to affect the output signal on different paths, their edge values are computed. An edge is a signal path containing combinational circuit elements between two flip flops. The edge value is a measure of how one end of the path controls the other end. Using these edge values, flip flop values are computed to filter out the flip flop signals to be traced for complete restoration of non-traced values.

In the attempt to reduce the time invested in validation, different concepts have been consistently proposed. In (6), a validation system for faster validation of microprocessors has been presented by Ilya Wagner and Valeria Bertacco. Microprocessors are validated by running constrained randomized sequences of instructions on the hardware and in simulation. The results from the tests on hardware are compared with those of simulations for pass/fail decisions. Obtaining the results from simulation is a time consuming process as it is very slow in comparison to the actual hardware. The validation time is reduced using the proposed scheme as the simulation stage, required to get the reference outputs of various sequences of random instructions, is bypassed. This is due to the fact that reversible programs are generated by this system enforcing the same state of the processor after the test execution as compared to the state before testing. Hence the final state is already known at the start of the test. By studying the instruction set of a microprocessor, the inverses of various arithmetic, logical, branching, memory, floating instructions are computed and a library is created. After a randomized sequence is generated, the inverse of each instruction from the library is appended to the sequence in the reverse order. Now, the complete sequence has instructions as well as their inverse code present within the same sequence in an order intended to bring the processor back to the initial state after the entire sequence finished execution. Thus the simulation stage is not needed as the final state is known beforehand and if a bug is present, the final state will not match the initial state.

In (5), the authors analyze the effectiveness in reducing the complexity of the validation of a core based chip having pre-validated IPs. In such scenarios the integration of the components and the communication architecture is fault prone. A systematic approach in which all the interfaces are validated first rather than the entire SoC, is proposed to reduce overall validation time. All data transfers, or communications, among

different SoC components, are routed through the communication architecture. Common integration problems for an SoC are identified and classified into three categories based on the parts of the SoC involved. These are component to communication architecture, component to component and communication architecture problems. This identification and classification leads to a logical sequence for interface validation. First the components to communication architecture interface is verified by high level test benches. Then the component to component interfacing is validated by enforcing communication among the different components in pairs. Finally the entire communication architecture is validated using all the components simultaneously. Validating the interfaces in that sequence followed by a full system simulation will reduce the overall time consumption as interface validation require small, directed test benches in simulation.

In (14) functional self testing is advocated by Krstic et al. to achieve at-speed testing of high speed circuits not possible with external testers. At-speed testing requires execution of test cases at the maximum clock frequency on which the circuit is designed to work. First the processor is self tested by a test program which applies functional patterns for testing. Then the processor is reused to test other components of the system. The processor executes functional tests by generating patterns and analyzing the response. The structural tests of stuck-at fault testing, path tests of delay testing, maximum aggressor tests of bus testing and test patterns for component testing are mapped to sequences of instructions which are executed by the processor core in order to perform the required tests. The use of instructions eliminates the need of scan chains and hence at-speed testing is made possible.

Chapter 3

Motivation

The growing demand of the electronic market for versatile products and time to market pressure have made the VLSI industry adopt a new electronic design paradigm which is a processor core based SoC. In such a design, one or more CPUs are the central entity and the required capabilities according to the design specifications are provided by multiple IPs interfaced with the core. Above it, the SoC is made programmable so that it is employable in multiple application domains. The hardware is tightly coupled to the embedded software to implement a particular functionality. Software has thus become an integral part of the SoC making it an integrated system rather than a standalone entity. Therefore the software must be validated along with the hardware. The increasing complexity of the hardware and the embedded software has made the validation process a complex time consuming task. Large investments, in terms of both time and capital, are required to verify the design in order to prevent severe bugs to reach the mass manufacturing phase.

From a commercial perspective, PSV faces the major challenges of growing resource investments, aggressive time to market schedules and at-speed testing of hardware. The efforts required in validation of an SoC having more IPs is naturally higher than in the one having less. So with every new generation of the SoC this effort tends to increase which forces the industry to employ more resources into validation, which increases the cost of validation, in order to remain consistent with the product timeline. At-speed testing of modern SoCs is becoming a necessity with operating frequencies reaching the gigahertz range as non at-speed testing may allow clock related bugs to escape. At the same time, at-speed testing is a difficult task at hand (14), as the components

present inside the SoC to assist testing, operate at a lower frequency compared to the surrounding SoC.

The new core based SoC designs allow the reuse of available IP cores which reduces the design time significantly (5). Reuse of IPs also alleviates the validation efforts to a great extent as they are pre-validated and the test plan largely remains same. Even though the reused IPs are expected to work, as they are functional in the previous generations of the SoC, their validation on the silicon prototype of every new design is necessary to ensure that the silicon prototype, including the reused IP, is free from any hardware bugs and ready for mass production. The presence of hardware bugs is a possibility due to changes in the manufacturing technology or in the design of other parts of the SoC. Hence except for the creation of the test plan, the entire validation procedure for both hardware and software of a particular IP is repeated for every new SoC. For IPs that work with an external device, the same procedure needs to be repeated against devices from different vendors. The PSV phase does not yet have the aid of standardized tools and methodologies. Considerable amount of human intervention is therefore inevitable for this phase making it a bottleneck in the product development cycle. Also the tightly coupled hardware software interaction renders standalone hardware and software validation as less relevant (8) creating a need for hardware software co-validation. The present scenario calls for major changes in the conventional validation methods and innovative approaches to address these challenges in order to regulate the flow of the product development cycle.

This work is motivated to address some of the challenges mentioned above. A methodological approach for validation is proposed which supports at-speed testing, hardware software co-validation and is suitable to introduce automation. All these contribute in reducing the time to market of the SoC.

Chapter 4

Post Silicon Validation

In PSV, the procedure to accomplish a particular test after determining its stimulus depends on the validation engineer aiming to cover all corner cases for the target. The hardware and software functionality is the prime aspect of validation, as malfunctioning of the system proves very costly when already deployed in the field. The functional validation discipline of PSV is performed to evaluate the functional correctness of different parts of the SoC. Ideally the pre-silicon validation phase should uncover all functional bugs, but this has become impossible to achieve with increasingly complex designs. Hence functional coverage is the prominent industrial coverage metric in which a set of functional points which the validation process should address and verify, are identified by the engineer (4) based on the specifications and datasheets. From a commercial view, successfully verifying all practical use cases in an end user environment is the highest concern.

Without loss of generality, an IP core is seen as a set of registers which are accessed by their addresses over a bus. An IP block exports its functionality via these registers (15). The register set includes configuration, command, data and status registers. To exercise a functionality, the associated register bits are modified and tested according to the values mentioned in the datasheet. The test cases are created following this procedure to verify the IP functionality and its integration. The tests are categorized as directed tests and random tests. Directed tests are manually created by a validation engineer to cover all functional coverage points as planned. These can be short tests and long tests. Short tests are aimed at individually validating each and every feature supported by the IP while long tests exercises multiple features and are combinations

of short tests. Randomized tests on the other hand are generated by tools and reach a final state which is unknown. In the absence of a golden response from simulation, it is difficult to analyze the output from such tests. A golden response is the correct expected output obtained by running the same test in a simulation environment, which is not practical due to the slow simulation speed (4). On the contrary, expected outcomes of direct tests are easily inferred by the validation engineer from the documentation. For IPs, directed tests are the most commonly used way for validation.

Specialized validation platforms known as hardware validation kits made for a particular SoC family, are required for functional validation. These have additional probings and connections for enhancing the controllability and observability for functional validation (5, 6). The validation kit is connected to a normal personal computer (PC) to receive the inputs. The debug hardware consists of a JTAG controller which allows access to the internal SoC components. The software used for debugging, runs on the PC attached to the validation kit via JTAG. The PC and the validation kit are also connected to each other over the serial port. In addition, both are connected to the local network. The complete set-up for validation is shown in figure 4.1.

In PSV, the hardware prototype must run under the expected load. Thus an operating system boot has become an important validation test for a modern complex SoC (4, 7). The Linux operating system is the most favorable choice by semiconductor companies for this purpose as it is open source, fully customizable and most probably it will be used in the end product. Linux has a low level software known as device driver which implements the different functionalities of an IP. It is a hardware abstraction

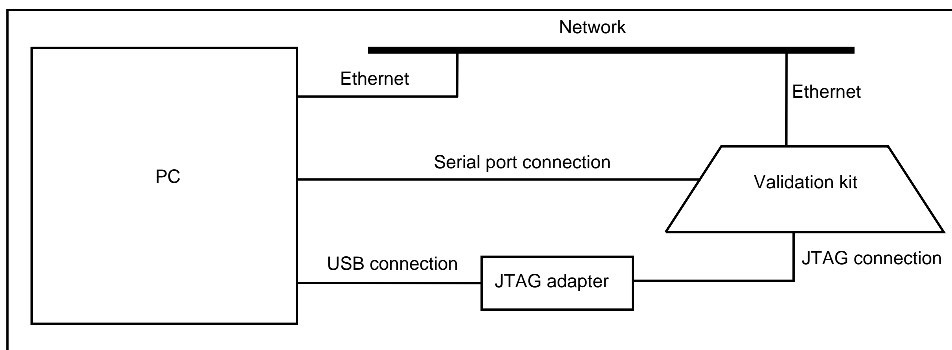

Figure 4.1: Set-up for validation

layer and exports only major functions to the high level software running on Linux for accessing the IP hardware.

Along with the working hardware, a working software platform is also an important part of the deliverables of the SoC product which applications are developed for by the customer. In the past, the software development for the IP cores was initiated after the tape out of the prototype using the actual hardware. Any part of the software which was not aligned with the intended functionality, was corrected during the development and therefore was validated simultaneously. Recently, to reduce the product development time, a virtual prototype of the SoC is used to start the software development in the early phases of product development cycle (9). Therefore the software needs to retested on the actual hardware.

Reuse of IP cores is a common practice to reduce hardware design time. Similarly, the associated software for the IP cores is also reused. Its validation on the new silicon is also mandatory to uncover any issues arising from the addition of software for the new IPs or due to the changes made in other parts of the SoC hardware, and hence conducted in the PSV phase.

Traditionally PSV followed scan and trace based methods. The scan based method reuses the JTAG test access port and the scan chain design for test (DFT) structures. A scan chain is a chain of flip flops connected back to back, allowing the input of data from the first flip flop and output of data from the last flip flop using the JTAG interface. The flip flops are enhanced with a multiplexer to select from two inputs, one is the normal data input of the circuit and the other is the test data input, based on a scan enable input. By asserting the scan enable signal, the test mode is selected in which the scan flip flops select the test vector input and the test vector advances through the scan chain on each clock cycle. After the proper test vector is loaded into the scan chain, it is applied to the circuit by deasserting the scan enable signal followed by a clock cycle. The resulting values of the internal state elements are captured and then offloaded in the same way as loading of the test vector.

In the trace based method, the JTAG interface is reused in conjunction with DFD structures like an ELA which samples the logic state of the selected signals in the circuit under test and stores them in the trace buffer. This trace dump is offloaded over JTAG and then further analyzed for discrepancies.

Writing test cases in a language such as C and then using the compiled binary files for executing various validation tests, is another methodology being followed in the industry for functional validation (2). Using JTAG, the compiled binary files of the test cases are loaded into the memory and then executed by pointing the program counter to its loading address. Many companies adopt mixed approaches in order to speed up the PSV process.

Chapter 5

Proposed Methodology

The widely accepted architecture of an SoC consists of central CPU core/cores and IP cores connected to the CPU via an interconnect matrix. The CPU core runs the software which further controls the functioning of the IPs by communicating with them over the interconnects. It is required to verify the individual functioning of the IP and its integration with the global flow (15) for a complete validation. As suggested in (14), the proposed methodology also uses the software running on the CPU core to apply the test sequences to validate the IPs and their integration with the system. A database of test cases is created by the validation engineer to cover all the planned functional points. A test program runs on the CPU core, applies the test sequences from the test database sequentially, samples the response and records it for future reference. The use of JTAG debugger for test application and sampling response is thus eliminated and all the internal SoC components are accessed using CPU instructions.

An operating system boot is a typical PSV test that is performed on modern complex SoCs (4, 7). Once the Linux boot is successful on the SoC, it is used for validating other IPs. Device drivers are the low level software which directly interact with the hardware and also act as an interface for the high level software to access the hardware. In this methodology, the Linux device driver of the IP under test is reused for performing its validation. The IP is tightly coupled to its device driver which contains modular routines to implement most of the functionality supported by the IP.

According to (2), the general anatomy of a test case is to-

- Bring the SoC into a state suitable for test stimulus application. This includes configuring all the involved IP cores according to the validation test case.

- Apply the stimulus for exercising the functionality to be validated.
- Check and record the post test state of the SoC. This includes reading the hardware registers and memories.

The conventional structure of a Linux device driver contains routines for initialization and other functionalities of the concerned IP. A test case can thus be accomplished by executing single or multiple routines of the driver.

This methodology addresses the major industrial concerns of at-speed hardware testing and reduces the resource investment in the validation phase. The test sequences are applied by the CPU core, hence the tests are performed at the rated speed of the system. As the CPU core and the IP cores are interacting, the communication architecture also gets validated along with the IP at the rated speed.

This strategy views testing as an application running on the operating system and allows application of test sequences for functional validation in a real operational environment. In addition to hardware related functions, the device driver performs many sanity checks, such as address alignment, which are important from software perspective for improving the performance of the system. When tests sequences are created from the device driver, the low level code gets validated implicitly bringing together the aspects of software and hardware validation. This hardware software co-validation paradigm reduces the resources invested in validation as conventionally the validation of hardware and low level software are carried out by different teams at different stages of product development cycle (8).

Further, the approach is suitable for bringing automation in the validation phase by instrumenting the device driver to allow access to the IP from the Linux user space and then automate the execution of tests using a script. If the tests are ready, the validation process requires their application to the IP under test and the response analysis. Using automation, the time spent by a validation engineer to execute the test cases is reduced. The response analysis still requires human intervention, but a considerable amount of time is saved when compared with the time required for validation without using the automation, which is a benefit for any company.

Chapter 6

Automation Framework

Modern SoCs have the capability to run an operating system like Linux which is used as a platform for a framework supporting automated test sequences. The proposed methodology is the core of the automation framework around which a set of programs have been developed to perform different tasks necessary to achieve test execution automation.

6.1 Linux

The Linux operating system has a kernel at its core consisting of various modules and directly interacting with the underlying hardware. In addition, the kernel handles the important tasks of process scheduling, memory management, virtual file system, networking interface and others. The kernel hides the underlying hardware details from the higher level software and runs in a privileged mode in a part of memory in which it has complete access to the hardware. The high level programs do not have direct access to the hardware and depend on the kernel modules for providing necessary support for the same. The device drivers are modules present in the kernel space and allow high level programs to interact with it using a virtual file system. The high level programs are a part of the user space which is a protected part of memory, and use the facilities provided by the virtual file system to communicate with the kernel space.

6.2 Sysfs

Sysfs is a virtual filesystem interface between the Linux kernel space and user space. It is an in-memory filesystem that was developed at the time of kernel 2.4.0 for debugging purposes. Originally called the device driver filesystem or ddfs, it was developed to debug the new driver model that was under development during that time. Ddfs became driverfs when the new driver model was merged in kernel 2.5.1. The development continued and sysfs evolved as a useful interface for other subsystems and provided a common object management mechanism with the inclusion of kernel objects or kobjects. Firstly featured in Linux kernel 2.6, sysfs allows legal exchange of information between the kernel code and user processes.

Sysfs is centered around kernel objects, their attributes and their relationships with each other giving a sense of object oriented paradigm. The sysfs interface can be divided into three major parts. One is the programming interface for exporting kernel space information into user space. Second is the representation in user space for navigation and manipulations and finally a mapping between the user space representations and kernel constructs. A kernel object is mapped to a directory, its attributes to regular files inside the corresponding directory and the relationships occur as symbolic links in the user space making sysfs easily workable with simple shell utilities. Sysfs is conventionally mounted on /sys at the system start up or after using the mount utility.

The sysfs gets populated with directories of the subsystems registered with it which are further filled with subdirectories of the objects registered with the respective subsystem. The major subsystems at the top of the sysfs directory tree include block, bus, class, devices, firmware, module, power and kernel. Each contains individual directories of the kernel objects pertaining to their type. E.g. the block directory contains directories for all block devices while the bus directory contains directories for different buses found in the system further having directories for the devices and drivers attached to that bus. The level of hierarchy can vary for different subsystems depending on the registration scheme of the kernel objects. It is also possible to have multiple mappings of a kernel object in the userspace, if it falls under multiple categories. Symbolic links are created to prevent redundancy of the kernel objects in cases of one to many mappings.

Attributes are the regular files inside the directory structure and are used to exchange information by reading from and writing to them. Each attribute has two

functions namely store and show associated with it in which the writing and reading functionality is implemented respectively. Whenever an attribute is accessed, a page sized buffer is allocated which is passed to either of the two functions depending on the operation being performed. On writing an attribute, the store function associated with it executes and on reading, the show function is executed. In additon to the normal attributes, there are binary attributes which cater to non-ASCII type of data. The usage of these attributes is similar to that of normal attributes. They may be useful in scenarios in which binary data needs to be exchanged between the Linux kernel and user space (16).

6.3 Expect

Expect is tool which allows automatic controlling of interactive programs for example telnet. It starts the program in a pseudo terminal and controls it by supplying the pre-written commands on encountering pre-defined expected strings. A list of expected messages from the program and the response to be sent is provided a priori in an expect script. The messages from the program are stored in a buffer which is searched for a matching string. On finding a match, it sends the corresponding response to the program and terminates, if an expected match is not found. Expect can be used with many programming and scripting languages like C, Bash and Tcl (17).

6.4 Architecture of framework

The set-up described in chapter 4 uses the Ethernet and UART interfaces in addition to the JTAG interface of the SoC. The Ethernet and UART interfaces are used to communicate between the PC and the validation kit. They are assumed to be pre-validated but can also be validated using the proposed framework. In such a case, other interfaces are used to communicate between PC and validation kit. To achieve test automation for this set-up, it is required that the activities on the PC and on the validation kit are performed automatically and in synchronization with each other. The major components of the proposed automation framework include support for validation in the device drivers, a test database, a testing program, a log monitor program and a set-up control program (figure 6.1).

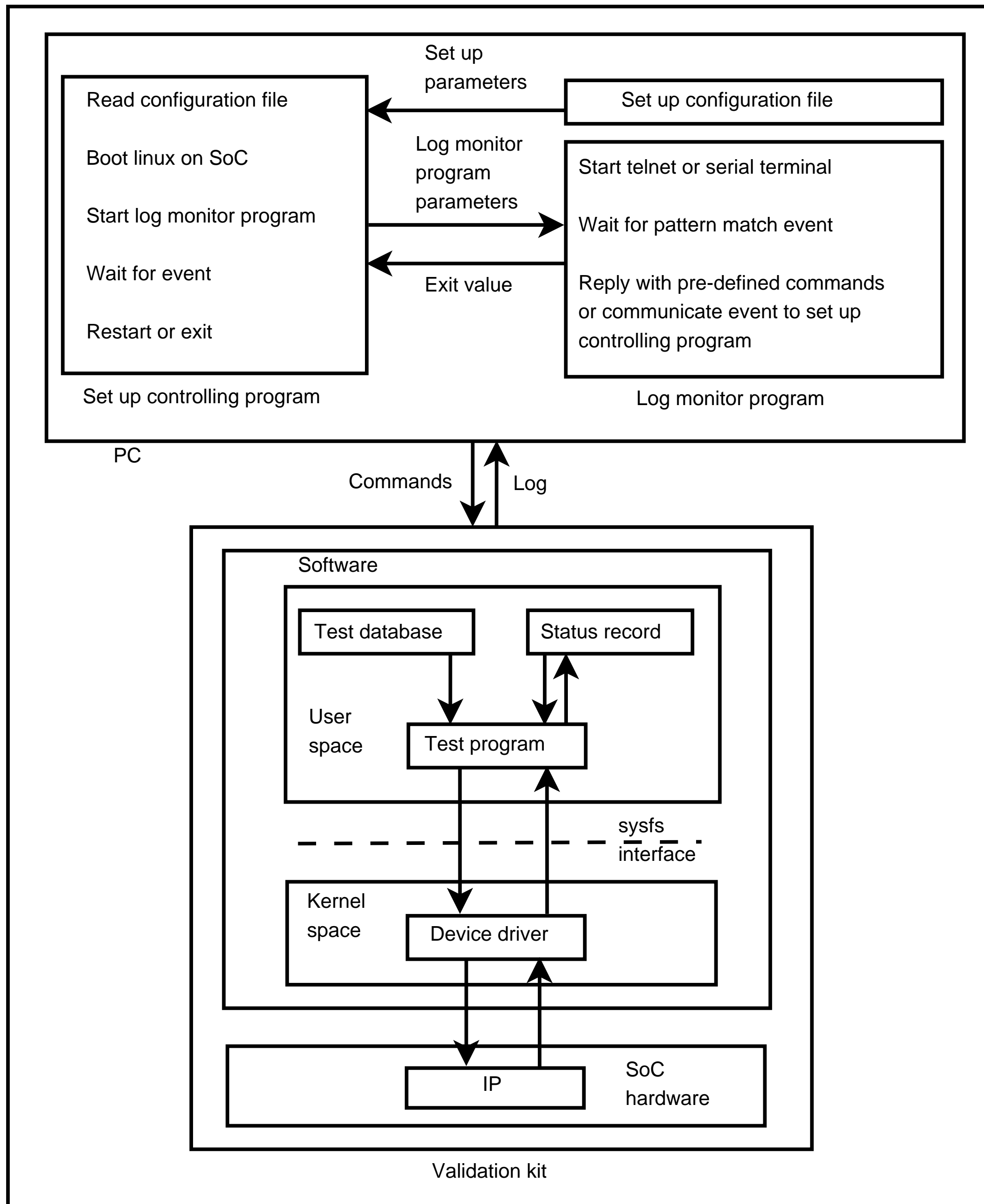

Figure 6.1: Architecture and work flow of the automation framework

6.4.1 Support for validation in the device driver

The proposed automation framework requires support in the device driver of the IP under test for allowing a user space process to execute any of its part. This support for validation is provided by instrumenting the device driver with its sysfs interface which takes test stimulus data from a user space process and returns the result back to it. There are standard kernel functions for creating the sysfs interface and its attributes. Further code for processing the test stimulus data is added to the device driver which allows the execution of an appropriate part of the device driver according to the test sequence. Code for functionalities not yet supported by the device driver may also be added for its validation.

6.4.2 Test database

The test database consists of the test cases and other files, such as a bootloader file required by some tests. Test cases are text files in which each line is composed of a keyword followed by its argument. Most of the keywords generates a part of the stimulus using the argument, required to perform the test. Some of the generic keywords of a test case in this framework are as follows:

- READ_SYS_ATTRIBUTE is the keyword to execute the show function of a sysfs attribute. The argument is the name of the sysfs attribute.
- WRITE_SYS_ATTRIBUTE is to execute the store function of a sysfs attribute. The argument is the combination of the attribute name and the value to be stored separated by @ character.
- READ_REGISTER is to read the value stored in a register. The argument is the hexadecimal value of the address of the register.
- WRITE_REGISTER keyword is to write a value in a register. The argument is a combination of the hexadecimal address value of the register and the value to be written separated by @ character.
- DESCRIPTION is used to describe the test case and its argument is the description. A test case must begin with the description of the test.

- INCLUDE is used to include a test case in another test case. The argument is the name of the test case to be included. It allows writing smaller and modular test cases.

- CONFIGURE replaces the current value of a parameter of the test program during execution time in order to handle situations in which a test case requires different values of some parameters. It is followed by two arguments separated by an @ character. The first argument is the name of the parameter to be modified. The second argument can be the new value of the parameter or DEFAULT for restoring the original value of the parameter.

- APPLICATION is the keyword to run Linux applications. The corresponding shell command is given as an argument to this keyword.

- REBOOT communicates to the programs running on the connected PC, that the SoC is reset followed by a reboot.

6.4.3 Test program

A test program is the program which executes the test plan on the SoC. Firstly it sets up an environment by setting its global parameters such as the location of the test database and base address of the IP. This allows the same test program to be used for different IPs by externally passing the values of its global parameters.

After setting the environment, the test program reads test cases from the test database sequentially, decodes each line and prompts the device driver to apply the corresponding stimulus to the IP. The test program maintains a record of the status of test plan execution before each test is executed in order to allow to restore the last working state, if the set-up goes into an irrecoverable state for example in the event of a reboot. This is necessary to prevent the set-up from going into a loop in which only subset of test cases are executed repeatedly. The program must be able to continue from the last executed test case in order to complete the test plan execution. In addition, the test program sends out synchronization messages to the program running on the controlling PC at specific stages of the test plan execution, for instance before a reboot.

6.4.4 Log monitor program

The log monitor program (figure 6.1) runs on the PC connected to the validation board, and remotely starts the testing program. It is an expect (refer to section 6.3) based program continuously monitoring the data being exchanged between the validation kit and the PC, and searching for specific patterns. The patterns are decoded and the resulting information is then relayed to the set-up controlling program, running on the same PC or a reply is sent to the validation kit. It terminates, if an expected match could not be found for a fixed time interval, a pattern for SoC reboot is matched or an indication for completion of validation is received. The log monitor program is reconfigurable and hence reusable, since parameters are passed to it by the set-up controlling program when it is started. The parameters include timeout value, the expected patterns and the parameter values to be passed remotely to the test program.

6.4.5 Set up controlling program

At the top level the set-up controlling program starts the process and invokes other components such as the log monitoring program. During the execution it receives information regarding the state of the set-up from the log monitor script and takes the appropriate action. It may happen that during the test execution, a fault occurs in the hardware of the validation set-up which disables further communication between the PC and the validation kit. In such events, the programs running on the PC, tries to restart the validation process without any success increasing the size of the log file. The size of the log file is monitored by this program and it terminates, if the size exceeds a predefined value.

For different IPs and SoCs, the framework supports parameters to easily change its behavior. A set-up configuration file is created which contains values of all the parameters. The set-up controlling program reads this file and configures the framework accordingly.

6.5 Work flow

First the set-up controlling program is started on the PC by the validation engineer. It reads the set-up configuration file and stores the values of different parameters present in the configuration file. The parameters are passed to different parts of the framework

as required through the set-up controlling program. Then it initiates a Linux boot on the SoC using the JTAG interface. After the Linux boot is completed, the log monitor program is started which initiates a telnet session with the operating system running on the SoC. The log monitor program synchronizes the PC with the validation kit and remotely starts the testing program. The testing program checks for the last known working state from the status record to continue from there. It reads the tests that have not yet been performed from the test database, and executes them by decoding the keywords and determining the associated data required for the test from the arguments. The test database is centrally stored on a network file system which is mounted into the root file system of the SoC during the boot process. As the tests are executed, messages are sent to the PC from the validation board which are constantly monitored by the log monitor program. It takes necessary actions for carrying forward the test execution process depending on the log content. The actions include starting the testing program, communicating an event of reboot, timeout or completion of the process to the set-up controlling program by terminating with different exit values which are monitored by the set-up controlling program. Based on the value, it reboots the validation kit over JTAG, starts a serial terminal program or terminates. The above flow continues until all the tests have been executed on the SoC. At the end of the validation process, a detailed log file is created on the PC.

The log file must be analyzed by the validation engineer in order to conclude the tests. No golden response obtained from the simulations is available for the tests and the validation engineer decides the pass or fail criteria based on the information present in the datasheets and specifications.

6.6 Example

6.6.1 Sample Linux device driver

The figure 6.2 shows the major functions of a Linux device driver for a NAND controller IP. They include erasing, reading, writing, switching modes, configuring DMA controller and initially configuring the NAND controller. When Linux boots up, the driver performs the initial configuration of the NAND controller and allocates a memory in which all information about the NAND controller is stored, which is used by the functions of its device driver.

```
erase_n_blocks(controller_structure, start_address, number_of_blocks);
erase_chip(controller_structure);
read_sector(controller_structure, start_address, buffer);
read_page(controller_structure, start_addess, buffer);
read(controller_structure, start_address, buffer, length);
write_sector(controller_structure, start_address, buffer);
write_page(controller_structure, start_address, buffer);
write(controller_structure, start_address, buffer, length);
configure_dma(controller_structure, transfer_size, read_or_write_dma);
switch_mode_global(controller_structure, global_mode);
switch_mode_local(controller_structure, local_mode);
initialize_controller(&controller_structure);
probe_controller();
module_init();
module_exit();
```

Figure 6.2: Sample Linux device driver

6.6.2 Support in device driver

For creating a sysfs interface in the device driver, Linux kernel offers standard kernel functions. The steps required are

- Declare a pointer to a kernel object.
- Declare the attributes.
- Create and attach the show and store functions for each attribute.
- Register all the attributes under an attribute group.
- Create a kernel object and assign it to the kernel object pointer.
- Create the interface using the kernel object and the attribute group.

The sample interface for the NAND flash controller has parameters namely *address, size, mode, id, write_reg_param* and *read_reg_param*. When the sysfs interface for the driver is created, it is visible in the Linux virtual file system and its attributes are written and read using the standard Linux *echo* and *cat* commands respectively. For

the sample interface, the kernel object is *sample_object* visible under /sys, the attribute group is *attribute_group* located in /sys/sample_object with the attributes in /sys/sample_object/attribute_group/. The *address, size* and *mode* attributes are used to pass new values to the device driver functions from user space. The attributes *write_reg_param* and *read_reg_param* are used for writing and reading the IP registers. The *id* attribute is used to call the driver function that is assigned the corresponding ID. Figure 6.3 shows a sample show function associated with the *id* attribute of the sysfs interface. The *buffer_to_file* and *file_to_buffer* functions have been added to write the local buffer data to a predefined file and to read the data of a predefined file into a local buffer.

```
switch(id)
{
    case 1: erase_blocks( controller_struct, address, size);
            break;
    case 2: erase_chip(controller_struct);
            break;
    case 3: read(controller_struct, address, buffer, size);
            buffer_to_file(buffer);
            break;
    case 4: file_to_buffer(buffer);
             write(controller_struct, address, buffer, size);
             break;
    case 5: configure_dma(controller_struc, size, mode);
             break;
    case 6: switch_mode_local(controller_struct, mode);
             break;
    default: break;
}
```

Figure 6.3: Sample show function for attribute id

6.6.3 Sample test case

Figure 6.4 shows a sample test case for writing a file of size two kilobytes to the NAND flash at zero address without the error correcting codes (ECC) data. This test case is read by the test program, decoded and executed. In the test case, first the flash is

erased, accomplished by filling the *id* attribute with a value assign to the *erase_chip* function, two in the sample case, and then reading the *id* attribute. When the *id* attribute is read, its show function is executed which calls the *erase_chip* function of the sample driver. The controller is switched into the no ECC mode by modifying the ECC configuration register. Then the *address, size* and *id* parameters are filled with appropriate values. The *size* attribute is filled with 800, which is the hexadecimal equivalent of two kilobytes, *address* with zero, and *id* with four. Then a predefined file is written into the flash by reading the *id* attribute. To verify that no ECC was generated, the NAND flash is read from address zero till the end of the file, including the ECC data which increases the overall file size by 6.25% to 2176 bytes or 880 in hexadecimal. For this, the controller is switched to another mode in which the read function reads the ECC data, along with the normal data, from the NAND flash. Then the read function, having id three, is called which creates a file *test_read* containing the contents read from NAND flash. The *test_read* file is renamed appropriately to prevent its overwriting in execution of another test case involving a read from flash. The appropriate device driver functions are executed by passing their respective IDs and other parameters, if required, from the user space through the different attributes defined in the sysfs interface. Thus a test is accomplished by using the code of the device driver. Similarly other test cases are created which are executed sequentially by the test program.

```
DESCRIPTION Check no ECC generation
WRITE_SYS_ATTRIBUTE 2@id
READ_SYS_ATTRIBUTE id
WRITE_REGISTER noecc@ecc_config_reg
WRITE_SYS_ATTRIBUTE 0@address
WRITE_SYS_ATTRIBUTE 800@size
WRITE_SYS_ATTRIBUTE 4@id
READ_SYS_ATTRIBUTE id
WRITE_SYS_ATTRIBUTE 2@mode
WRITE_SYS_ATTRIBUTE 880@size
WRITE_SYS_ATTRIBUTE 3@id
READ_SYS_ATTRIBUTE id
APPLICATION mv test_read read_2K_with_ecc_area
```

Figure 6.4: Sample test case

The output of the test program when the sample test case is executed, is shown in figure 6.5

```
echo 2 > /sys/sample_object/attribute_group/id
cat /sys/sample_object/attribute_group/id
echo noecc@ecc_config_reg > /sys/sample_object/attribute_group/write_reg_param
echo 0 > /sys/sample_object/attribute_group/address
echo 800 > /sys/sample_object/attribute_group/size
echo 4 > /sys/sample_object/attribute_group/id
cat /sys/sample_object/attribute_group/id
echo 2 > /sys/sample_object/attribute_group/mode
echo 880 > /sys/sample_object/attribute_group/size
echo 3 > /sys/sample_object/attribute_group/id
cat /sys/sample_object/attribute_group/id
mv test_read read_2k_with_ecc_area
```

Figure 6.5: Output of the test case from test program

```
set PROGRAM [lindex $argv 0]
set IP [lindex $argv 1]
set TIMEOUT [lindex $argv 2]
set timeout $TIMEOUT
if { $PROGRAM == 1 }
{
    spawn <terminal_program>
}
else
{
    spawn telnet $IP
}
expect
{
    timeout { exit 1 }
    "linux_prompt" { send "<commands to start the test program>\r" }
    "rebooting" { exit 2 }
    "validated" { exit 3 }
    "bootloader_prompt" { exit 4 }
}
```

Figure 6.6: Sample log monitor program

6.6.4 Sample log monitor program

A sample log monitor program in figure 6.6 spawns a telnet or terminal program based on the PROGRAM argument passed to it. Then it monitors the log received in the

spawned program and either exits with different values or sends commands for starting the test program on the SoC. The argument TIMEOUT sets the time to wait for a string match. If no matching string is received in that time, the program terminates. The IP argument is further used as a parameter for the telnet program.

6.6.5 Sample set-up controlling program

In figure 6.7 an excerpt of a set-up controlling program is shown. First the parameters are set and then the test execution process starts with booting Linux on SoC over JTAG for validation. Then the log monitor program is started with appropriate parameters passed to it, to monitor the log. The set-up control program waits to collect the exit value from the log monitor program and acts accordingly on receiving the value. For instance, when the bootloader prompt is detected by the log monitor program, it terminates with an exit value which forces the set-up controlling program to restart the test execution, to continue validation. This process continues until all the tests are executed and the *validated* string is received.

```
$program=0
$ip=x.x.x.x
$timeout=50
<Boot Linux on SoC over JTAG>
while [ $exit_val -ne 3 ]
do
    <Start log_monitor_program $program> $ip $timeout
    exit_val = $?
    case $exit_val in
                    1)  <Boot Linux on SoC over JTAG>
                     ;;
                    2) program=1
                     ;;
                    3) echo "Completed"
                     ;;
                    4) program=0
                       <Boot Linux on SoC over JTAG>
                     ;;
                    *) exit
    esac
done
```

Figure 6.7: Sample set-up controlling program

Chapter 7

Results

The above methodology is applied successfully to a NAND and an SPI NOR flash controller IP. The test plan for these IPs involves tests which require SoC reboots and both the IPs work with external flash memory devices which are manufactured by different vendors so the tests are run for each of them separately.

7.1 Conventional procedure

The conventional validation procedure uses a combination of approaches mentioned in chapter 4. The basic flow involves booting Linux on the SoC using the JTAG interface. Then the registers are tested and modified to apply a test sequence, which is achieved either using the debugger or pre-compiled test binaries. This is followed by checking the hardware response recorded in the register values using the debugger, and the received log. Next the SoC is rebooted if required, or the execution of the remaining tests is continued. All the above actions require continuous manual intervention.

7.2 NAND controller

The NAND controller IP is responsible for managing all transfers that take place between the CPU core or DMA engine and the external NAND flash device. In addition it has single and multiple bit error correction capabilities, up to 30 bits per 1024 bytes, using Hamming and BCH codes respectively. The bit error tolerance is dependent on the mode in which the NAND controller operates and the size of the external flash device. It has an engine in which code sequences of various operations supported by the

NAND flash, can be programmed and the controller generates the complying control and data signals which when physically routed into the NAND flash, accomplish the intended operation inside the NAND flash. It also supports DMA in some modes, when the related registers are programmed accordingly. Booting from NAND flash is also a key feature of this controller allowing to boot even when initial blocks of the NAND flash are faulty by performing address remapping. The boot mode is read only while reading and writing are both supported in the data mode. Both, boot and data, are sub-modes under two global modes. Switching among them is possible by programming the appropriate registers. One of the global modes has DMA support and is used for bulk data transfers, the other is used when small number of bytes, typically less than 1024 bytes, need to be transferred.

7.3 SPI NOR controller

Similar to the NAND controller mentioned in the previous section, the SPI NOR controller accomplishes the same tasks using NOR flash devices. A NOR flash works in various modes using different numbers of data lines. In normal mode the SPI NOR flash uses only one line for transfers. In dual mode two lines and in quad mode four lines are used for transfers. In addition it has a performance enhancement mode used in combination with the previously mentioned modes. The IP supports these flash modes in its read only and read/write modes. It has a controller which generates the appropriate signals on the lines according to the values programmed in its registers completing the intended operation in the NOR flash.

Table 7.1: Description of engineers performing validation

Engineer#	Experience	Frequency of performing validation(approx.)	
		NAND IP	SPI NOR IP
Engineer 1	5 years	3 months	N.A.
Engineer 2	2 years	6 months	1 month
Engineer 3	6 months	2 month	N.A.
Engineer 4	10 months	2 weeks	6 months
Engineer 5	2 years	N.A.	1 month

Various experiments were conducted using the automation framework and their results are summarized in this section. The effectiveness of using the automation framework is verified by comparing the time or man hours requirement for PSV of the two IPs under test. The time required for validation by engineers having different levels of experience, is compared with the time required using the automation framework. Table 7.1 describes the engineers in terms of experience and frequency of performing validation of the NAND and the SPI NOR controller IPs .

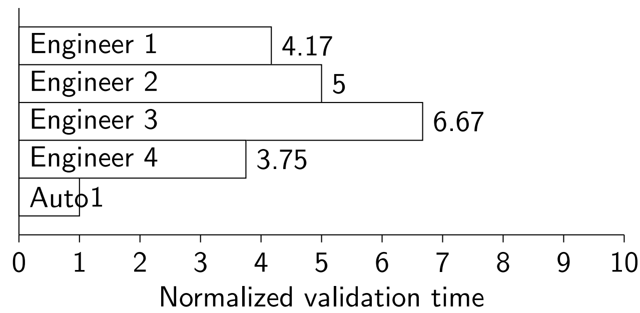

Figure 7.1: Time for validation with a single NAND flash normalized with respect to time taken for the same using the automation framework

The experience of an engineer greatly affects the time invested in validation while following the conventional way. Figure 7.1 shows the relative average timing of the engineers for fully testing the NAND controller IP against a single NAND flash device normalized with respect to the time taken for the same by engineer 4 using the proposed automation framework. Even the best time without using framework takes more than three times the time taken with the help of automation.

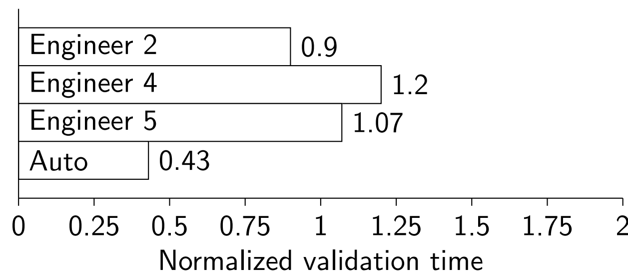

Figure 7.2: Time for validation with a single SPI NOR flash normalized with respect to time taken for single NAND flash validation using the automation framework by engineer 4

Similar results are shown for SPI NOR controller IP in figure 7.2. All the timing values are normalized with respect to the time taken for single NAND flash validation using the automation framework by engineer 4. It is evident that using the framework greatly reduces the time invested in validation.

In another experiment, the automation framework was tested on multiple validation set-ups. Only a few modifications in the configuration file are required to ready the set-up for a different board. A validation set-up has various components which affects the time taken by the tests to execute. If the version of the Linux kernel, validation kit or SoC changes, then the set-up is considered as different as these can impact the time of various steps involved in a test. As mentioned in previous sections, tests involve booting Linux over JTAG, connecting the debugger for testing addresses, rebooting from flash which can have different timing values for multiple set-ups. Table 7.2 describes the three set-ups.

Table 7.2: Description of the validation set-ups

Setup #	System On Chip used	Linux Kernel version	Validation kit used
1	SoC 1	3.10.27	Kit 1
2	SoC 2	3.10.53	Kit 1
3	SoC 3	3.10.53	Kit 2

NAND validation was performed on each of these set-ups by one engineer with and without using the automation framework. As reported in table 7.3, the time taken for validation on different set-ups using the automation framework is almost the same. The entire framework is reused for different set-ups with modifications made in the configuration file.

Table 7.3: Validation times of NAND controller IP on different set-ups using the automation framework normalized with respect to time taken for single NAND flash validation using the automation framework by engineer 4

Setup #	Normalized validation time
1	1
2	1.08
3	1.08

Similar experiment was performed for the SPI NOR controller IP. The results are reported in table 7.4.

Table 7.4: Validation times of SPI NOR controller IP on different set-ups using the automation framework normalized with respect to time taken for single NAND flash validation using the automation framework by engineer 4

Setup #	Normalized validation time
1	0.4
2	0.44
3	0.42

Validation involves test execution and result analysis. The combined time taken in both these activities, is considered as time for validation. This division is exploited while validating multiple flashes using the automation framework. While another flash memory device undergoes test execution, the validation engineer analyzes the log from the previous test. This is not possible while performing validation on multiple flashes without using the automation framework as the validation engineer remains involved in running the test cases and analyzing the results after every test for each flash device. The automation framework requires the values of the supported parameters, which are provided via a configuration file. The configuration file must be created before using the automation framework on different set-ups. When the set-up remains the same, the creation of the configuration file is a one time activity as the configuration file is

reused. An example timeline for validation of three flashes using the same set-up is shown in figure 7.3.

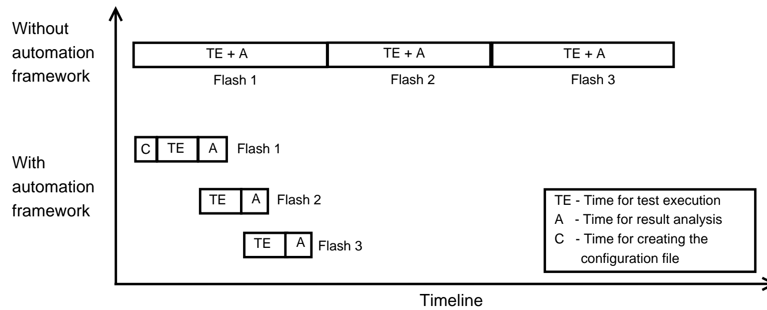

Figure 7.3: Timeline for validation with multiple flashes using the same set-up

Figure 7.4 shows the time taken for validation with different number of flashes by one engineer on the same set-up when the automation framework is used, in terms of the validation time taken for single NAND flash validation using the automation framework by engineer 4. It can be seen from the graph that the time required for validation relatively decreases as the number of flash memory devices increase.

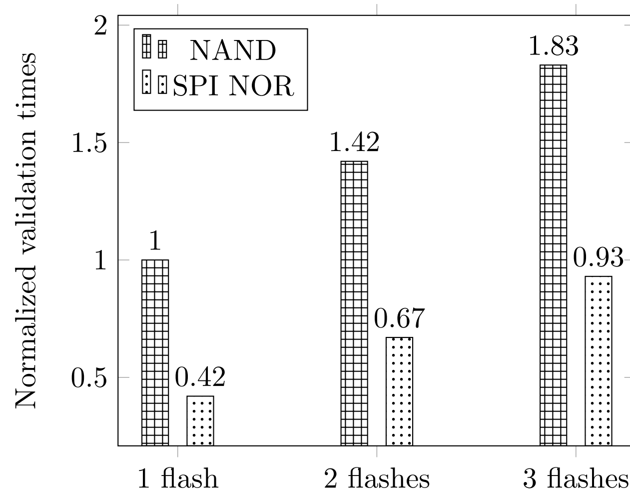

Figure 7.4: Validation times of different number of flashes using the automation framework normalized with respect to time taken for single NAND flash validation using the automation framework by engineer 4

Chapter 8

Conclusion

In this work, a method is proposed for bringing together the functional validation of the hardware and the embedded software, related to an IP in an SoC. The architecture of an automation framework using the proposed methodology is also presented. This work targets the changes required in the validation procedure for modern SoCs to allow at-speed testing, hardware and software co-validation and validation time reduction.

In the proposed methodology, the tests are created using the software which directly interacts with the hardware validating both the hardware and the software simultaneously. Issues related to timings, hardware and software compatibility can be detected earlier using this methodology, compared to when the hardware and the software validation are performed separately. The methodology is successfully applied to a NAND and an SPI NOR flash controller IP using the automation framework. Its effectiveness is shown by the reduction in the average validation times of the NAND and the SPI NOR flash controller IP, of 76% and 61% the validation time without using the automation framework.

The future work includes implementing a robust automatic log analysis scheme which would further reduce the manual intervention allowing additional savings of resources.

References

[1] Subhasish Mitra, Sanjit A Seshia, and Nicola Nicolici. **Post-silicon validation opportunities, challenges and recent advances**. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 12–17. IEEE, 2010. 2

[2] P Manikandan, Juhee Mala, and S Balamurugan. **Post Silicon Functional Validation from an Industrial Perspective**. *Middle-East Journal of Scientific Research*, **15**(11):1570–1574, 2013. 3, 14, 15

[3] Jagannath Keshava, Nagib Hakim, and Chinna Prudvi. **Post-silicon validation challenges: how EDA and academia can help**. In *Proceedings of the 47th Design Automation Conference*, pages 3–7. ACM, 2010. 3, 4

[4] Mehdi Karimibiuki, Kyle Balston, Alan J Hu, and Andre Ivanov. **Post-silicon code coverage evaluation with reduced area overhead for functional verification of SoC**. In *High Level Design Validation and Test Workshop (HLDVT), 2011 IEEE International*, pages 92–97. IEEE, 2011. 3, 6, 11, 12, 15

[5] Debashis Panigrahi, Clark N Taylor, and Sujit Dey. **Interface based hardware/software validation of a system-on-chip**. In *High-Level Design Validation and Test Workshop, 2000. Proceedings. IEEE International*, pages 53–58. IEEE, 2000. 3, 7, 10, 12

[6] Ilya Wagner and Valeria Bertacco. **Reversi: Post-silicon validation system for modern microprocessors**. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 307–314. IEEE, 2008. 3, 7, 12

[7] Tommy Bojan, Igor Frumkin, and Robert Mauri. **Intel first ever converged core functional validation experience: Methodologies, challenges, results and learning**. In *Microprocessor Test and Verification, 2007. MTV'07. Eighth International Workshop on*, pages 85–90. IEEE, 2007. 3, 12, 15

[8] Yael Abarbanel, Eli Singerman, and Moshe Y Vardi. **Validation of SoC firmware-hardware flows: Challenges and solution directions**. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–4. IEEE, 2014. 4, 10, 16

[9] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. **Coverage evaluation of post-silicon validation tests with virtual prototypes**. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 318. European Design and Automation Association, 2014. 5, 6, 13

[10] Ho Fai Ko, Adam B Kinsman, and Nicola Nicolici. **Distributed embedded logic analysis for post-silicon validation of SOCs**. In *Test Conference, 2008. ITC 2008. IEEE International*, pages 1–10. IEEE, 2008. 6

[11] Miron Abramovici. **In-system silicon validation and debug**. *Design & Test of Computers, IEEE*, **25**(3):216–223, 2008. 6

[12] Xiao Liu and Qiang Xu. **Trace signal selection for visibility enhancement in post-silicon validation**. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1338–1343. European Design and Automation Association, 2009. 6

[13] Kanad Basu and Prabhat Mishra. **Efficient trace signal selection for post silicon validation and debug**. In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pages 352–357. IEEE, 2011. 7

[14] Angela Krstic, Wei-Cheng Lai, Kwang-Ting Cheng, Li Chen, and Sujit Dey. **Embedded software-based self-testing for SoC design**. In *Proceedings of the 39th annual Design Automation Conference*, pages 355–360. ACM, 2002. 8, 9, 15

[15] Lionel Blanc, Amar Bouali, Jérôme Dormoy, and Olivier Meunier. **A Methodology for SoC Top-Level Validation using Esterel Studio**. In *Electronic Design Process Workshop (EDP)*. IEEE/DACT, 2002. 11, 15

[16] Patrick Mochel. **The sysfs filesystem**. In *Linux Symposium*, page 313, 2005. 19

[17] Don Libes. *"Exploring Expect: A Tcl-based toolkit for automating interactive programs"*. O'Reilly Media, Inc., 1995. 19

[18] Amir Nahir, Avi Ziv, Rajesh Galivanche, Alan Hu, Miron Abramovici, Albert Camilleri, Bob Bentley, Harry Foster, Valeria Bertacco, and Shakti Kapoor. **Bridging pre-silicon verification and post-silicon validation**. In *Proceedings of the 47th Design Automation Conference*, pages 94–95. ACM, 2010.

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other examination board.

The thesis work was conducted from July 2014 to June 2015 under the supervision of Dr. Alexander Fell.

Delhi,