

INTERACTIVE OFFLINE INTERFACE TO DEBUG
SIMULATION FAILURE

and

IMPROVED DUAL SIMULATION APPROACH
TO GATESIM

Thesis

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF TECHNOLOGY

in

VLSI DESIGN

by

MEERA MOHAN

(Reg. No.: 11VL09F)



DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA,
SURATHKAL

MANGALORE - 575025

JUNE 2013

INTERACTIVE OFFLINE INTERFACE TO DEBUG SIMULATION FAILURE

and

IMPROVED DUAL SIMULATION APPROACH TO GATESIM

by

Meera Mohan

(Reg. No.: 11VL09F)

Under the Guidance of

Dr. M. S. Bhat

Professor

Dept of E & C, NITK

Mr. Narendran. K

SMTS Design Engineer

AMD India Pvt. Ltd



Thesis

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF TECHNOLOGY

in

VLSI DESIGN

Department of Electronics and Communication Engineering

National Institute of Technology Karnataka, Surathkal

Mangalore - 575025

June 2013

DECLARATION

by the M.Tech student

I hereby *declare* that the Report of the P.G. Project Work entitled **Interactive Offline Interface To Debug Simulation Failure and Dual Simulation Approach To Gatesim** which is being submitted to **National Institute of Technology Karnataka, Surathkal**, in partial fulfillment for the requirements of the award of degree of **Master of Technology in VLSI Design** in the department of **Electronics and Communication Engineering**, is a *bonafide report of the work carried out by me*. The material contained in this report has not been submitted at any other University or Institution for the award of any degree.

11VL09F, Meera Mohan,

(Register Number, Name and Signature of the student)

Department of Electronics and Communication Engineering

Place : NITK, Surathkal

Date :

CERTIFICATE

This is to *certify* that the Post Graduation Project Work Report entitled **Interactive Offline Interface To Debug Simulation Failure and Dual Simulation Approach To Gatesim** submitted by **Meera Mohan** (Register number: 11VL09F) as the record of the work carried out by him/her, is *accepted as the P.G Project Work Report submitted* in partial fulfillment for the requirements of the award of degree of **Master of Technology in VLSI Design** in the department of **Electronics and Communication** at **National Institute of Technology Karnataka, Surathkal** during the academic year 2012-2013.

Mr. Narendran Kumaragurunathan
Project Guide

Prof. M. S. Bhat
Project Guide

Prof. Muralidhar Kulkarni
Chairman DPGC

ABSTRACT

PART I- Interactive Offline Interface To Debug Simulation Failure

Processor execution logs created during simulation, contains in depth details pertaining to processor execution. In the event of a simulation failure, debugging necessitates tracing through the execution logs for failure diagnosis. Due to comprehensive information contained in these log files, it becomes overwhelming to comprehend it quickly enough as information is spread across. Such manual tracing is error-prone and time consuming. This project aims to implement a GUI and thereby enable effortless, faster execution debug even from remote locations. The interface gathers data from multiple sources related to execution flow and represents it correlated, as appropriate. The graphic interactive navigation windows attempt to reduce the user's time spent on tracing cause of failure. The project attempts to use web based technologies to enable remote debug.

Keywords: *Debug Interface, Execution Log Files, SoC Verification.*

PART II- Improved Dual Simulation Approach To Gatesim

Gatesim or gate level simulation verification focuses on verifying the post layout netlist of the design. Gatesim verification is an important milestone and confidence builder for verification. Multiple methodologies are in existence in the industry for this purpose. In AMD, Gatesim verification uses co-simulation methodology for the purpose, where full chip behavioral RTL and gate netlist are simulated simultaneously in one simulation. Verification is achieved by driving corresponding RTL stimulus onto netlist and by comparing response every cycle. The objective of this project is to improve the simulation turn-around times and reducing resource requirements involved in Gatesim verification without compromising on verification. The thesis proposes a new dual simulation approach to Gatesim where RTL and gate level simulations are performed separately, by exporting test vectors for netlist simulation from RTL simulation. The approach attempts to overcome performance issues with current co-simulation methodology.

Keywords: *Gatesim, GLS, Co-simulation Methodology, Functional Verification, Netlist Simulation.*

Contents

ACKNOWLEDGEMENTS	i
ABSTRACT	ii
CONTENTS	iii
LIST OF FIGURES	iv
LIST OF TABLES	v
LIST OF ALGORITHMS	vi
SYMBOLS AND ABBREVIATIONS	1
I	1
1 INTRODUCTION	2
1.1 VERIFICATION METHODS	3
1.2 SoC VERIFICATION	3
1.3 PROPOSED ENHANCEMENT TO AID PROCESSOR EXECUTION DEBUG	4
1.4 ORGANIZATION OF THE THESIS	4
2 AMD64 ARCHITECTURE OVERVIEW	6
2.1 AMD64 ARCHITECTURE	6
2.2 FEATURES OF AMD64	7
2.3 MEMORY ORGANIZATION	9
2.4 MEMORY MANAGEMENT	10
3 VERIFICATION ENVIRONMENT	13

3.1	FUNCTIONAL VERIFICATION	13
3.2	VERIFICATION	13
3.3	DEBUGGING A SELF TEST FAILURE	15
4	PROPOSED ENHANCEMENT	19
4.1	PROCESSOR EXECUTION DETAILS	19
4.2	SAMPLE DEBUG CASES IN CONSIDERATION	20
4.3	NECESSITY TO ENHANCE MANUAL VERIFICATION	25
4.4	ENHANCING MANUAL VERIFICATION	25
5	INTERFACE IMPLEMENTATION	27
5.1	INTERFACE	27
5.1.2	INTERFACE	29
5.1.3	JS DATASTRUCTURE	30
5.2	EXTRACTION OF INFORMATION	31
5.2.1	STIMULUS	31
5.2.2	EXECUTION LOG	32
5.2.3	INFORMATION CORRELATION	33
5.3	GUI FEATURES	35
5.3.1	EXECUTION FLOW GRAPH	35
5.3.2	REGISTER WATCH WINDOW	36
5.3.3	INSTRUCTION WINDOW	37
5.3.4	EXECUTION LOG WINDOW	38
5.3.5	MARKERS	38
6	RESULTS: GRAPHIC USER INTERFACE	39
6.1	EXECUTION FLOW GRAPH	39
6.2	REGISTER WATCH WINDOW	40
6.3	INSTRUCTION WINDOW	41
7	CONCLUSION	43

II	44
8 INTRODUCTION	45
8.1 GATE LEVEL SIMULATION	46
8.2 ORGANIZATION OF THE THESIS	47
9 GATE LEVEL SIMULATION	48
9.1 NEED FOR GATESIM	49
9.2 LIMITATIONS OF LEC AND STA	50
9.3 ISSUES CAUGHT BY GATESIM	51
9.4 ISSUES FACED BY GATESIM	52
10 GATESIM METHODOLOGIES	53
10.1 EARLY DUAL-SIM METHODOLOGY	53
10.2 CO-SIM BASED GATESIM METHODOLOGY	55
11 IMPROVED DUAL-SIM APPROACH TO GATESIM	59
11.1 DUAL-SIM FLOW	60
11.2 IMPLEMENTATION	61
12 RESULTS	70
12.1 SIMULATION PERFORMANCE ANALYSIS	71
12.2 MEMORY REQUIREMENT	71
13 CONCLUSION	72
14 REFERENCES	73

List of Figures

2.1	AMD64 Registers	8
2.2	Segmented Memory Model	11
2.3	Paged Memory Model	12
3.1	Self-Test	15
3.2	RTL-Reference Model Cosimulation	16
4.1	Illustration of debug process : Test A	22
4.2	Illustration of debug process : Test B	24
5.1	GUI Implementation	29
5.2	Web Page Generation	30
5.3	Assembler	32
5.4	Asm List File Extraction	33
6.1	Execution Flow Graph	40
6.2	Execution Graph With Branching and Memory Writes	40
6.3	Register Watch Window	41
6.4	Register Value Comparison	41
6.5	Instruction Window	42
9.1	Design Flow	49
10.1	Early Dual-Sim Flow	54
10.2	Co-sim Based Gatesim	55
10.3	Co-sim Based Gatesim Flow	56

11.1 Dual Sim	60
11.2 RTL Simulation	62
11.3 Accessing FSDB Signals	63
11.4 API For Accessing FSDB Signals	64
11.5 C++ Routine for FSDB Signal Extraction	65
11.6 FSDB Format to DKF format	66
11.7 Netlist Simulation	67
11.8 Dummy RTL	68
11.9 Netlist Instantiation	69
11.10 Cycle Compare Output Signals	69

List of Tables

12.1 Simulation Performance Comparison	71
12.2 Simulation Memory Requirement	71

List of Algorithms

1	Memory Read-Write	21
2	String Lower Case Conversion	23
3	Creating JavaScript Object	31
4	Combining List and Log File Information	34
5	Creating Execution Graph	36
6	Creating Register Window	37
7	Creating Instruction and Execution Log Window	38

SYMBOLS AND ABBREVIATIONS

API Application Programming Interface

CSS Cascading Style Sheets

DUT Design Under Test

FSDB Fast Signal Database

GUI Graphical User Interface

HTML Hyper Text Markup Language

ILS Instruction Level Simulator

IP intellectual Property

JS JavaScript

LEC Logic Equivalence Check

PC Program Counter

RTL Register Transfer Level description of circuit

SoC System on Chip

SP Stack Pointer

SR Status Register

STA Static Timing Analysis

VCD Value Change Dump

Part I

Chapter 1

INTRODUCTION

With rapid growth of deep sub micron technology, there has been an aggressive shrinking in physical dimension of silicon structures, that can be realized on silicon. This advancement has enabled the transition of multi-million gate designs from large printed circuit boards to SoC (System on Chip) . SoC design has the advantages of smaller size, low power consumption, reliability, performance improvement and low cost per gate. Another major high point of SoC from design point of view is that SoC allows use of predesigned blocks called semiconductor intellectual property (IP) . These hardware IP blocks can be mix-and-matched, thereby providing design reuse in SoC and thereby reducing time-to-market.

Over the past few years, major challenges faced by semiconductor industry is to develop more complex SoCs with greater functionality and diversity with reduction in time-to-market. One of the main challenge among this is verification. Integration between various components, combined complexity of multiple sub-systems, software-hardware co-verification, conflicts in accessing shared resource, arbitration problems and dead-locks, priority conflicts in exception handling etc makes SoC verification very hard. It is said that verification consumes more than 60 percent of design effort. This can be easily explained as there is no single design tool that can completely verify a SoC on it's own. Instead a complex sequence of tools and techniques, including simulation, directed and random verification and formal verification are used to verify a SoC. Achieving cent percent functional verification coverage is next to impossible due

to time-to-market constraints.

1.1 VERIFICATION METHODS

Two widely adopted verification methodologies are stimulus based dynamic verification methodology and static formal verification methodology. In stimulus based verification the verification engineer develops a set of tests, based on design specifications. Design correctness is then established through simulations. Formal verification is a mathematical proof method of ensuring that a design's implementation matches its specification. The most prominent distinction between stimulus-based verification and formal verification is that the former requires input vectors and the latter does not. In stimulus based verification the idea is to first generate input vectors and then derive the reference output where as in formal verification process it is the reverse approach.

In formal verification the behavior of design is captured as a set of mathematical equations called properties and then the formal verification tool proves or disproves each property appropriately. In formal verification, user need not generate stimulus but specification of invalid stimuli would be necessary. At SoC level the designs are huge, typically beyond the capacities of automatic formal tools. Formal methodology tools have large memory and long run time requirements. When memory capacity is exceeded, tools often shed little light on what went wrong, or give little guidance to fix the problem. As a result, formal verification software, is best suited only to circuits of moderate size, such as blocks or modules.

1.2 SoC VERIFICATION

Most SoCs are built around one or more processing cores (multi processor) and verification is done using stimulus based verification methodology where the design model is simulated using random or handwritten test programs. Reference models are simulated in parallel with the design and results are compared. Comparison between the design architecture state and the reference model states are done after each instruction retire. Difference detected in states is considered as “*mismatches*”. Memory contents

are compared at the end of simulation and any discrepancies are reported as “*memory mismatches*”. The simulator writes entries for each event it processes into processor execution log file.

On event of a simulation mismatch, the processor execution log entries are helpful in understanding and tracing the cause associated with the failure. Logs contain in-depth details pertaining to processor execution.

1.3 PROPOSED ENHANCEMENT TO AID PROCESSOR EXECUTION DEBUG

Tracing a typical failure is a manual process and is time consuming since relevant informations are buried under a wealth of information, and related informations are spread across in different files. This greatly challenges an engineer’s ability to debug and converge on the cause. If there exists a representation which correlates different information and presents them as required by the user, it would aid debug significantly. The proposed interactive interface provides graphical data representations and navigation, helping in faster tracing through processor execution. Such represented information is correlated, filtered and sorted making debugging a lot more intuitive than existing manual method.

Such an envisioned graphical user interface is proposed to aid processor execution debug by representation of data to user. This interface would have properties of a software debugger, and it would work offline. This project work concentrates on implementing such an enhancement.

1.4 ORGANIZATION OF THE THESIS

The organization of this project report is as follows:

Chapter 2-AMD64 Architecture Overview gives brief introduction to AMD64 architecture.

Chapter 3-Verification Shortfalls discusses existing verification methodologies and their

shortfalls.

Chapter 4-Proposed Enhancement discusses desired features of proposed enhancement.

Chapter 5-Implementation of Proposed Enhancement gives implementation details of proposed enhancement.

Chapter 6-Implementation Results a final look at the implemented enhancement.

Chapter 2

AMD64 ARCHITECTURE OVERVIEW

Typically a SoC will bring together functionality that used to be distributed across chips or maybe even devices. Various functional modules of the system are integrated into a single chip set. Naturally SoC is a very complex design which will include programming elements, hardware elements, software elements, bus architecture, clock and power distribution, test structures etc.

The heart of any SoC design is the core which is nothing but some sort of processor, and practically all SoC must have at least one processor. In AMD, most processors are based on AMD64 architecture. Most of the verification scenarios are developed in reference to the behavior of core and hence makes it most important design of interest.

2.1 AMD64 ARCHITECTURE

The AMD64, originally called x86-64, architecture is a 64-bit, backward compatible extension of industry-standard x86 architecture (legacy). It adds 64-bit addressing and expands register resources to support higher performance for recompiled 64-bit programs, while supporting legacy 16-bit and 32-bit applications and operating systems without modification or recompilation. The need for a 64 bit x86 architecture is driven by applications that address large amounts of virtual and physical memory, such as

high-performance servers, database management systems, and CAD tools. These applications benefit from both 64-bit addresses and an increased number of registers.

2.2 FEATURES OF AMD64

The main features of AMD64 architecture are its extended 64-bit registers and new 64-bit mode of operation.

2.2.1 REGISTERS

One of the main features of AMD64 architecture is the 64-bit register extension. The small number of registers available in the legacy x86 architecture limits performance in computation-intensive applications. Increasing the number of registers provides a performance boost to many such applications. In addition to the 8 legacy x86 General-Purpose Registers (GPRs), AMD64 introduces additional 8 GPRs. All 16 GPRs are 64-bit long and an instruction prefix (REX) accesses the extended registers. The architecture also introduces 8 new 128-bit media registers.

Figure 2.1 shows the AMD64 application-programming register set. They include the general-purpose registers (GPRs), segment registers, flags register (64-bits), instruction-pointer register (64-bits) and the media registers.

2.2.2 MODES OF OPERATION

In addition to the x86 legacy mode, another major feature of AMD64 is its long mode. This is the mode where a 64-bit application (or operating system) can access 64-bit instructions and registers. The different modes of operations in AMD64 architecture are detailed below:

Long Mode: Long mode is an extension of legacy protected mode. It consists of two sub modes: 64-bit mode and compatibility mode. 64-bit mode supports all of the new features and register extensions of the AMD64 architecture. Compatibility supports binary compatibility with existing 16-bit and 32-bit applications. Long mode

2. AMD64 ARCHITECTURE OVERVIEW

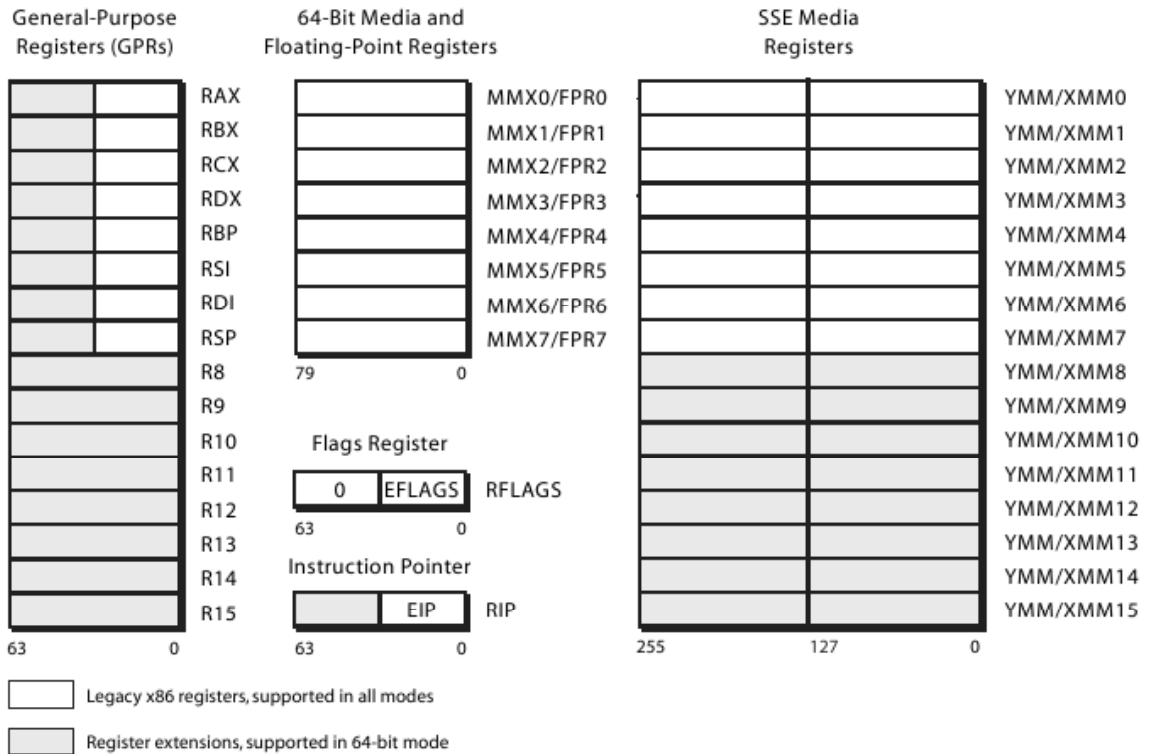


Figure 2.1: AMD64 Registers

does not support legacy real mode or legacy virtual-8086 mode, and it does not support hardware task switching.

- **64-bit mode:** 64-bit mode supports the full range of 64-bit virtual-addressing and register-extension features. This mode is enabled by the operating system on an individual code segment basis. As 64-bit mode supports a 64-bit virtual-address space, it requires a 64-bit operating system and tool chain.
- **Compatibility Mode:** Compatibility mode allows 64-bit operating systems to run existing 16-bit and 32-bit x86 applications. These legacy applications run in compatibility mode without recompilation. This mode is also enabled by operating system on an individual code segment bases as in 64 bit mode. However unlike 64-bit mode x86 segmentation function similar to legacy x86 architecture using 16-bit or 32-bit protected mode semantics.

Legacy Mode: Legacy mode has compatibility existing 16-bit and 32-bit operating systems in addition to compatibility with existing 16-bit and 32-bit application. Legacy

mode has the following three submodes :

- Protected Mode: Legacy protected mode supports 16-bit and 32-bit programs with memory segmentation, optional paging, and privilege-checking. Programs in this mode can access up to 4GB of memory space.
- Virtual-8086 Mode: Virtual-8086 mode supports 16-bit real-mode programs running as tasks under protected mode. It uses a simple form of memory segmentation, optional paging, and limited protection-checking. Programs in virtual-8086 mode can access up to 1MB of memory space.
- Real Mode: Real mode supports 16-bit programs using register-based memory segmentation. It does not support paging or protection-checking. Programs running in real mode can access up to 1MB of memory space.

2.3 MEMORY ORGANIZATION

The AMD64 architecture organizes memory into virtual memory and physical memory. Virtual memory and physical-memory spaces are usually different in size with virtual address space being much larger than physical-address memory. System software relocates applications and data between physical memory and the system hard disk to make it appear that much more memory is available than really exist and then uses the hardware memory-management mechanisms to map the larger virtual-address space into the smaller physical-address space.

2.3.1 VIRTUAL MEMORY

Virtual memory consists of the entire address space available. It is a large linear address space that is translated to a smaller physical address space. Programs use virtual address space to access locations within the virtual memory space. System software is responsible for managing the relocation of applications and data in virtual memory space using segment-memory management. System software is also responsible for mapping virtual memory to physical memory through the use of page translation.

2. AMD64 ARCHITECTURE OVERVIEW

The architecture supports different virtual-memory sizes using the following modes:

- Protected Mode: Supports 4 gigabytes of virtual-address space using 32-bit virtual addresses.
- Long Mode: Supports 16 exabytes of virtual-address space using 64-bit virtual addresses.

2.3.2 PHYSICAL MEMORY

Physical addresses are used to directly access main memory. This is the installed memory in a particular system that can be physically accessed by the bus interfaces. The larger virtual address space is translated to smaller physical address space through two translation stages called segmentation and paging. The architecture supports different physical-memory sizes using the following modes:

- Real Mode- Supports 1 Megabyte of physical-address space using 20-bit physical addresses.
- Legacy Protected Mode- Supports several different address-space sizes, depending on the translation. supports 4 gigabytes of physical address space using 32-bit physical addresses and when the physical-address size extensions are enabled, the page-translation mechanism can be extended to support 52-bit physical addresses.
- Long Mode- Supports up to 4 petabytes of physical-address space using 52-bit physical addresses. Long mode requires the use of page-translation and the physical-address size extensions (PAE).

2.4 MEMORY MANAGEMENT

Memory management refers to the process involved in translating address generated by software to physical address through segmentation and paging. This process is hidden from application software and is handled by system software and processor hardware.

2.4.1 SEGMENTATION

Segmentation mainly helps system software to isolate software processes (tasks) and the data used by that process to increase the reliability of system running multiple process simultaneously. The AMD64 architecture is designed to support all forms of legacy segmentation. In 64-bit mode segmentation is not adopted (use flat band segmentation) [AMD64]. Segmentation is, however, used in compatibility mode and legacy mode. Figure 2.2 shows segmented virtual memory.

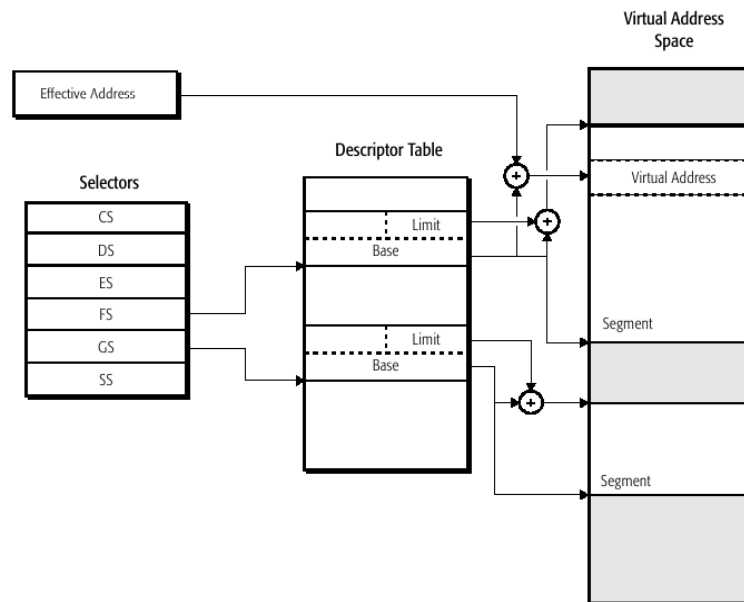


Figure 2.2: Segmented Memory Model

The segmentation mechanism provides ten segment registers, each of which defines a single segment. Six of these registers (CS, DS, ES, FS, GS, and SS) define user segments. The remaining four segment registers (GDT, LDT, IDT, and TR) define system segments. The segment selector points toward a specific entry in descriptor table. This can be Global Descriptor Table (GDT) or Local Descriptor Table (LDT). The descriptor table entry base value plus the effective address which is the offset from base gives the virtual address. Effective address is calculated from the value stored in general purpose register and a displacement value encoded as part of instruction.

2.4.2 PAGING

Paging allows software and data to be relocated in physical address space using fixed-size blocks called physical pages. It translation uses a hierarchical data structure called a page-translation table to translate virtual pages into physical pages. Paging also provides protection as access to physical pages by lesser-privileged software can be restricted. Figure 2.3 shows an example of paged memory with three levels in the translation-table hierarchy.

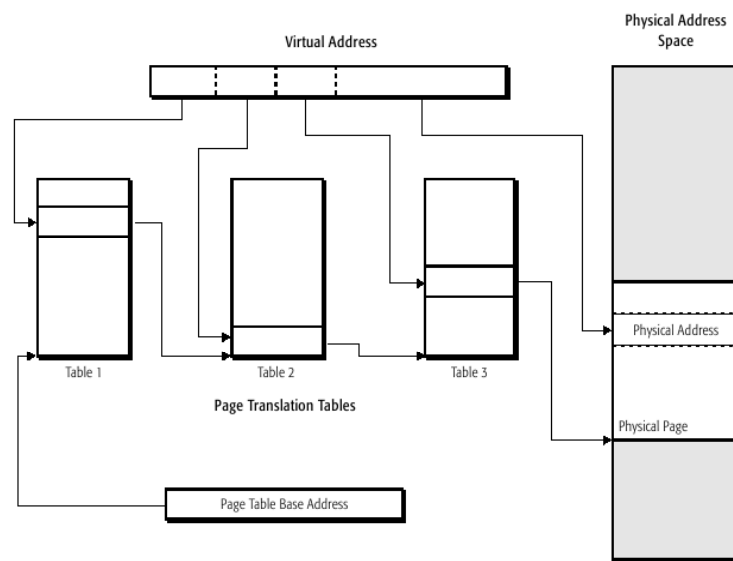


Figure 2.3: Paged Memory Model

The number of levels in the translation-table hierarchy can be as few as one or as many as four, depending on the physical-page size and processor operating mode. Each table in the translation hierarchy is indexed by a portion of the virtual-address bits. The entry referenced by the table index contains a pointer to the base address of the next-lower level table in the translation hierarchy. Last page table entry plus the offset value from the virtual address (lsb bits), points toward the actual physical address.

Chapter 3

VERIFICATION ENVIRONMENT

3.1 FUNCTIONAL VERIFICATION

In SoC design methodology, the first step is to define the specifications. Once the system specifications are completed, design phase starts. The behavioral modeling of the design is done using hardware description languages like VHDL or Verilog HDL and in this stage the design is said to be Register Transfer Level (RTL). Such design could be partitioned to aid reusability, concurrent development and tool effectiveness. In general, reusable components of designs are packaged into components called Intellectual Property (IP). The RTL description of design is verified against functional specifications. The system level verification is done to verify the RTL description against the intended functionality among other requirements such as timing, power and gate-count.

Functional verification validates that the design meets the requirements. Test cases are created based on specifications. Various aspects of data and control flow are verified by passing information between external environments, communication with I/O devices, software interactions etc. [ieee]

3.2 VERIFICATION

Most SoC verification concentrates on verifying the processor cores and their interactions with SoC level IPs. At this level of abstraction, verification of interaction between

3. VERIFICATION ENVIRONMENT

the IPs and functional verification of top level modules are done. Test conditions are written in x-86 assembly and in some cases written in high level languages like C++. The intension of each test is to verify specific functionality of the design and ensure its validity. The test plans must to be in sync with the specifications of RTL design and are to be updated with new specification changes to ensure that it is efficient enough to deal with all possible corner cases and boundary conditions.

Tests are developed, so that they stimulate the design in a specific manner and compare outcome against expected outcome to assert accuracy of design behaviour. Ideally the design should be verified against all possible scenarios that could arise and once it passes all tests, it can be considered as completely verified. In case when a particular test fails, the verification engineer needs to find out the cause of failure with his understanding of design or verification aspect that led to the particular failure. This process of root causing a failure is called as a “*debug*” in verification. Once root-caused, he then has to suggest appropriate changes to either design, verification or documentation to keep them in unison.

There are many possible issues that can lead to a test failure. Each test defines conditions for pass and fail. A fail or pass is basically the outcome of a test run. There can be many different causes for failures, with majority of them being

Self check fails In a self check failure, the program code running on the simulated processor is able to identify and report a failure. The program tests are written such that they evaluate the results, compare it with desired value and finally report fail or pass.

Assertion/Checker fails Assertion or checker fails are very common kind of failures and occur when a test-bench component reports an unexpected behaviour of either the design or a verification component. In general, most checkers or assertions monitor particular design states during the simulation and report failure whenever monitored value deviates from the expected value.

In general, a self check fail is caused when program execution deviates from the intended execution flow that was determined for the program under test. Hence the

debug of a self check fail would require knowledge of the program under test and its intended execution flow. Without these knowledge, it would be a challenge to debug such fails. This project is aimed to improve debug of such self check fails.

3.2.1 SELF CHECK FAILURE

Processor tests are C/assembly program written to assert that the processor under test is indeed functioning as expected under that specific setup. These processor tests are designed such a way that they are capable of deciding if the processor execution results were successful. These tests are normally hand written by the verification engineer rather than randomly generated. Hence such tests can string together specific stimulus of interest and determine pass or fail status on its own without relying on other verification components. Such tests are called as “self tests”.

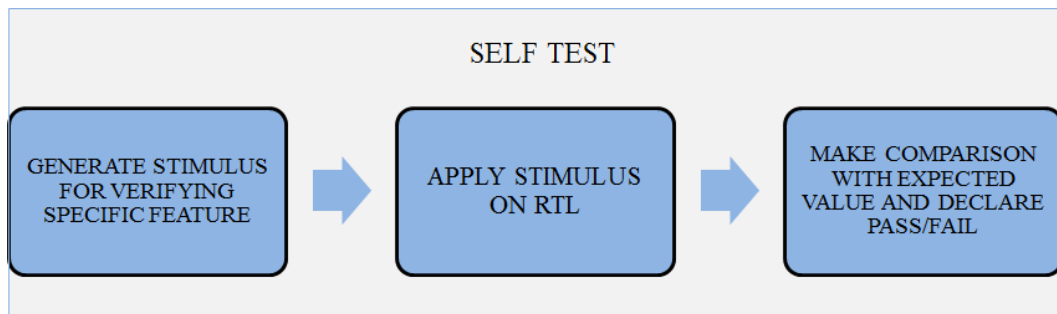


Figure 3.1: Self-Test

Figure3.1 details the flow of a self test. self tests are a collection of x86 assembly language program. Tests are compiled by a script which calls an assembler followed by a linker. The final output being a linked binary image can be loaded and executed by the RTL model of DUT . Stimulus is generated by the test and is applied to the DUT. Comparisons are done by the test itself after which pass/fail result is generated.

3.3 DEBUGGING A SELF TEST FAILURE

Self-tests report the occurrence of test case failure. Once this is available, next step is to analyze the reason for failure. For this, a traceback from the point of failure to the

point of error is required.

A failure message indicates that the result is deviating from the desired value. This desired value can be understood from analysing the asm test code. But to understand at which point during execution the design deviated from the desired course, detailed information regarding execution flow as well as a reference flow which has the ideal values and status are required. In general, a reference flow could be established by the engineer after understanding and interpreting the test in its completeness.

To aid execution flow, RTL is simulated along with an instruction level reference simulator. The reference simulator is a software model which imitates the design functionally and executes same instructions in parallel with the RTL. This parallel simulation is also called as co-simulation and produces a log of processor activity.

3.3.1 CO-SIMULATION

The instruction level reference simulator (ILS) is an x86/x86-64 compatible model, generally written in software languages such as *c++*. It models the processor in great detail including registers, caches and modes of operation. The test provides stimulus to both the RTL and reference models. An interface between RTL and reference model compares the states after each instruction retire and report any mismatch.

The following section details the features and functions of simulator and interface.

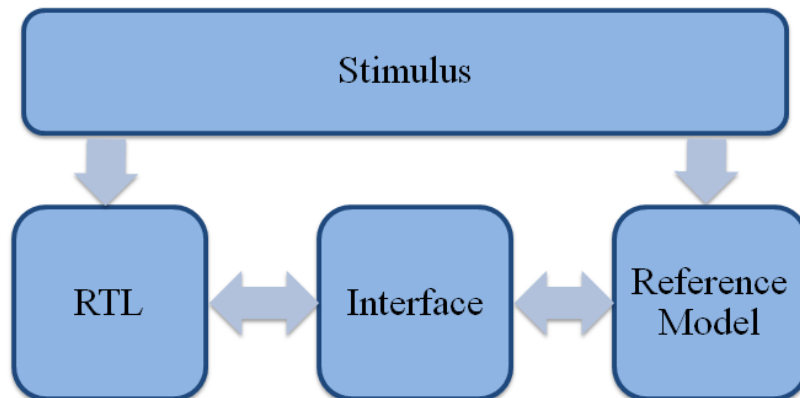


Figure 3.2: RTL-Reference Model Cosimulation

INSTRUCTION LEVEL SIMULATOR

The x86 instruction level simulator starts simulation after initializing the contents of its memory with linked binary image of the test. The ILS emulates the fetch/decode/execute algorithm of a scalar processor, producing an output log known as processor execution log. Each log entry describes the instruction, its results and any side effects it had on the processor state. The simulator models debug features, exceptions and interrupts as well as processor specific features. Supported processor states include x86 general purpose registers, flag registers, control registers, media registers, model specific registers as well as memory and I/O spaces. The simulator runs multi-threaded code to simulate multi-processor and multi-core processor systems. The simulator runs in step with the RTL. Whenever an instruction or exception is retired in RTL that thread within the simulator is stepped-up and the processor states in the RTL and the simulator are compared. Difference detected in processor states are considered as a mismatch; difference in memory locations are also considered as mismatches and upon any mismatch, the co-simulation is terminated. At the end of the simulation when all threads stop executing, the memory states of RTL and the simulator are compared and any discrepancies are reported as memory mismatches.

INTERFACE

The interface between RTL and reference model keeps the ILS in step with RTL signal. It steps the ILS when an instruction in RTL simulation has retired and then compares the results of execution between RTL and ILS. If the reference model is unable to model anything that is present in the RTL, the interface also re-synchronizes the ILS with the state obtained after execution of the RTL instruction. Main functions of this interface includes

Initialization Initialize memory model, attach RTL signal and initialize some specific RTL signals.

Increment Based on number of instruction retired per cycle, the interface informs ILS how many instructions to step.

3. VERIFICATION ENVIRONMENT

Interrupt Handling Interface informs ILS about pending interrupts

Comparing Compares RTL and reference model registers; integer, FP, control and status word. Also reports any mismatches.

Interfacing with the memory model Tell memory model what operations are seen by the RTL.

During the course of simulation, the ILS generates log files holding processor's program execution details. These are called "*processor execution log*"s. These log files will contain cycle by cycle information regarding register states, memory values, flags, threads etc. Basically all the comparison and debugging will require these information.

Once a failure is reported, simulation terminates and it would be required to be debugged. For root-causing the failure, verification engineer needs to trace through this processor execution log. Mismatches with reference model values provide information regarding the cause of failure.

Tracing through the log files is done manually. Data obtained from instruction simulation log and the assembled test files needs to be analyzed and co-related for debugging the failure. This is a very tedious effort consuming a lot of verification time. This is because of two main reasons:

1. Relevant/required information is buried under a wealth of information
2. Co-related information is spread across in different files

As the design itself is very complex, these reasons makes manual tracing too time consuming. This will stretch the verification time and ultimately time-to-market.

Chapter 4

PROPOSED ENHANCEMENT

4.1 PROCESSOR EXECUTION DETAILS

During simulation of a processor core, the linked program image is loaded in processor's memory followed by its execution. Most modern microprocessors adopt instruction level parallelism for high throughput. Micro-architectural features like instruction pipeline, superscalar execution, register renaming, speculative execution, branch prediction etc. are employed in order to exploit instruction level parallelism. These micro architecture features work together for high execution throughput. All these internal operations results in a complex execution flow with multiple operations happening in each cycle with different levels of dependencies between data, instruction and memory.

As explained in Section 3.3.1 execution log file captures important information regarding the processor state and activity during execution of program code. This could be considered as history of events in the simulation. Each entry in the log file contains the following information regarding processor execution:

- The instruction number, simulation cycle and opcode
- Thread Id (relevant in multi-core and multi-processor)
- Memory read/write information
- Code read/write

- I/O read or write
- Interrupt and exception information
- Branch Target
- Paging info
- Flag values
- Register affected on that instruction execution

On the onset of simulation failure, these information are vital in debugging the cause of a failure. The following case study of two test scenarios affirm this fact.

4.2 SAMPLE DEBUG CASES IN CONSIDERATION

Let us consider two simple assembly tests as case studies to demonstrate the usefulness of processor execution log during debug of a self test failure.

4.2.1 TEST A

Consider an assembly test in Algorithm 1 that intends verification of a memory module. The test writes a value into a memory location (Line 3). The data is later read from the same location (Line 5) and compared with the original value written into the location (Line 6). The test flags fail or pass based on the comparison.

The test verifies a single write/read from a memory location. Ideally the values written to the memory location should match the value read from the memory and test completes with a pass. Now consider a case where the comparison fails. This could occur due to many reasons. Following are a few scenarios which can lead to a failure:

Case 1 : If from another thread of program execution, the control bit for bank selection is changed during the execution, the data read will be from wrong memory location, leading to a failure.

Input : *data, address*

Output: Test result: *pass* or *fail*

```

1 Initialization: Select memory bank by setting ControlRegister
2 RegA  $\leftarrow$  data
3 RegB  $\leftarrow$  address
4 Memory [RegB]  $\leftarrow$  RegA
5 RegD  $\leftarrow$  Memory [RegB]
6 if RegA == RegD then
7   |   report pass
8 else
9   |   report fail
10 end

```

Algorithm 1: Memory Read-Write

Case 2 : If the address value is invalid. This can happen when the test generates a random address value for storing the data and this value might not exist in the current selected memory bank range.

Case 3 : If the register *RegD* chosen is unavailable in this mode of processor operation. This will cause the wrong data to be updated into the memory and comparison fails.

Debug Process

In case 1 10, the memory bank selected should stay same throughout the program execution. Any change in memory bank selects will cause the read operation to take value from wrong memory block, leading to a self-test failure. In case 2 10, given that each memory block has a fixed size, if the generated random address generated is not in valid range of offset, then this error could occur. In case 3 10, if care is not taken in choosing processor mode prior to the test execution, and if *RegD* is either completely or partly unavailable (like 32 bit mode Vs 16 bit mode), then this error could occur.

Figure4.1 depicts a typical debug process. Note that there could be other causes

4. PROPOSED ENHANCEMENT

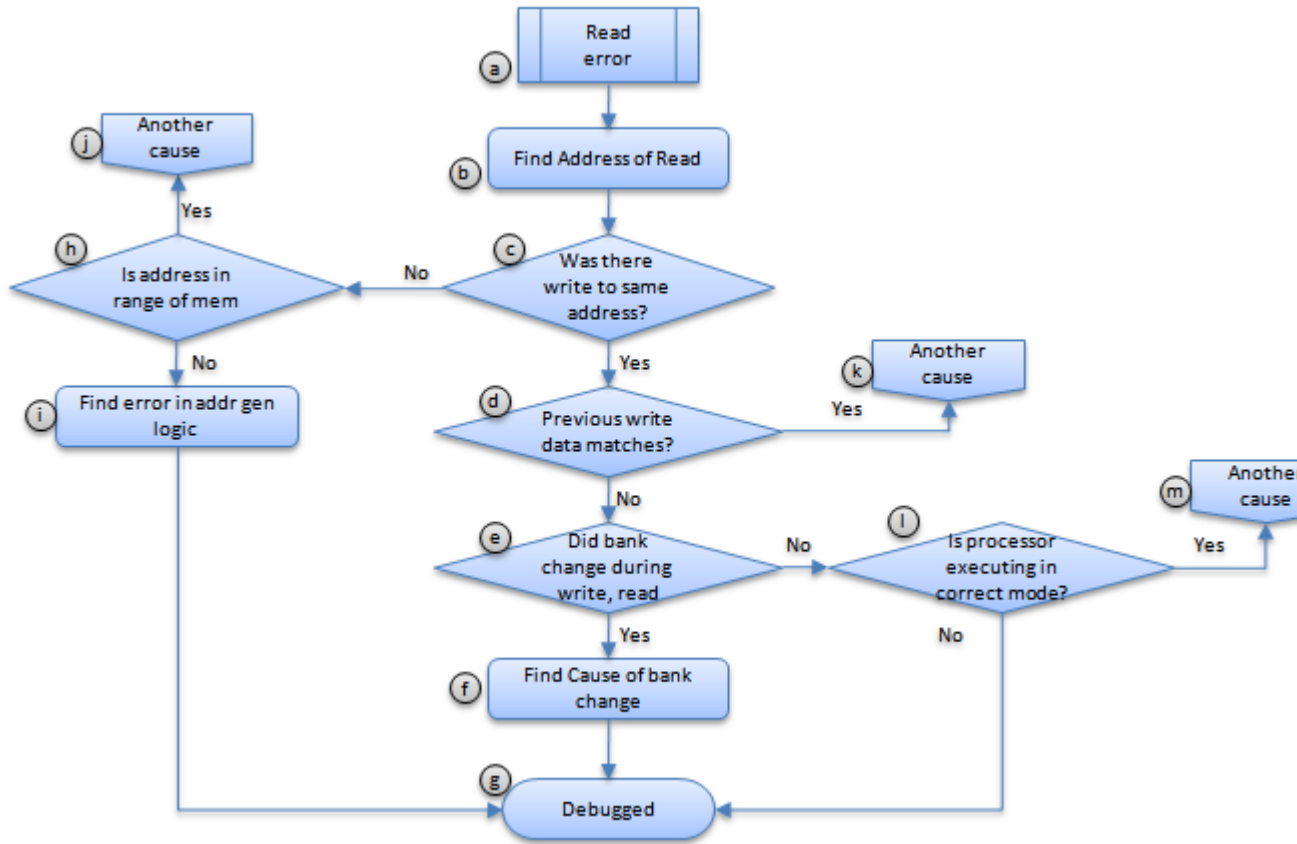


Figure 4.1: Illustration of debug process : Test A

for failure as depicted in states ①, ⑫, & ⑭. During debug, the engineer needs access to certain information as listed here

1. state ⑥: Examine Read to address, examine data obtained.
2. state ③: Obtain all accesses occurred to same address, find the latest write to same address.
3. state ④: Examine data written to latest write to same address.
4. state ⑤: What is the bank select location, did it change between write and read.
5. state ①: Which program code changed bank select?
6. state ①: What is processor mode?
7. state ①: What was the random address that got generated?

4.2.2 TEST B

Consider a string *tolower()* program. The program reads an input string in upper case and converts it to lower case. Considering ASCII character encoding, conversion of upper case to lower case could be accomplished by addition of constant to the ASCII encoding.

Input : *string*

Output: *string*

```

1 for every character c in string do
2   |
3   |  $c \leftarrow c + CONST$ 
4 end
5
```

Algorithm 2: String Lower Case Conversion

The program in Algorithm 2 converts each character from upper case to lower case in a loop, to achieve the result. This program can fail due to many reasons. Let us consider a couple of scenarios.

Case 1 : Consider that the loop variable is not initialised incorrectly. Loop variables are used as the index variable to select every character of the string. This will lead to an invalid conversion.

Case 2 : If loop exit condition is wrong, then the loop could terminate early.

Debug Process

In case 1 5, initialisation of loop variable is the cause and could be found by looking at value of the variable just after loop is entered. If after the completion of loop execution for the first time, if the first character remains un-modified, then that also suggests an initialisation problem. In case 2 5, the resulting string after loop exit would normally have characters unmodified towards the end. This case could be identified with such symptom.

4. PROPOSED ENHANCEMENT

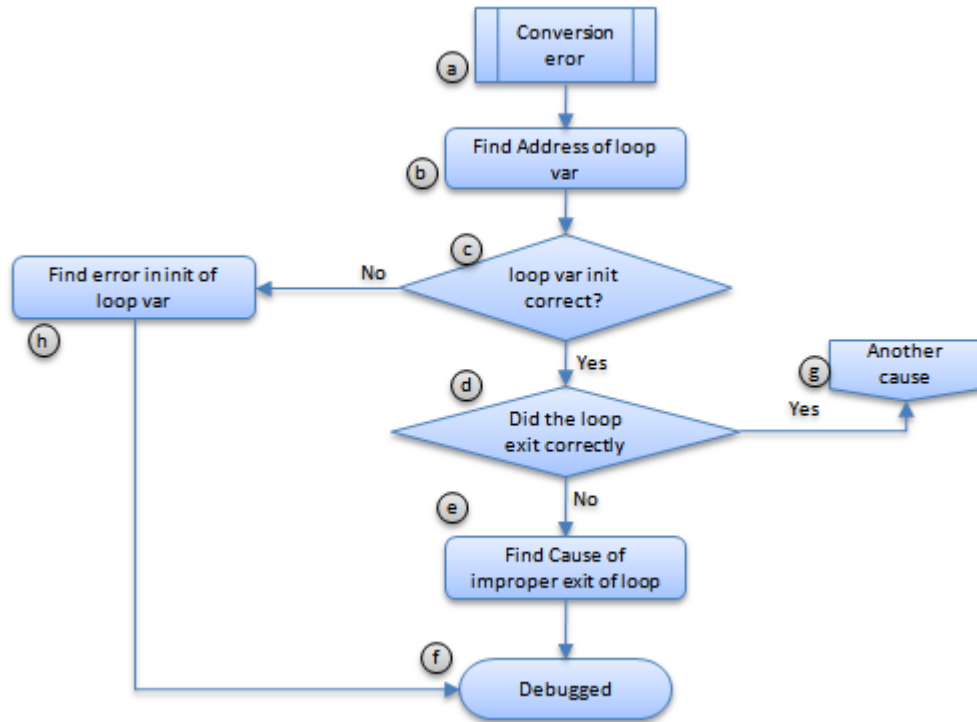


Figure 4.2: Illustration of debug process : Test B

Figure 4.2 depicts a typical debug process for this case. Note that there could be other causes for failure as depicted in state (g). During debug, the engineer needs access to certain information as listed here

1. state (b): Find address of loop var.
2. state (c): At start of loop, check initial value of loop var.
3. state (d): When the loop exited, what was the value of loop var.
4. state (e): Need computation values of loop exit condition.
5. state (f): Need computation values of loop init condition.

4.3 NECESSITY TO ENHANCE MANUAL VERIFICATION

Debug engineer has to extract required information whenever needed, manually from assembly files, linked object files and processor execution logs. Correlation of information between different files is also required to make a successful debug. Such manual extraction of information and correlation of it, though may seem obvious consumes considerable debug time and is error prone. Hence an enhancement to such manual process, if available, would considerably decrease the time required to debug an issue.

Information exists in processor execution log, assembly file and list file. Correlation of list file and execution log file can be accomplished by correlating address. For each cycle the Instruction pointer register (RIP) hold the current and next instruction address. The verification engineer has to search through the files for address values to correlate cycles in log file with lines in list file. As observed in Section 4.2, data extraction and correlation from different files are required at different stages of debug to effectively debug how and why the test failed.

4.4 ENHANCING MANUAL VERIFICATION

After analysing debug process, it is very evident that availability of correlated information while during debug process would enhance it in a big way. In this thesis, we are proposing a Graphical User Interface (GUI) which will help in navigating through the log files quickly, aiding in comparison with list file and also some additional features to help in faster analysis of failure cause. The interface will help to get rid of traditional method of comparing Instruction Pointer Register (RIP) values and string search by providing graphs that will connect each cycle in log file with corresponding asm line code. The proposed interface enhances the data navigation through log files and failure analysis. It is possible to lay down the requirements of such an enhancement

Visualisation of Execution Flow Visual representation of data is always very appealing, than without a visual representations. In our case, if there exists a visual

4. PROPOSED ENHANCEMENT

representation of execution control flow, it would be obvious if a loop had executed and if so, how many times it was.

Navigating Execution Flow In the visual representation of execution flow, it would also aid debug when user is able to navigate to point of interest by mouse gestures on such representation. For eg. if program code exists across different source files, the user would be able to navigate to the point of interest with ease.

Processor State Information At any execution point, it would help the debug engineer if processor state information such as SP, PC, SR are readily accessible.

Processor State Information Change It would aid debug to be able to list the differences in Processor State Information between two arbitrary points of execution.

Processor Writes and Reads It would aid debug if processor writes and reads could be easily listed. It would also help if in such listing one could classify between IO accesses and memory accesses.

Code listing It would aid debug when source code is listed along with its context, on any point of execution.

Chapter 5

INTERFACE IMPLEMENTATION

5.1 INTERFACE

The necessity to enhance manual debug methodology has been established in Section 4.3 and certain requirements of such an interface is established in Section 4.4. This chapter concentrates on design and implementation of the debug interface.

The debug interface is aimed at being intuitive and user friendly. It should aid data navigation so as to reduce manual efforts. If user requires to refer to actual processor execution log or assembly code, it should be readily available through the interface. The interface should also provide graphical representation of processor execution log, to aid traversal and filtering of processor activity. The interface should aid traversing through processor execution easy. Critical events made by processor should be extracted, categorized and made available to user.

In addition to interface requirements listed in Section 4.4, the following set of features would also aid in debug:

- Visualizing thread-wise execution flow
- Ability to highlight instances of critical activities such as Memory write/read, I/O write/read, Branching etc
- Interrupt and exception happening during execution
- Assembly code traversal and its linkage with execution flow

5. INTERFACE IMPLEMENTATION

- Register current state value traversal and its linkage with execution flow
- Comparison of register values between arbitrary instances of execution
- Simulation cycle of execution event

Every stimulus is a unique assembly test and hence the debug interface should be as generic as possible. It should be able to accomodate any relevant assembly test, accompanied with execution logs.

5.1.1 CHOOSING GUI

There are multitude of languages providing GUI capability, the project doesnot need a very sophisticated GUI implementation. Moreover the interface should aid remote debug and if possible thee should not be any requirement for any user to install a tool-kit or library for the debug interface to be used. Hence the decision to use web-browser as the debug interface was a default choice. A web browser could also execute *javascripts* and that makes it customisable. The design is also scalable and aids introduction of new features for debug with ease.

Each stimulus will have its own set of assembly test files and execution log files. The interfaces implementation starts by consuming these files. Two programming languages are used for implementation:

Python Script for data extraction and correlating related information.

JavaScript for designing the interface features and user interaction.

Figure5.1 depicts information extraction and use of different scripts to achieve the same. Major implementation steps

- Extraction of information from assembly list file and processor execution log by python script.
- Correlation of information from previous step.
- Coding generic JavaScript code for user interactions.

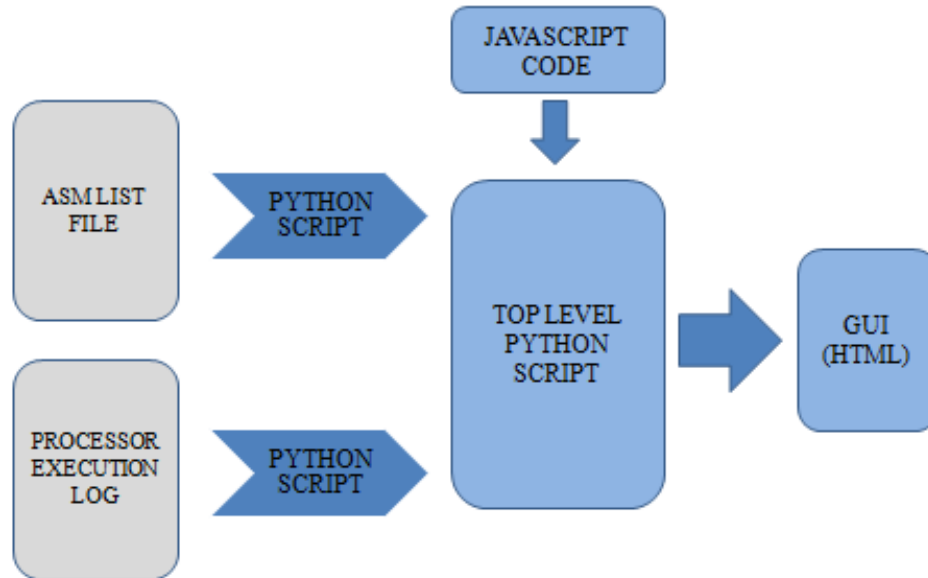


Figure 5.1: GUI Implementation

- Top level python program to manipulate available information and to convert it to JavaScript information.
- The top level python program will combine the data extracted and javascript code to generate a single HTML page.

The whole infrastructure is packaged neatly. Given execution log file and assembly list file, the top level python script generates the HTML interface web page. Different stages involved in this conversions are described in the following sections.

5.1.2 INTERFACE

GUI is targetted to run on any browser, hence layout design is done using *HTML*. However for providing interactive features to the user, a much more powerful language is need along with HTML and default choice is JavaScript.

JavaScript (JS) is an interpreted computer programming language. It is implemented as part of web browsers so that client-side scripts could react to user inputs,

5. INTERFACE IMPLEMENTATION

customize the browser, communicate asynchronously, and to filter contents being displayed. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. In addition a style sheet language called CSS is used for describing the presentation semantics (the look and formatting) of the interface page written in HTML.

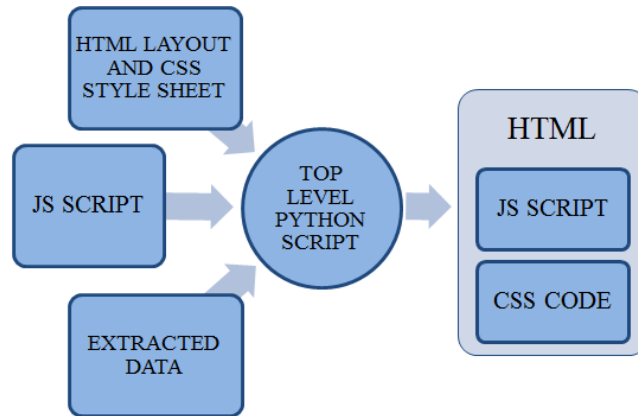


Figure 5.2: Web Page Generation

Resultant HTML page embeds JavaScript routines and style sheets. This is done by the top level python script. Figure?? shows a representation of information flow.

5.1.3 JS DATASTRUCTURE

All dynamic features of the interface are handled by JS script routines. Data input to the JS is the extracted information from execution log files and list files. The top level python script converts the extracted objects into JS data array “*dataArray[]*”; where each array element corresponds to an active thread. Active threads refer to active core or processor in a multiprocessor system.

Once the extracted information is converted to JS compatible data, the interface features can utilize this. The layout for various GUI features called windows, are designed using HTML and CSS code. All dynamic interactive features are handled by embedded JS script in HTML web page. Later sections introduces different windows available in the web page.

```

1 CreateDataArray()
2 begin
3   for i in activeThreads do
4     |
5       for log in logObj do
6         |
7           if log.Id == i then
8             | Append log → DataArray[i]
9           end
10        end
11    end
12 end

```

Algorithm 3: Creating JavaScript Object

5.2 EXTRACTION OF INFORMATION

The first step in creating a generic debug interface is to extract relevant information and convert it to a format that could be loaded by the debug interface.

5.2.1 STIMULUS

Stimulus is written in X-86 assembly language. The engineer is expected to debug this stimulus. Each cycle in processor execution log corresponds to a particular assembly code. An assembler is used to assemble to object code, in that process each instruction is also mapped to a particular address. Figure 5.3 shows this process which also produces a list file which hold an macro expanded, loop unrolled, assembled code with corresponding address details. The configuration file defines certain random operands, segmentation, gdt, ldt, page tables to aid in assembling process. The include files contribute common routines used across different assembly stimulus.

List file holds a wealth of details including instructions, opcodes, operand, linear address, module/register configuration details etc. A *python* script is used to extract

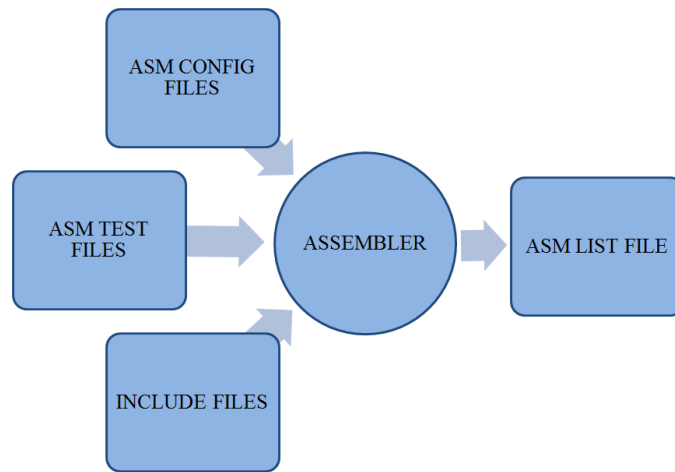


Figure 5.3: Assembler

relevant information corresponding to each instruction line in the list file. In this project, the following information is extracted for each instruction from the list file:

- Base and offset address value
- Instruction line number
- The assembly code

The *python* script internally contains a data structure of list objects, which helps in adding features to the script with ease (Figure5.4).

5.2.2 EXECUTION LOG

Execution log file contains instruction by instruction execution details from simulation. It also includes register states, thread details, flag values and other details which help in tracing out the cause of failure. As explained in previous chapters, debugging require detail traversal through this log file.

Details in the log file with reference to each cycle is required to be extracted. *Python* script is used to extract these informations as well and a comprehensive data structure is created. The data structure objects contain properties corresponding to major operations and register values. Thread details are handled as seperate data structures as it aids

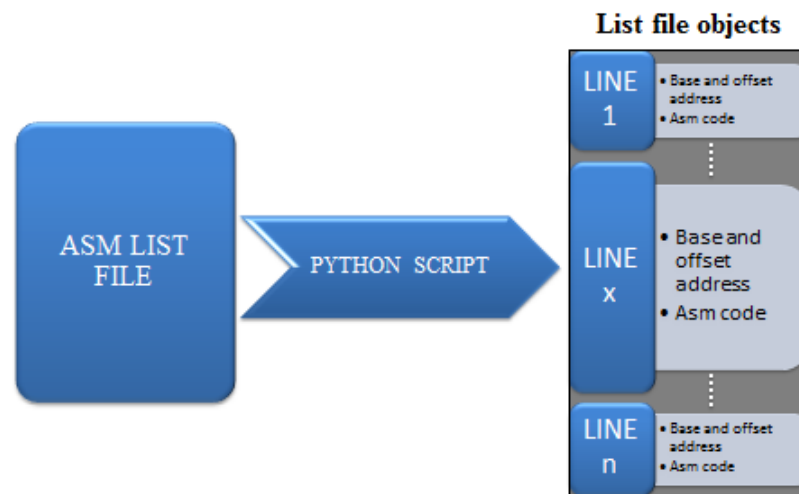


Figure 5.4: Asm List File Extraction

processing thread-wise informations. Following properties are extracted from execution log file for every execution event

- Thread number
- Mode of operation
- Cycle number
- Linear address
- Memory write, Memory read, I/O write/read, Code read
- Branch target (linear address)
- New values of registers, if modified

5.2.3 INFORMATION CORRELATION

Once both the input files are processed, next step is to correlate assembly with execution log. A *python* script can accomplish this by correlating respective data structures

5. INTERFACE IMPLEMENTATION

based on linear addresses. As x86 architecture follows segmented memory model along with paging, address translation is required for generating the linear/physical address⁵.

Linear address is calculated as

$$\text{Linear address} = \text{Base address} + \text{Offset value}$$

```
Input : list file objects  $\rightarrow$  listObj[], log file object  $\rightarrow$  logObj[]

1 Start:
2 for each object list in listObj do
3   |
4   | list.address = list.Base + list.Offset
5 end
6 for each object log in logObj do
7   |
8   | set count = 0
9   | for each object list in listObj do
10  | |
11  | | if log.address == list.address then
12  | | | Append each list.property  $\rightarrow$  log.property
13  | | | // log properties are lineNo, opcode and address
14  | | | count = 1; break loop
15  | | end
16  | end
17  | if count == 0 then
18  | | assert: "noaddressmatch"
19  | end
20 end
21 End
```

Algorithm 4: Combining List and Log File Information

For each object of execution log, the corresponding list file object could be found

based on linear address value. Once found, both are cross-linked to aid further processing. The algorithm employed in this search is depicted in Algorithm 4. Data extraction is complete with this linkage and the result is execution log objects linked to corresponding assembly objects.

5.3 GUI FEATURES

5.3.1 EXECUTION FLOW GRAPH

Main feature of debug interface is the graph showing the execution flow of the code. It is obtained by plotting asm list file line numbers against the execution cycle during which it is executed. All the active threads have different graphs which are tabbed. Hovering the mouse over any execution point on the graph will display X and Y axis values. Features include zoom-in, zoom-out of plot. Double click at any point is featured to zoom-out completely. On clicking on any execution point, selects the point and synchronises data across windows. Proximity click feature helps in automatically choosing nearest execution point for further analysis.

The flow graph also features customisable selection of specific processor operations out of Branching, Memory Write, Memory Read, Code Read. This helps in filtering redundant execution points those may not be of interest to the engineer. Each distinct processor operation could be distinguished by color coding.

IMPLEMENTATION

Execution flow graph is created with Dygraph JavaScript Visualization Library⁹. The library provides inbuilt functions that enable zooming, x-y axis value display and point “onClick” callbacks. *onClick CallBack()* function is called whenever an execution point on Dygraph is clicked. The function has been extended to activate other windows and to synchronize components across windows.

Graph could be built by providing independent axis values followed by depended axis values. For each thread, a different graph is generated but layered out on different

HTML tabs. The x-axis is cycle number and y-axis is the list file line number. Algorithm 5 shows the pseudo-code used in building the graph in our application.

```

1 CreateGraph()
2 begin
3   for each i in activeThread do
4     |
5     for each element in dataArray[i] do
6       |
7       Dygraph[i]  $\leftarrow$  [element.cycleNo, element.lineNo]
8     end
9   end
10 end

```

Algorithm 5: Creating Execution Graph

5.3.2 REGISTER WATCH WINDOW

Register window is used in displaying instantaneous register values at any execution point. Under the hood, thread based register values are maintained separately, as it aids user switching back and forth between threads of execution without loosing data on each thread. At any instant the register window holds the values of *selected* execution point. Selection could be changed by simply clicking on a different execution point. Register window provides values of following registers to the user:

- 64 bit general purpose registers (RAX, RBX, RCX etc)
- RFLAG (64 bit)
- Instruction Pointer (RIP)
- Stack Pointer (RSP)

Another important feature provided by register window is a comparison of register values between two different execution points. It highlights differences in register

values with different colours, to capture user's attention with ease. The diff is between *reference execution point* and *selected execution point*. *Reference execution point* is chosen by the *set marker* button.

IMPLEMENTATION

The values contained in register window is updated when a different execution point is selected in the execution flow graph. This event also triggers comparison of current values with the reference point. Algorithm 6 lists the pseudo-code used in updating register window values.

```

1 pointonClickListener(element)
2 begin
3   for each reg in RegisterSet do
4     regRow[reg] ← element.[reg]
5     if (element.[reg] ≠ referenceRow.[reg]) then
6       |
7       |   HighlightregRow
8     end
9   end
10 end

```

Algorithm 6: Creating Register Window

5.3.3 INSTRUCTION WINDOW

Instruction window lists the assembly code corresponding to the selected execution point. Context around the assembly code is also displayed to aid debug. This window is also updated when a different execution point is selected in execution graph. The window highlights assembly code for visual attention of the user.

IMPLEMENTATION

The pseudo-code used to update values in this window is shown in Algorithm 7.

```

1 onClickCallBack(element);
2 begin
3   for each item from elemnt-50 to element+50 do
4     |
5     |   Add item.opcode → InstructionWindow
6   end
7   Add item.logInfo → ExecutionLogWindow
8 end

```

Algorithm 7: Creating Instruction and Execution Log Window

5.3.4 EXECUTION LOG WINDOW

In addition to the instruction and register information, the relevant processor execution log in its actuality would be useful to the user as it contains different information that may not be presented by the debug interface. Having this information readily accessible to user also assures the user that the data extracted through processing by different scripts is indeed correct. Internally these informations are stored as a JavaScript object "*logInfo*".

5.3.5 MARKERS

As it was discussed in Section 5.3.2, markers are used to “mark” reference execution point through a “Set Marker” button. Button “Clear Marker” could be used to clear the marker.

IMPLEMENTATION

Set and Clear buttons are implemented using HTML form’s callback feature. A *JavaScript* variable is used to store the reference execution point.

Chapter 6

RESULTS: GRAPHIC USER INTERFACE

The final GUI is a HTML page generated by the master *python* script which derives informations from assembly list files and execution log files. The generated .html file can be viewed by any standard web browser (like Internet Explorer, Firefox, Chrome) those support JavaScript. Care is taken that this htm is devoid of dependencies and hence enables interactive debug from remote locations. The only requirement for the user is network connectivity to receive the html page. The following figures shows various windows of final GUI

6.1 EXECUTION FLOW GRAPH

Figure 6.1 shows the main execution flow graph for selected active thread.

Figure 6.2 highlights execution points in which memory writes and branch operation occurs. The appropriate checkboxes towards bottom indicates the selection made. Note that thread *P000* is the active tab and inactive tab for *P001* is layered below it and could be selected.

6. RESULTS: GRAPHIC USER INTERFACE

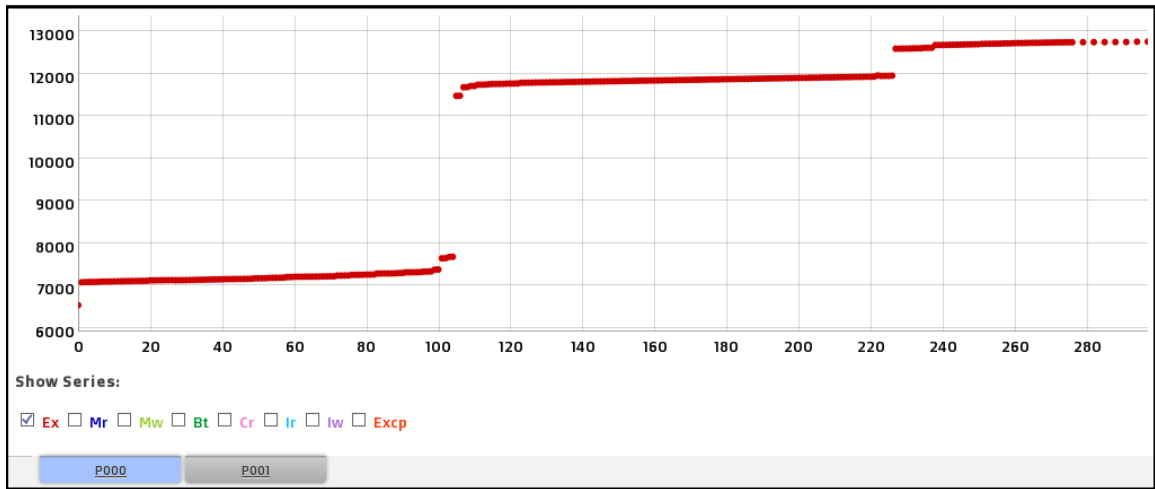


Figure 6.1: Execution Flow Graph

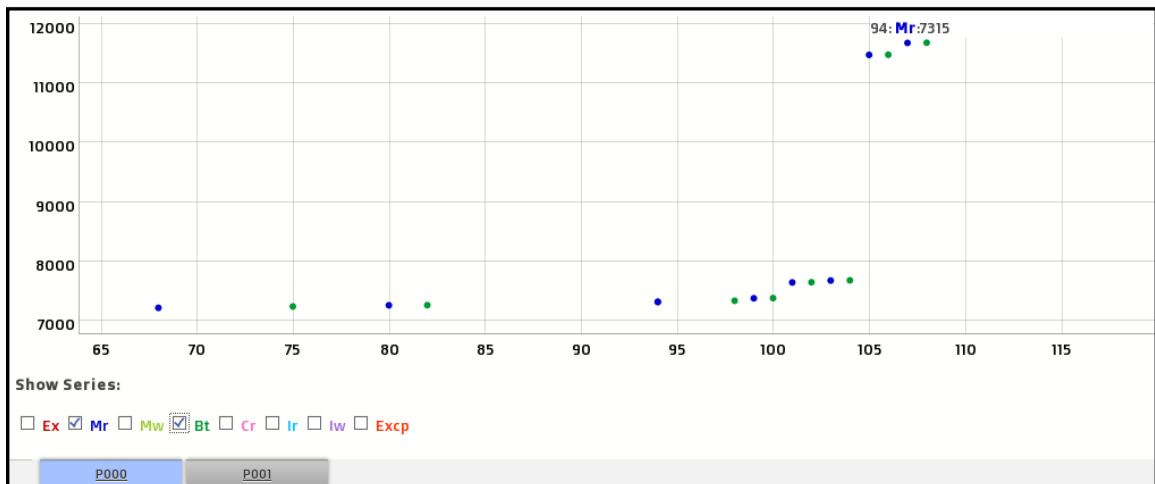
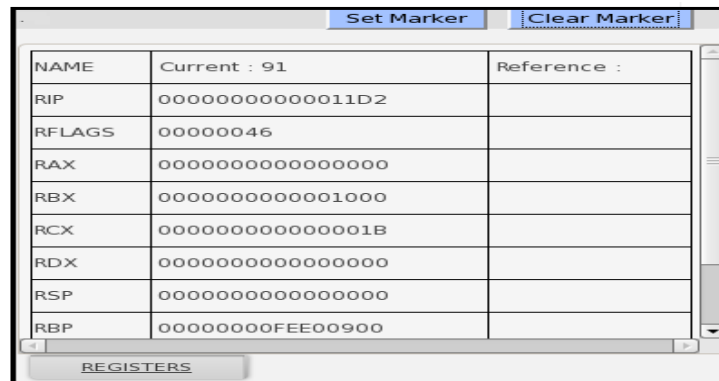


Figure 6.2: Execution Graph With Branching and Memory Writes

6.2 REGISTER WATCH WINDOW

Figure 6.3 is the register window. It could be observed that register values corresponding to selected execution point *cycle 91* is shown in *current selection* column. *Set Marker* and *Clear Marker* buttons could also be observed in the picture. Figure 6.4 shows the comparison of register states at two different points. Differences are highlighted.

6. RESULTS: GRAPHIC USER INTERFACE

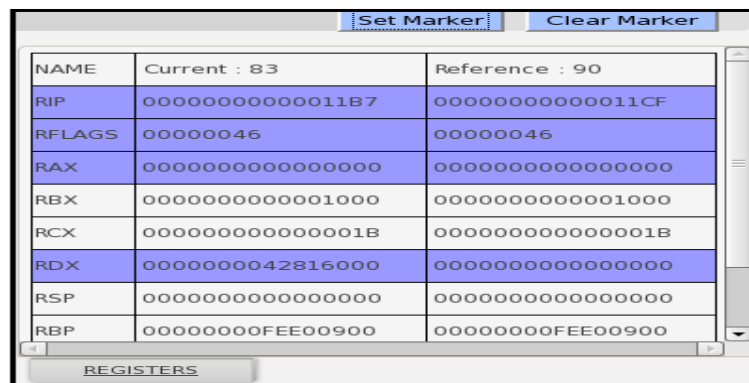


The Register Watch Window displays a table of CPU registers. At the top, there are two buttons: 'Set Marker' and 'Clear Marker'. The table has three columns: 'NAME', 'Current', and 'Reference'. The 'Current' column shows the current value of each register. The 'Reference' column is currently empty. The registers listed are RIP, RFLAGS, RAX, RBX, RCX, RDX, RSP, and RBP.

NAME	Current : 91	Reference :
RIP	00000000000011D2	
RFLAGS	00000046	
RAX	0000000000000000	
RBX	0000000000001000	
RCX	000000000000001B	
RDX	0000000000000000	
RSP	0000000000000000	
RBP	00000000FEE00900	

REGISTERS

Figure 6.3: Register Watch Window



The Register Watch Window displays a table of CPU registers. At the top, there are two buttons: 'Set Marker' and 'Clear Marker'. The table has three columns: 'NAME', 'Current', and 'Reference'. The 'Current' column shows the current value of each register. The 'Reference' column shows the reference value of each register. The registers listed are RIP, RFLAGS, RAX, RBX, RCX, RDX, RSP, and RBP.

NAME	Current : 83	Reference : 90
RIP	00000000000011B7	00000000000011CF
RFLAGS	00000046	00000046
RAX	0000000000000000	0000000000000000
RBX	0000000000001000	0000000000001000
RCX	000000000000001B	000000000000001B
RDX	0000000042816000	0000000000000000
RSP	0000000000000000	0000000000000000
RBP	00000000FEE00900	00000000FEE00900

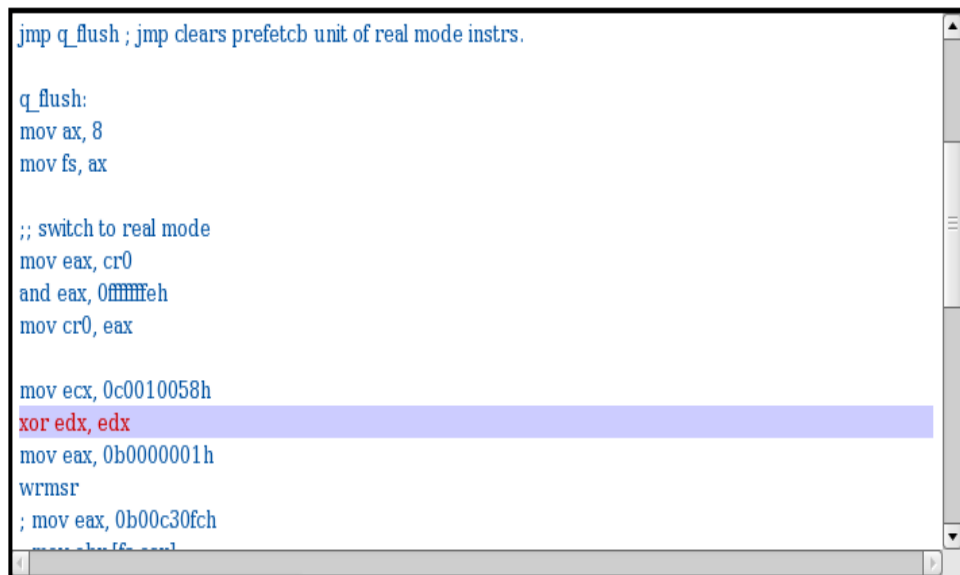
REGISTERS

Figure 6.4: Register Value Comparison

6.3 INSTRUCTION WINDOW

Figure 6.5 is the instruction window. The instruction window could be seen highlighting the assembly along with its context, corresponding to the selected execution point.

6. RESULTS: GRAPHIC USER INTERFACE



```
jmp q_flush ; jmp clears prefetch unit of real mode instrs.  
  
q_flush:  
mov ax, 8  
mov fs, ax  
  
;; switch to real mode  
mov eax, cr0  
and eax, 0xfffffeh  
mov cr0, eax  
  
mov ecx, 0c0010058h  
xor edx, edx  
mov eax, 0b0000001h  
wrmsr  
; mov eax, 0b00c30fch  
mov ebx, fs:eax
```

Figure 6.5: Instruction Window

Chapter 7

CONCLUSION

An interactive off-line graphical user interface was designed and developed. The interface is completely web based and achieves primary goals of off-line debug. The interface contains all relevant information from processor execution log and assembly list files commonly needed for debug. Further debug if required, could be accomplished by traditional methods.

Execution graph helps in analysis of various operations like branching, exception, read/write etc and also to navigate through processor execution with ease. The graph presents thread-wise informations seperately, making tracing and navigation easy. The register window provides comparison between registers and flag values at two different execution cycle which helps in tracing a wrong value written into register or flag. Instruction window and execution log window provide common information necessary for debug.

Potential for future work: The project demonstrates that web-based technologies like *html*, *JavaScript*, *CSS* are effective in enabling interactive remote debug. There is larger scope for this work in future as it gets adopted by different teams. New debug features like break points, cache containers, address filters among others could be needed in the future. Care has been taken to preserve design modularity, so that such fetures could be added in the future.

Part II

Chapter 8

INTRODUCTION

Functional verification of a design is the task of verifying that the logic design conforms to specification and ensures functional correctness of the design. With the rapid increase in design complexity and size, this phase of circuit design has become the most crucial and resource-intensive. It is widely accepted that more than 70 percent of design efforts is spend on verification and with the advancement in silicon technology and adoption of complex SoC designs; this situation is only projected to get worse in future.

Verification is done at different abstraction levels. Two main abstraction levels from verification point of view are RTL and gate level. RTL enables relatively easier management of complex designs and has less verification time requirements compared to gate or netlist level. However it cannot eliminate the need for verification at netlist level as each level is required for specific type of verification. While RTL verification is apt for functional validation or architectural analysis, more detailed analysis like timing, power etc require detailed models of lower levels. Another important challenge is ensuring the functional equality of models at different abstraction level.

Traditionally simulation based verification has be used as the primary approach for verification at both RTL as well as gate level. But with very complex designs, this approach has become inefficient in finding subtle design bugs. Also at gate level, simulation based verification takes rather too much time that an exhaustive verification is impossible. On the other hand, formal verification tools have gained popularity since it can mathematically prove or disprove the design validity. These mathematical meth-

ods require less manual effort than simulation based verification and hence a lot faster. However these mathematical models cannot comprehend all kind of complexities that could occur in the design and it is very much clear that this alone can solve all issues. Rather a mix of simulation and formal verification methods are practiced to ensure all corner cases are covered.

8.1 GATE LEVEL SIMULATION

Gate Level Simulation (also called as gatesims) still play crucial role in verification, in spite of advancement in formal verification techniques like LEC and STA . When having to verify with gatesim one has to start planning early, as it has to pass through various stages before sign off. The setup for gate level simulation starts right after the prelim netlist is available and will continue till final post layout netlist.

Even though gatesims help in finding many issues related to timing and power, it is considered as a “*necessary evil*” by engineers. This is mainly because gate level simulation is inherently very slow. It is also hard to find the optimal list of test cases to effectively utilize gatesims. Debugging is also very tedious process at gate level combined with the long run time makes turn-around for debug long hence ultimately affects time-to-market of the product.

Various approaches have been adopted over the years for gate level simulation with each method trying to improve simulation performance and ease of debug. Since generally netlists are Verilog based, test environments used for RTL verification could be reused for netlist verification by replacing RTL with netlist as appropriate. It should also be possible to have the test vectors generated for RTL simulation used as stimulus for netlist simulation. Such application of testvectors can be done either by having the RTL be simulated in parallel with gatesim or by applying captured the test vectors from RTL simulation onto netlist simulation. The first method of stimulus application is called a co-simulation approach and the second is called a dual-simulation or sim after sim approach. Each method has its own set of pros and cons. Simulation time, design complexity, memory, ease of debug, testbench complexity tradeoffs are considered while choosing any particular method for gatesims in a project. This thesis analyzes the

advantage and disadvantage of past and present gatesim methodologies and proposes a new improved sim after sim or dual simulation approach to gatesims.

8.2 ORGANIZATION OF THE THESIS

The organization of this project report is as follows:

Chapter 9 -*Gate Level Simulation* explains the relevance, advantages and limitations of gate level simulations.

Chapter 10 -*Gatesim Methodologies* briefly explains various approaches used for gatesims, their advantages and disadvantages.

Chapter 11 -*Improved Dual-Sim Approach To Gatesim* describes the implementation and flow of proposed dual sim approach .

Chapter 12 -*Results* gives comparison of simulation performances and memory utilization of current and proposed approaches.

Chapter 9

GATE LEVEL SIMULATION

In a typical VLSI design flow for verification, the first step after RTL level model of the design availability is writing behavioral test bench for functional verification. The functionally verified RTL goes through design synthesis during which it is mapped into low level design components in terms of primitives or logic gates. Synthesis is mostly an automated process using a “*synthesizer*” tool that converts RTL-level design source code into corresponding gate-level netlist mappings. This netlist is also called the pre-layout netlist.

The pre-layout netlist that was obtained from synthesis is then fed into a layout tool which maps the gate primitives to silicon structures such as channels, gates, vias, etc. During this process certain modifications are done on netlist but it should not alter its functionality to its corresponding RTL. To validate this, another netlist called the post-layout netlist is generated back from the laid-out silicon structures by the layout tool itself. Validation is made by running LEC tool over both pre-layout and post-layout netlists or between post-layout netlist and RTL.

Though it would be ideal to use post-layout netlist for the purpose of gatesims, it would be too late in the design process. So work on gatesims starts with pre-layout netlist and progresses to post-layout netlist as it becomes available. Figure 9.1 depicts progress of gatesim with respect to other design flows.

Gate Level Simulation or Gatesim focuses on verifying the post layout netlist of the design. Gatesims are historically present from the days when designs were done

9. GATE LEVEL SIMULATION

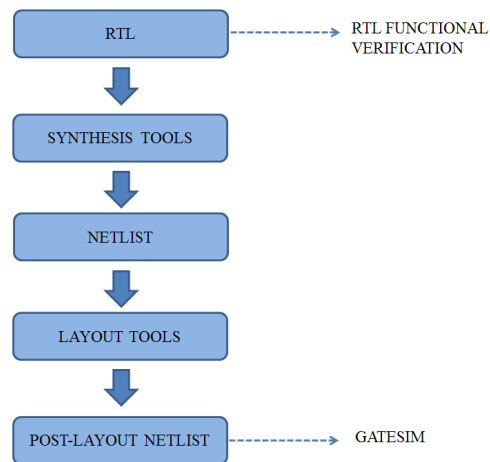


Figure 9.1: Design Flow

with gates rather than at RTL abstraction. Verification with gates is a huge confidence builder before manufacturing of actual silicon as they are also thought to complement and reassure results obtained from formal flow.

9.1 NEED FOR GATESIM

Gatesims are particularly effective for the following verification

- Power-up, reset propagation and initialization of the design
- DFT structures those are absent in RTL and added during or after synthesis
- non-resettable or un-initialized components such as memories
- Power related circuits those are absent in RTL
- Power switching verification
- Dynamic power estimation
- Validation of pessimistic behaviour of X-propagation in RTL simulation
- Asynchronous interfaces those are false-paths in STA
- Synchroniser logic and clock domain crossing verification

- Analog-circuit and digital circuit co-verification

Finally, Gatesim is a great confidence-booster in ensuring the high quality of the netlist. It lowers the risk of finding design, methodology or process issues in silicon.

9.2 LIMITATIONS OF LEC AND STA

Gatesims are targetted on post-layout netlist and that is almost clean of RTL bugs. The netlist also passes through couple of important verification steps such as Logic Equivalence Check (LEC) and Static Timing Analysis (STA), before it is targeted for Gatesims.

LEC: Logic Equivalence Check (LEC) is a formal verification tool that compares a reference design against a derived design to prove equivalence or to report differences. LEC does not require test patterns. Instead, LEC uses Boolean arithmetic techniques to prove equivalence between two design descriptions³. Although LEC uses sophisticated formal algorithms to identify, map, and compare nodes in the netlists, the complexity is hidden from the user⁴.

STA: Static Timing Analysis does a input-independent timing analysis of the gate level netlist. It asserts if the circuit could operate flawless without timing issues. It computes the worst-case behaviour of the circuit, over all possible manufacturing variables. STA tools are at ease in handling a complex design with huge number of paths as they consider one path at a time (whether they are real or potential false paths).

These formal static verification techniques are much faster and evolved than simulation based methods. However these verification methodologies, in spite of advancements in tools, cannot cover all verification requirements on netlist. In addition to reassuring results obtained by formal methods, gatesim helps in filling up the gaps left by these methods.

Limitations of LEC, which could be covered by Gatesims are:

- Limitation of Static Equivalence Checking tools to catch all X-propagation or X-generation issues.

- Two-state methodologies can miss RTL-versus-netlist simulation and RTL-versus-RTL simulation differences.
- Incorrect mapping issues due to naming at sub-block level which can result in false pass. This will not be reported at the sub-block level LEC, but Gatesims can flag such incorrect connectivity.

Limitations of STA, which could be covered by Gatesims are:

- **X-propagation:** STA deals only with logic domain of logic-0 and logic-1. There could be many sources of indeterministic states in the design such as uninitialized flops, output of memories, synchronisers, etc. Such indeterministic state value (X), could propagate through and cause failure of operation. Gatesim accurately models this behaviour and but STA does not.
- **Asynchronous Interfaces:** STA ignores certain asynchronous paths called as “false paths”, like with analog blocks or primary IO’s. And hence, STA cannot verify timing between digital and analog blocks whereous Gatesims could.
- **Reset sequence:** Verifying that all flip-flops resets into their required logical value. STA cannot check this as certain declarations such as initial values on signal are not synthesizable and are verified only during simulation.
- **Asynchronous clock-domain crossings:** STA does not check if the indeterministic value X produced for one clock cycle when logic passes clock domains, is suppressed or not.

9.3 ISSUES CAUGHT BY GATESIM

Some of the design flaws, those missed by other methods but caught by gatesims:

1. **X-Squashing** X-Squashing is a terminology to denote when uninitialised state value X get wrongly suppressed in simulation and does not propagate anymore through the logic, which it should have. In one case there was an X-Squashing

issue in behavioral RTL where the issue should have been found but a valid value was present, it was later found in gatesims.

2. **Glitches** Glitches are produced by combinational logic, and are not of concern in synchronous circuits as they are suppressed before next clock. Glitches in clock and reset paths are of concern. Here all methodologies fall short and gatesims are good in finding such issues.
3. **Uninitialized states in design** Source of un-initialized design states (X) could be easily found in gatesims. After identifying such scenarios appropriate initialization modelling needs to be performed to proceed with Gate simulation flow.
4. **Partitioning Issues** Design is partitioned to ease front-end design flows such as synthesis, STA, layout and LEC. Such act introduces discontinuity in such flows such as wrong constraints for different partitions and so forth. Gatesims are good at catching such issues, if appropriate stimulus is chosen.

9.4 ISSUES FACED BY GATESIM

At system level, Gatesim is one of the most challenging verification task. This is because as design complexity increases, the limitations with gatesims become more prominent. Important difficulties associated with gate level simulation are:

- Larger turn-around time (run, debug cycle).
- Limitation on size of netlist that can be verified through gatesim. This is an indirect cause due to larger build times and run times.
- Debugging the netlist simulation is challenging.
- Large compute and storage resource requirements.

Chapter 10

GATESIM METHODOLOGIES

Different methodologies could be adopted for netlist simulation and verification. First step would be to obtain the test vector stimulus that needs to be applied onto the netlist. One widely used method is to reuse the RTL testbench around the netlist. Another variant of this method could be to replace only a portion of circuit with netlist in the existing RTL verification environment.

Another method could be to capture test vectors from RTL simulation followed by applying it on corresponding netlist simulations. In such a method, comparison could be done between RTL behaviour (stored as captured test vectors) with that of netlist simulation. In AMD, gatesim verification is accomplished by one such methodology. Over the years, two different methodologies were adopted for test vector capture and stimulus application. These are now called as *Early Dual-Sim methodology* and *Co-sim based Gatesim methodology*. Due to its many shortcomings, the early dual sim methodology was discontinued over Co-sim based Gatesim methodology. Co-sim based Gatesim is the current de-facto methodology for gate level simulations in AMD.

10.1 EARLY DUAL-SIM METHODOLOGY

Early method for gate level simulation was a dual-sim or simulation-after-simulation method. In this methodology RTL simulation was done initially with test bench components. The test vectors for gatesims were generated during this RTL simulation using

“\$display” or VCD (value change dump). During netlist simulation, these test vectors were used as stimulus and comparison was done with the RTL output vectors. Figure 10.1 shows the simulation flow.

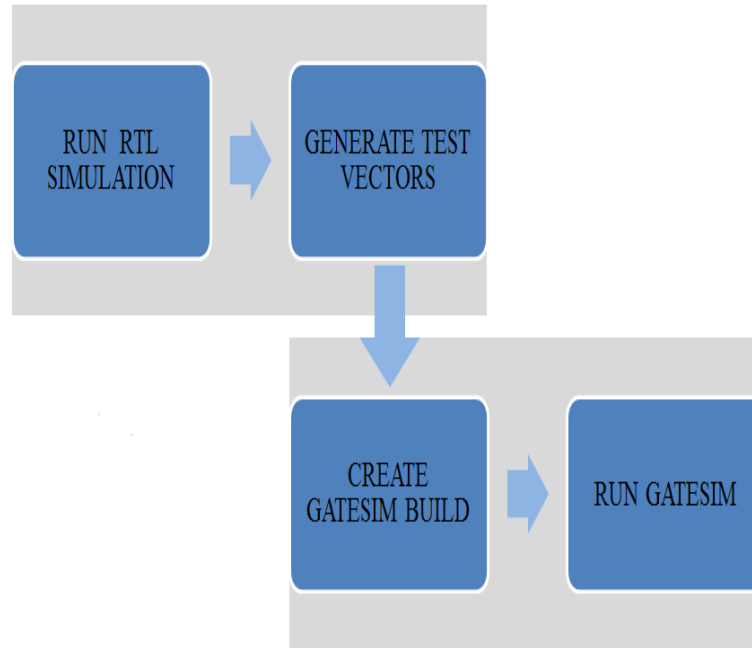


Figure 10.1: Early Dual-Sim Flow

This dual-sim method was widely used across industry due to its many advantages. The main advantage being best simulation performance with least compute requirements. However the earlier implementation of this method had multiple shortcomings which became more prevalent with increasing design complexity.

Shortcomings: The main issue with earlier implementation of dual sim methodology was the huge disk space requirement. Vector files were text files which had cycle based stimulus information. These files were large and simulation performance was also affected by disk input/output accesses. Another shortcoming of this methodology was, when stimulus is converted to cycle based information sampling errors were introduced. At times, these sampling errors were themselves causing simulation mismatches when compared to RTL simulations.

With increasing design complexity, the disk-space requirements became too high that the method could no longer be sustained and a new co-simulation based methodol-

ogy was adopted instead.

10.2 CO-SIM BASED GATESIM METHODOLOGY

Cosim-based methodology was conceived to solve some problems that existed with earlier dual-sim approach. To its advantage, the new method enabled ease of debug while maintaining consistent input vectors (devoid of sampling errors). It also made results comparison and debug easier. Figure 10.2 shows how stimulus is applied to netlist and comparison of output is done in cosim method.

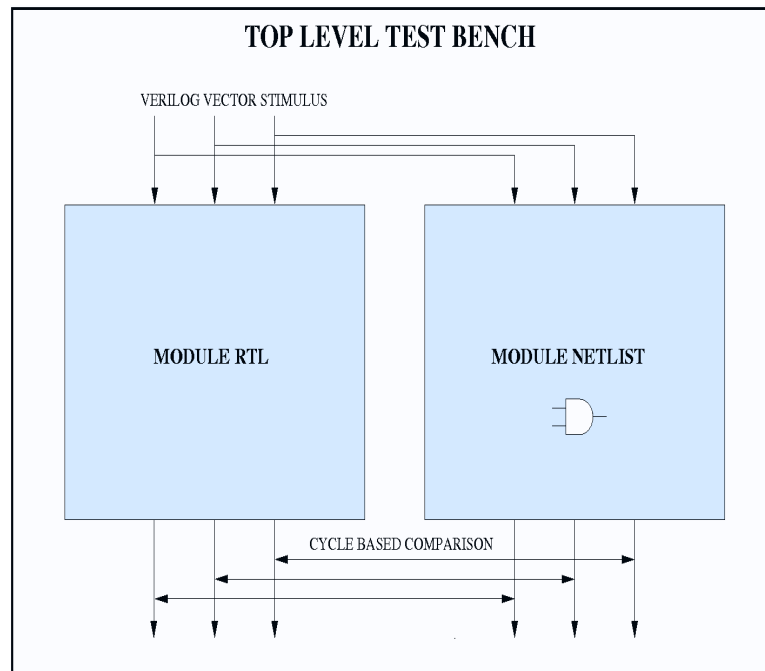


Figure 10.2: Co-sim Based Gatesim

In Cosim methodology, a single combined simulation consisting of the netlist with behavioral RTL and stimulus components is made. In this simulation, behavioral RTL and gate models are run in lock-step with their inputs tied and the comparison of the behavioral RTL and gate outputs is done “on the fly”. Figure 10.3 shows cosim flow.

Major steps involved in this flow are:

1. Getting gatesim files

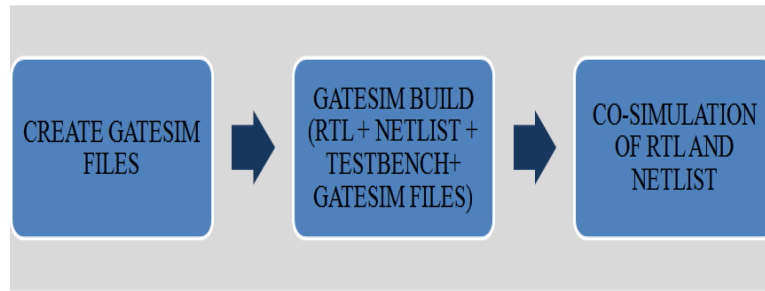


Figure 10.3: Co-sim Based Gatesim Flow

Input to gatesims are obtained from LEC flow. The list includes the netlist file, files holding information regarding IO/Register mapping, gate defines, compare enables and testbench force files. These inputs from LEC stage are processed by a set of scripts for developing intermediate files which are needed by cosim build infrastructure. These files are exclusively required for netlist simulations and described below

`gatesim.v` instantiates top level netlist and ties its inputs with the behavioral RTL.

`compare.v` contains comparators those compare the outputs and register mappings, for each cycle of simulation.

`forces.v` contains all the force/release/assign commands for the gates corresponding to RTL force/release/assign statements. Forces are required in the design to shorten reset, initialize training parameters, initialize fuses among other things.

2. Getting gatesim build

Next stage is to enable a build structure supporting the co simulation of RTL and Netlist. The build includes:

- Netlist to be verified
- Complete RTL of SoC
- Test bench components in its entirety
- Gatesim files

3. Run cosimulation of RTL and netlist

Running co-simulation is very similar to running RTL simulation. Output files include <testname>.out which contains the simulation transcript of the entire test and <testname>.fsdb (waveform dump file). The simulation transcript also contains gatesim errors called as “miscompares”.

As the netlist stimulus is obtained from a live RTL instead from stored vectors, cosim based gatesim overcame the biggest limitation associated with dual-sim. Over time, some good set of scripts aiding testbench generation, force generation were standardized. This method became the standard method for gatesims, close to a decade.

10.2.1 ISSUES WITH CURRENT CO-SIM METHOD

Co-sim based Gatesim overcame all known limitations associated with early methodology. As design complexity grew, it brought in new set of unforeseen limitations. Of those, the important ones are already discussed in Section 9.4.

In order to better understand these limitations certain experimental analysis were done. Experiments showed that the simulation performance of gatesim was affected sometimes as low as 10% with respect to its counterpart RTL simulations. This indicates that:

- RTL Simulations contribute major to simulation performance than netlist.
- Simulator spends more time in simulating RTL and verification components than netlist.

On further investigation it became clear that RTL simulation, which is simulated redundantly for the sole purpose of generating test vectors influences the simulation performance greatly. Such complex SOC design has multitude of Verification components in different programming languages including C, C++, SVTB, OVA, SVA and that these verification components take a big share of simulation cycles and have negative effect on simulation performance.

10. GATESIM METHODOLOGIES

Evidently it was not an appropriate use of compute resources by having live RTL simulation every time, for the sole purpose of test vector generation. The analysis provides convincing evidence for us to attempt changes in existing cosim-based methodology.

Chapter 11

IMPROVED DUAL-SIM APPROACH TO GATESIM

Analyzing limitations associated with *early dual-sim approach* shows that the main cause of inefficiency was the method used to capture, store, and applying test vectors onto the netlist. Analyzing limitations associated with *co-sim approach*, it was inferred that the cause was bulky test-bench components associated with RTL simulations. Hence an improved solution would contain minimal testbench components retained and have an efficient method to capture, store, and apply test vectors.

Test vectors are nothing but signal values at specific point in time. There are already different formats to store this information efficiently. FSDB is one such format. Hence it was suggested to improve gatesim methodology using FSDB itself as the format to store test vectors. The proposed solution should also improve on

Storage requirements : Ensuring that storage resources are effectively used

Turn-around times : Should avoid re-build for different test vectors

FSDB⁸ or Fast Signal Database is a signal data file, similar to VCD¹ but much more compact. This format is in wide use across industry. Quick analysis revealed that FSDB as input test vectors could be accomplished. Existing API's provided by Verdi⁸ tool set for FSDB format could be used to retrieve values from FSDB. PLI/VPI¹ could be used to drive stimulus onto netlist.

11. IMPROVED DUAL-SIM APPROACH TO GATESIM

The improved methodology becomes a dual-simulation methodology with two separate simulations.

1. First simulation with non-gatesim components to generate the test vectors in FSDB format.
2. Second simulation with only gatesim components with capability to apply test vectors from FSDB directly.

11.1 DUAL-SIM FLOW

In dual simulation approach the idea is to have two simulation but unlike co-sim not in parallel. Instead here the simulations are separate from each other. We run the RTL simulation which will have test bench components for generating test vectors that will drive the inputs. This same test vectors are required for simulating the netlist environment. In cosimulation the same test bench will provide the stimulus to RTL and netlist. However in a dual simulation environment first RTL simulation will happen and the generated test vectors are stored in FSDB, which is used for simulating netlist. Figure 11.1 gives the flow of proposed dual-sim approach.

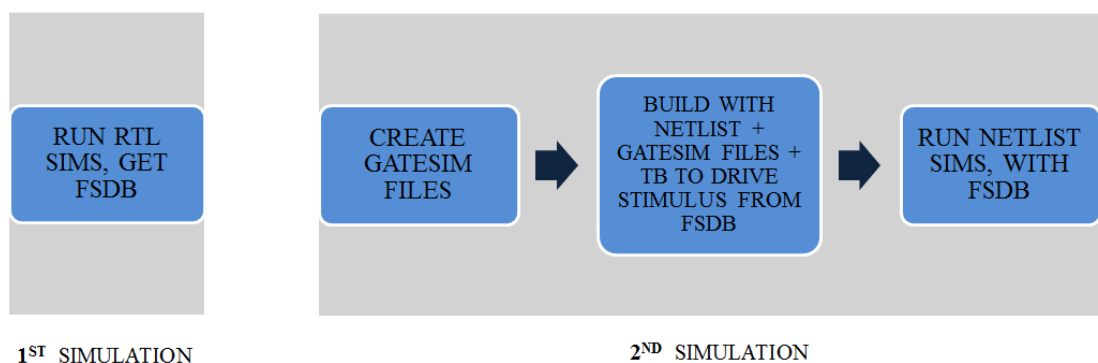


Figure 11.1: Dual Sim

Major steps involved in this flow are:

1. Run RTL simulation

11. IMPROVED DUAL-SIM APPROACH TO GATESIM

RTL simulation is done along with test bench components for generating test vectors and verification. This is the same build for RTL verification stage. The simulation signals needs to be dumped into FSDB and for this the signal dump should be enabled during the run.

2. As in the case of co simulation, Gatesim files need to be generated from input files obtained from LEC. Same infrastructure used in co-sim can be used for this.
3. Get the gatesim build. Here the build structure will have the netlist and gatesim files only. RTL and the bulky testbench components are absent.
4. Run simulation with specific FSDB file.

11.2 IMPLEMENTATION

The implementation of the FSDB based dual simulation approach can be explained in three steps:

- How the FSDB file is generated.
- Using FSDB dump files as test vector source.
- Applying test vectors on to netlist.

11.2.1 Generating test vector source file - FSDB file

As discussed in previous section, the tests vectors for netlist simulation are same as the test vectors used for RTL simulation. These vectors are generated by testbench environment created for RTL simulation run. These test benches will have various components for stimulus, assertion, debug feature etc. In this project we have used VSC Verilog simulator developed by Synopsys for RTL as well as net list simulation.

VCS: VCS is a high-performance Verilog simulator that incorporates advanced, high-level abstraction verification technologies into a single open native platform. VCS provides a fully featured implementation of the Verilog language as defined in the IEEE Standard Hardware Description Language Based on the Verilog Hardware Description

11. IMPROVED DUAL-SIM APPROACH TO GATESIM

Language (IEEE Std 1364-1995) and the Standard Verilog Hardware Description Language (IEEE Std 1364-2001). It supports most of the design and assertion constructs in SystemVerilog and PLI's for interface with other models, provides direct C kernel interface etc. This is accepted as one of the fastest simulator when it comes to RTL verification.

RTL SIMULATION

For RTL simulation the RTL sources and test bench files are compiled first and an executable file is generated for running the simulation. During the simulation VCS generate log files or reports giving details of the simulation. One feature provided by VCS is parallel FSDB dump. For this VCS have options to enable "dump" during simulation run. This will dump all the signals involved in RTL simulation and these FSDB dump files are used by waveform viewers as signal source.

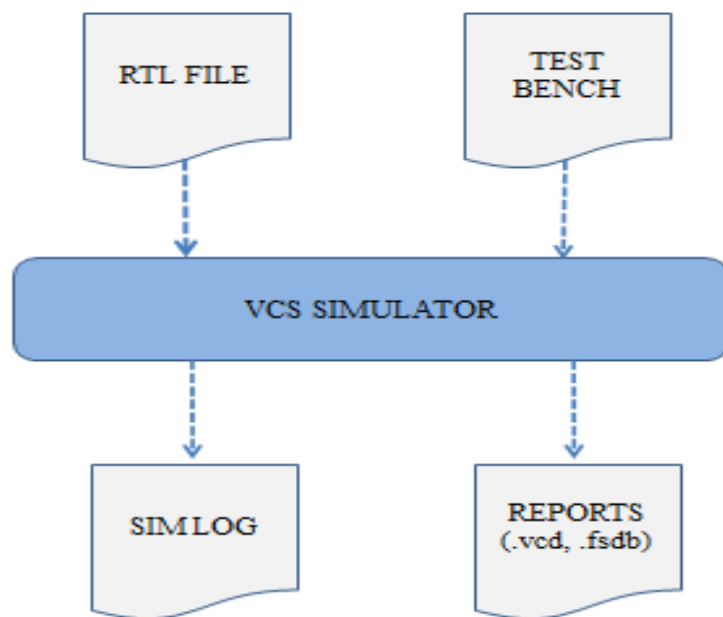


Figure 11.2: RTL Simulation

Once the RTL simulation completes with VCS signal dump option enabled, a .fsdb file is generated. This file is the vector source for netlist simulation. For each design and for each test condition, we need to perform dump enabled RTL simulation initially be-

fore going to netlist simulation. This is a onetime simulation from gate level verification point of view.

11.2.2 USING FSDB AS TEST VECTORS

Once the FSDB file is generated, next stage is extracting the signal values from this file and converting it into test vectors that can be applied for netlist simulation. Figure 11.3 shows the layout of programming infrastructure developed for this.

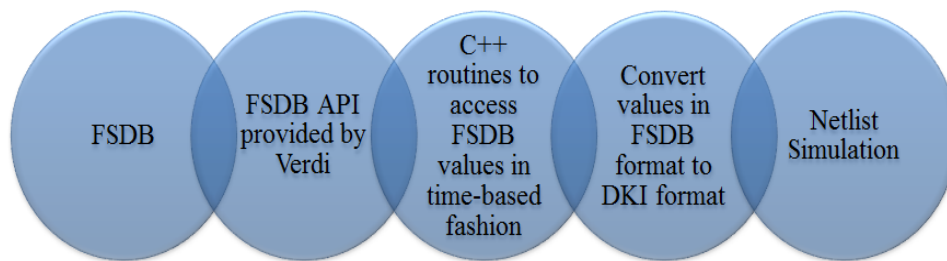


Figure 11.3: Accessing FSDB Signals

Figure 11.3 shows how test vectors are obtained from .fsdb file and attached onto the netlist simulation flow. The various stages involved are explained below.

Extracting data from FSDB: In FSDB signals are stored in binary format and can be accessed only by specific tools or using some appropriate APIs. In this project we are using FSDB APIs provided by Verdi. These API's allow us to access each signal separately.

Once the list of RTL signals that need to be extracted is decided, make FSDB signals object handles corresponding to each signal. FSDB signal objects are also defined by the API and allow signals in FSDB file to be attached to these objects. Special "Attach()" routines are provided for signal attach with objects.

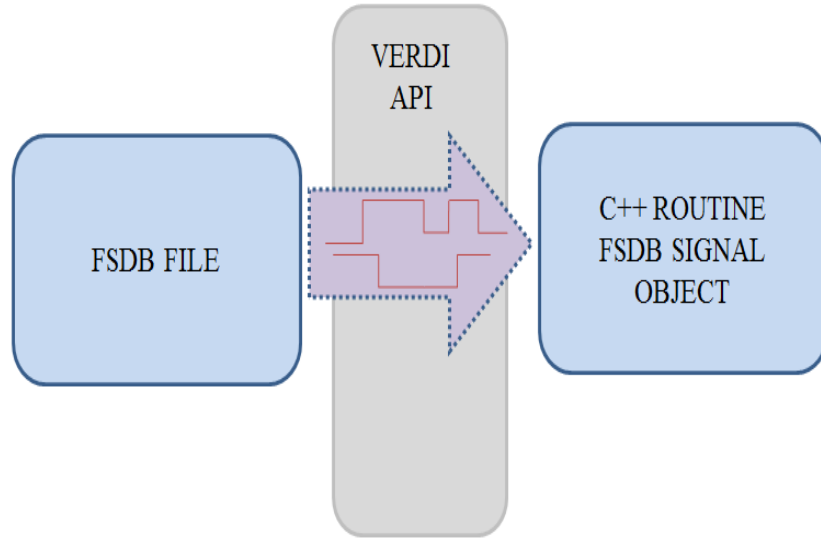


Figure 11.4: API For Accessing FSDB Signals

However just attaching to the signal is not enough. A larger C++ infrastructure is required for accessing signals for gatesim because of the following reasons:

1. Values contained in FSDB needs to be accessed in time-based fashion.
2. Typical netlist contained hundreds of stimulus points with many wider bus-signals.

The C++ infrastructure will open the FSDB file and initiate a playback through the file. Whenever a signal that is attached using APIs changes, the C++ will identify this and will ensure the changed value is made available to the netlist simulation. Figure x shows the complete flow of the C++ routine developed for FSDB signal access, conversion and application onto Verilog component.

A set of RTL signal accessed at time based manner and applied onto netlist is effectively doing the job of test vectors as vectors are nothing but signal values at different instants of time. Once signals are accessed from FSDB file by API's and C++ routine developed next step is applying it onto netlist. However Verilog based netlist cannot directly interact with extracted FSDB format objects. For this a conversion stage is required.

11. IMPROVED DUAL-SIM APPROACH TO GATESIM

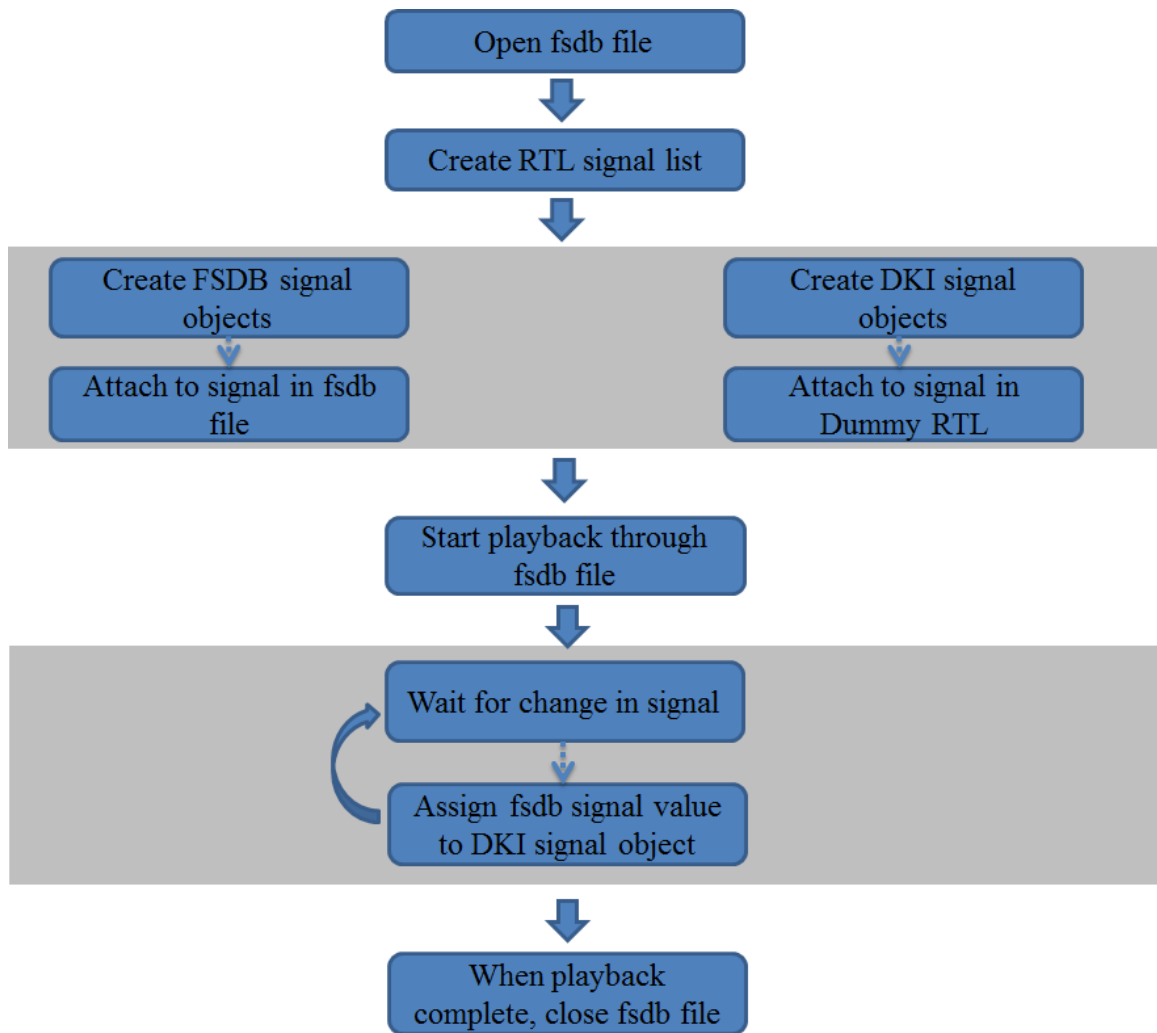


Figure 11.5: C++ Routine for FSDB Signal Extraction

Convert FSDB format to DKI format : The FSDB format signals are converted to a standard format that can work with Verilog. There are many interfaces that allow Verilog-C interaction such as PLI, VPI (or PLI 2.0) or DKI. In this work we are using DKI which is an API that is supported only by VCS. This has the advantage of less simulation overhead and smaller memory footprint compared to VPI interface. For converting FSDB signal format to DKI format, we are assigning signal values of the FSDB signal object to a corresponding DKI signal object. Similar to the creation of FSDB signal object list, create a DKI object list corresponding to the RTL signals.

Whenever the C++ moves in time and identify a change in signal value in FSDB signal, an assign statement will assign the new FSDB signal value to DKI signal object.

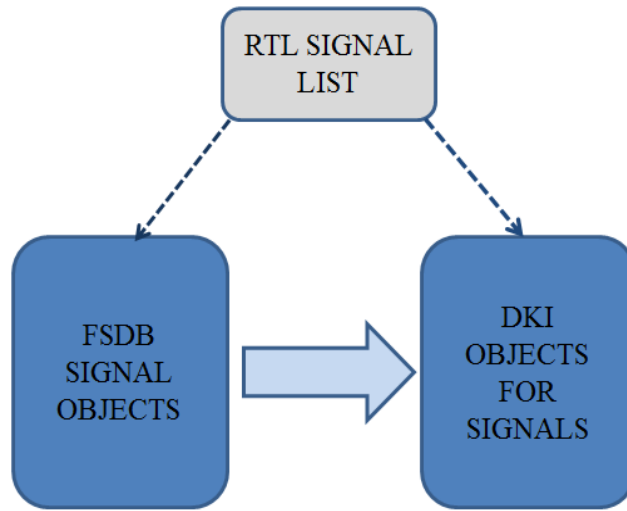


Figure 11.6: FSDB Format to DKI format

These DKI signals can drive Verilog signals by using "Attach" routine provided by API, that ties together DKI object with RTL signal.

11.2.3 NETLIST SIMULATION FLOW

In the previous section we have discussed extracting FSDB, accessing it in a time based manner and converting it to a format that can interact with Verilog. Next step is using these signals for netlist simulation.

Figure ?? shows data flow for applying test vectors onto the netlist and how final comparison is done.

DUMMY RTL

The test vectors obtained from the FSDB are applied to a dummy RTL module. Dummy RTL module has no logic other than the RTL signal list that are required for netlist simulation, that is the driving Input signals and the output signal for RTL-Netlist output comparison. The DKI attach will link these empty RTL signals with DKI signal objects, and any change in DKI object is reflected as change in value of the attached dummy RTL signal. Figure 11.8 shows a code snippet of dummy RTL module. The input and output signal list from the original RTL code is consired as "reg" in dummy RTL module.

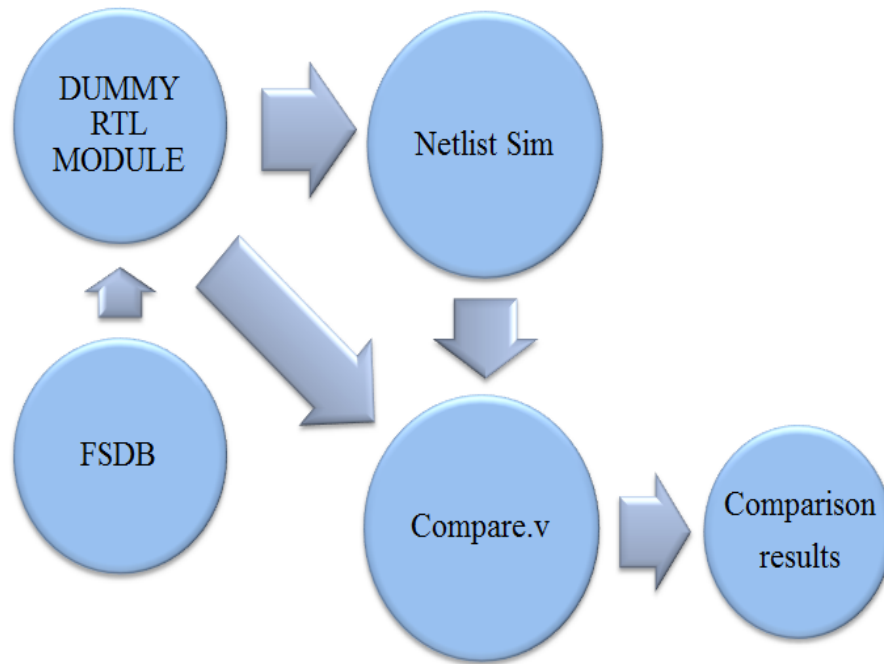


Figure 11.7: Netlist Simulation

DUMMY RTL DRIVING NETLIST

Dummy RTL signal which is having the original RTL signal values from FSDB through C++ routine will drive the netlist input signals. This is done by regular Verilog module instantiation of netlist top level module with dummy RTL inputs driving the netlist input. Figure 11.9 gives a snippet of this Verilog instantiation of netlist module.

Now during simulation the netlist inputs are driven by the dummy RTL signals.

OUTPUT COMPARISON

During the simulation, a testbench compare module (compare.v) will have cycle by cycle comparison of the netlist output signal vs the dummyRTL signals which corresponds to the original RTL output signals. Figure 11.10 is a code snippet showing comparison of an output signal.

The Netlist simulation is also done using the same VCS simulator. Netlist signals can be dumped into fsdb in the same way as RTL signal DUMP. This netlist FSDB file and the original RTL FSDB file will be used by waveform viewers for netlist verification and debugging.

11. IMPROVED DUAL-SIM APPROACH TO GATESIM

```
module dummy_RTL(  
    reg input_signal_1;  
    reg input_signal_2;  
    reg input_signal_3;  
    -  
    -  
    reg input_signal_n;  
  
    reg output_signal_1;  
    reg output_signal_2;  
    reg output_signal_3;  
    -  
    -  
    reg output_signal_m;  
);  
endmodule
```

Figure 11.8: Dummy RTL

11. IMPROVED DUAL-SIM APPROACH TO GATESIM

```
netlist GATE (  
    .input_signal_1(dummy_RTL.input_signal_1);  
    .input_signal_2(dummy_RTL.input_signal_2);  
    .input_signal_3(dummy_RTL.input_signal_3);  
    .input_signal_4(dummy_RTL.input_signal_4);  
    -  
    -  
    .input_signal_n(dummy_RTL.input_signal_n);  
  
    .output_signal_1();  
    .output_signal_2();  
    -  
    -  
    .output_signal_m();  
);
```

Figure 11.9: Netlist Instantiation

```
always @(negedge CLK) begin    // comparator for output signal 1  
    #1  
    if (GATE.output_signal_1 != dummy_RTL.output_signal_1)  
        $display("Mismatch for signal 1");  
    end  
end
```

Figure 11.10: Cycle Compare Output Signals

Chapter 12

RESULTS

A new gate simulation flow based on separate simulation of RTL and netlist was developed. The simulation performance analysis was done by comparing with the performance of existing methodology. A set of standard test cases were used to simulate Interface TLM (Top Level Module). First the test was run in co-simulation environment and then in the new proposed dual simulation environment on the same machine for benchmarking. the machine features are :

- Linux 2.6.18-308.1.1.el5
- Authentic AMD family F model 1 stepping 2
- AMD FX(tm)-8150 Eight-Core Processor
- MemTotal: 32925800 kB

Table 12.1 shows simulation performance comparison and Table 12.2 shows simulation memory requirement comparison.

12.1 SIMULATION PERFORMANCE ANALYSIS

Table 12.1: Simulation Performance Comparison

Stimulus	Co-sim Simulation Time (in sec)	Dual-sim Simulation Time (in sec)	Improvement (X times)
Pattern 1	821245	79965	10.27
Pattern 2	883227	85731	10.3
Pattern 3	854760	83083	10.28
Pattern 4	456881	46071	9.91
Pattern 5	709871	69605	10.19

12.2 MEMORY REQUIREMENT

Table 12.2: Simulation Memory Requirement

Stimulus	Co-sim Simulation Mem Req (in Mb)	Dual-sim Simulation Mem Req (in Mb)	Improvement (X times)
Pattern 1	1103.9	98.8	11.17
Pattern 2	1103.9	98.8	11.17
Pattern 3	1105.1	98.8	11.18
Pattern 4	1103.3	98.8	11.17
Pattern 5	1103.3	98.8	11.17

Chapter 13

CONCLUSION

A new dual-sim or sim after sim flow for gatesim was developed and simulation performances was benchmaked on dedicated machine and compared against current co-simulation method. The simulation performance shows a consistent improvement of around 10 times over the current method. Use of FSDB as the source of test vectors keep the file size small and get rid of bulky time hogging test bench components, and thereby reducing rerun time. This 10 times improvement in simulation performance will help in reducing delay in tape out due to gatesim verification overheads. The method still require a one time RTL simulation run for generating FSDB file. The existing gatesim flow can be maintained as it is and only few changes need to be included in current flow for accommodating the new method.

Potential for future work: The performance of dual-sim method can be further increased by improving the VPI/PLI access methods.

Chapter 14

REFERENCES

Bibliography

- [1] IEEE, “1364-2005 - *IEEE Standard for Verilog Hardware Description Language*”. IEEE STANDARD, 2005
- [2] IEEE, “1800-2012 - *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*”. IEEE STANDARD, 2009
- [3] Randal E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” *IEEE Transactions on Computers*, vol.C 35, no. 12, 1986.
- [4] McDonald, William, and Janny Liao, “Logic Equivalence Checking Has Arrived For FPGA Developers.” *Design and Verification Conference (DVCon)*, 2006.
- [5] AMD Corporation (May 2011). “Volume 2: System Programming”. *AMD64 Architecture Programmer’s Manual*. AMD Corporation. Retrieved 2011-10-29. http://support.amd.com/us/Embedded_TechDocs/24593.pdf
- [6] Cadence. (2013, Jan.) Functional Verification Survey-Why Gate-Level Simulation is Increasing.[Online]. Available: <http://www.cadence.com/Community/blogs>
- [7] Cadence. (2012, Jan.) Gate-Level Simulation Methodology-Improving Gate-Level Simulation Performance.[Online]. Available: http://www.cadence.com/rl/Resources/white_papers/Gate_Level_Simulation_WP.pdf
- [8] SpringSoft. (2013, May.) Verdi Automated Debug System. [Online]. Available: <http://www.springsoft.com/products/debug-automation/verdi>

- [9] Dygraphs. (2013, Jun.) JavaScript Visualization Library. [Online]. Available:
<http://dygraphs.com/>

BIODATA

Name : Meera Mohan

Qualification : B.Tech (Electronics and Communication)
Mahatma Gandhi University, Kottayam

Contact Address : D/O Mohandas. E. K
Margangattu House
Memana, Oachira
Kollam, Kerala-690526

Contact Number : 7411352081

Email id : miramohan@gmail.com