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   Mr. Narendran Kumaragurunathan

Professor   SMTS Design Engineer

Dept of E & C, NITK   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 **Prof. M. S. Bhat** **Prof. Muralidhar Kulkarni**

Project Guide **Project Guide** **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 there by 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 spends 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.

Contents

List of Figures

List of Tables

SYMBOLS and ABBREVIATIONS

API   Application Programming Interface

FSDB Fast Signal Database

GUI   Graphical User Interface

IP   intellectual Property

LEC   Logic Equivalence Check

RTL   Register Transfer Level description of circuit

SoC   System on Chip

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 verfication 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 verifcation tool proves or disproves each property appropriately. In formal verification, user need not generate stimulus but specification of invalid stimuls 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 IN AMD

In AMD, most SoCs are built around one or more x86 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 comparied 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 the event of a simulation failure, these 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 co-related information are spread across in different files. This greatly affects an engineer's ability to debug and converge on the cause. If there exists an interface which correlates different information and presents only information as required by the user, it would aid significant improvement convergence in debug. 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 ??-*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 introduce 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 for 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
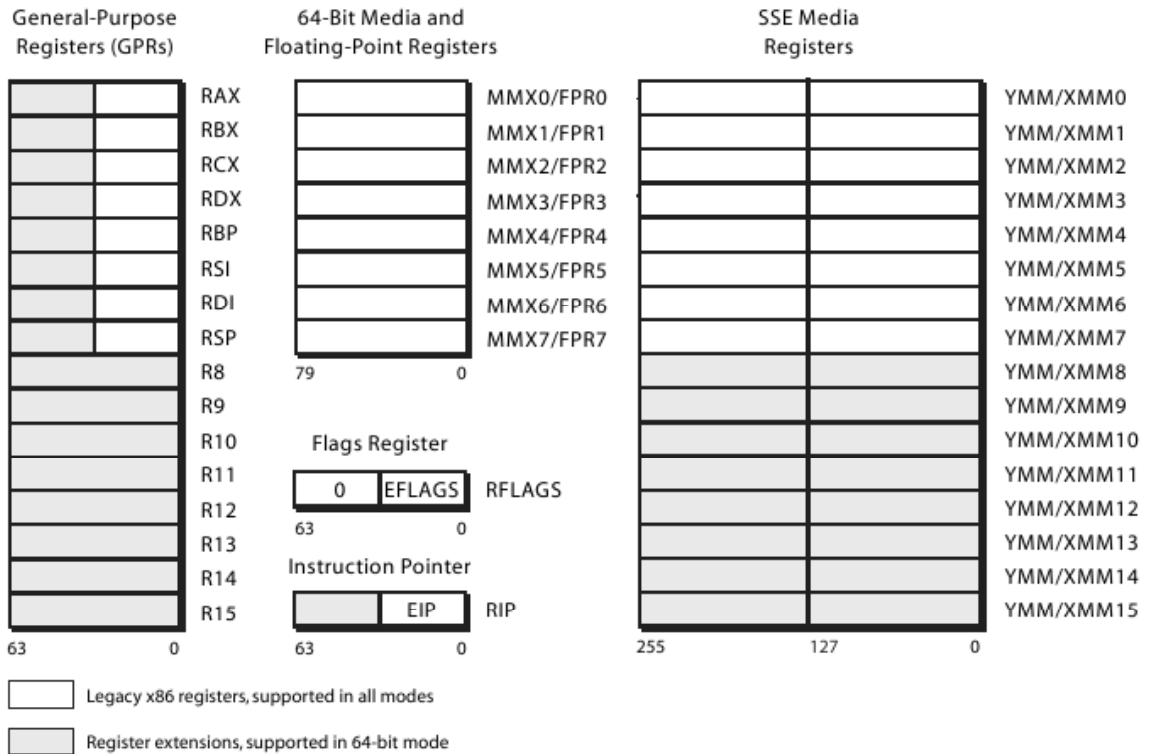
Figure 2.1: AMD64 Registers

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 uses 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.

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.
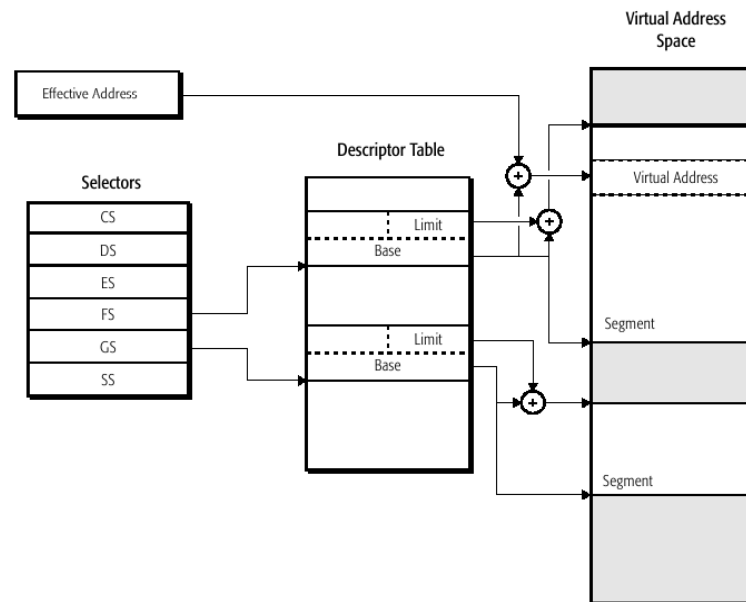
Figure 2.2: Segmented Memory Model

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 2.3shows an example of paged memory with three levels in the translation-table hierarchy.
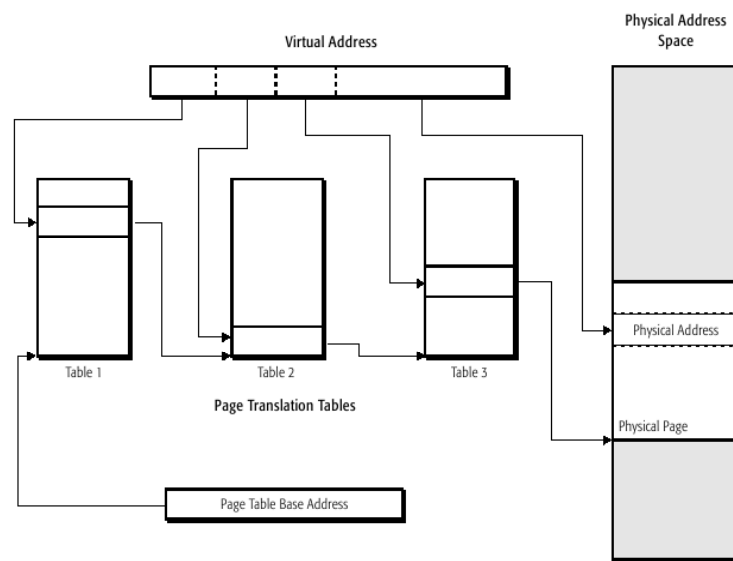
Figure 2.3: Paged Memory Model

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 aka RTL. It is then verified against functional requirements. The verified RTL model is them mapped into an architecture made up of intellectual properties (IP) blocks. The system level verification is done to verify the architecture against the intended functional and other requirements.

Functional verification validates that the design meets the requirements. Test cases are created based on the 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   TEST

Most of the verification at AMD is for verifying the x-86 processor cores and SoC level IPs. At this level of abstraction, verification of interaction between 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 as to provide specific stimulus to the design and the outcome is compared aganist desired values to ensure validity. Ideally the design should be varified against all possible senarios that could arise and once it pass all the test it can be considered fully valid design. In cases where a particular test fails, the verification engineer needs to find out the cause of failure, understand the design or verification aspect that led to the particular situation and make appropriate changes. This is called debug step in verification.

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 manly different causes for test but most test failures are due to:

- Self check

- Assertions/Checkers

A self check failure is caused due to self-tests. These are tests written such that they evaluate the results, compare it with desired value and finally report fail or pass.

Assertions or checkers generally monitor particular signal or state during the design and report failure when ever the monitored value deviates from desired value.

In this project we are considering self-test based verification and debugging of self-test failure.

3.2.1 SELF TEST FAILURE

Self-tests are test scenarios written to prove that device under test are functioning correctly in a specific situation. These are call "Self-test" as they are capable of deciding if the outcome was successful or not after simulation. These tests are normally hand written by the verification engineer rather than randomly generated. Hence self-test can string together specific stimulus of interest and determine pass or fail status on its own

without relying on another tool.
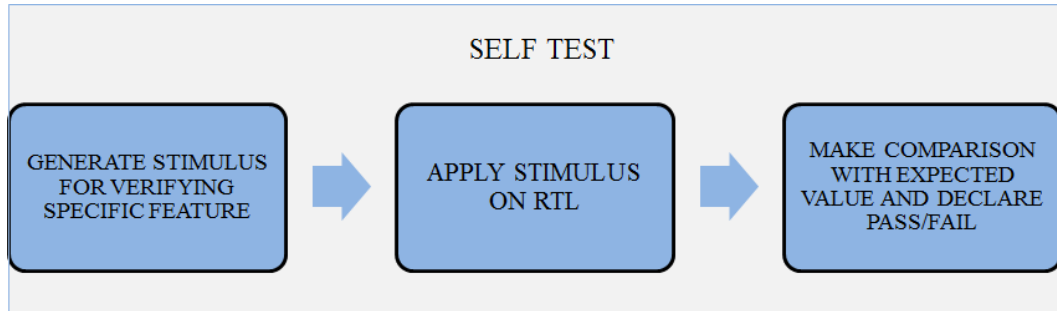
Figure 3.1: Self-Test

Figure x details the flow of a self test. In AMD most of the self-tests are a collection of x86 assembly language program. Tests are compiled by a script which calls an assembler producing an output that can be run on the RTL model. The stimuli are generated by the test and are applied to the DUT. The comparisons are done by the test itself and pass or fail flag is generated.

3.3   DEBUGGING A SELF TEST FAILURE

Self-tests report the occurrence of test case failure. Once this happens, next step is analyzing 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, all the information regarding the execution flow as well as a reference flow which has the ideal values and status are required.

For this purpose of analyzing, the RTL is simulated along with an instruction level reference simulator. The reference simulator is a software model which imitates the design functionally and the runs in parallel with the RTL. This parallel simulation is known as co-simulation.

3.3.1   CO-SIMULATION

The instruction level reference simulator aka ILS is an x86/x86-64 programming model. It mainly models the register states and memory features and act as the ideal reference point to compare with. The test provide stimulus to both the RTL model and reference model, the model will act as the ideal reference point. 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.
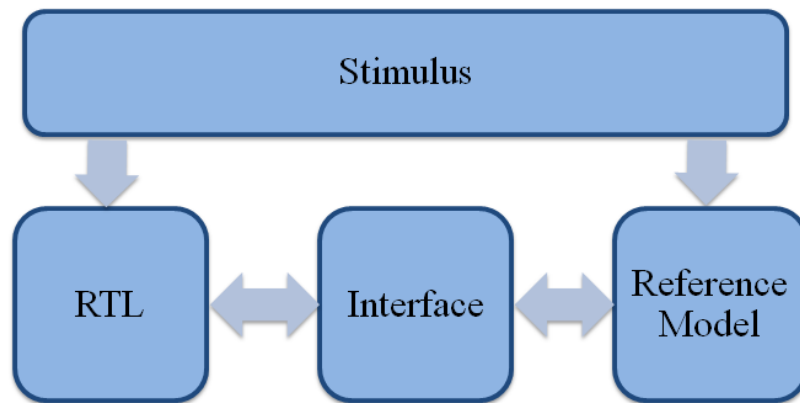
Figure 3.2: RTL-Reference Model Cosimulation

INSTRUCTION LEVEL SIMULATOR

The x86 instruction level simulator starts simulation with first initializing the contents of its memory from input files. The simulator emulates the fetch/decode/execute algorithm of a scalar processor, producing an output log known as processor execution log, describing the instruction, its results and any side effects on the processor state. The simulator models debug features, exceptions and interrupts as wellas processor specific features. Supported processor states include x86 general purpose registers, flag registers, control registers, media registers and most common model specific registers and both memory and I/O space. The simulator runs multi-threaded code to simulate multiprocessor 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 simulator are compared. Difference detected in processor states are considered as mismatches; difference in memory locations written are considered as write mismatches and once these are found the co-simulation terminates. 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 after every instruction of the RTL 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 include [] :

- Initialization: Initialize memory model, attach RTL signal and initial some specific RTL signals.

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

- Interrupt Handling: Interface informs simulator about pending interrupts

- Comparing: Compares RTL and reference model registers; integer, FP, control and status word. Also reports any write 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 all the execution details. These are called "*processor execution log*". 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 the self test report a failure, simulation terminates and debug phase starts. For figuring out the cause of failure, verification engineer need to trace through this processor execution log files. 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 compiled test files needs to be analyzed and co-related for debugging. 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, and

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 VISUALIZING PROCESSOR EXECUTION

Once the program is loaded in processor memory the execution starts. Generally modern microprocessors today adopt instruction level parallelism for high throughput. Micro-architectural like instruction pipeline, superscalar execution, register renaming, speculative execution, branch prediction etc. are employed in order to exploit instruction level parallelism []. These micro architecture elements will work together for high performance execution. All these internal operations results in a complex execution flow with multiple operations happening in each cycle with different level of dependencies between data, instruction and memory.

An execution log file captures all the information regarding the processor state and activity during the execution of a code. This is actually the entire history of simulation. Each entry in the log file will have mainly the following information regarding processor execution cycle:

- The instruction number and opcode

- Thread Id (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 updates made

On the onset of a simulation failure, these information are very vital. For understanding how execution log information helps in debugging failure let us consider two test scenarios.

4.2 CASE STUDY

Let us consider two simple asm tests as case studies for understanding how processor execution log details helps in finding out why and how errors occur. In each case a few failure cases are considered.

4.2.1 TEST A

Consider an assembly code of testing a memory module. The test writes a value into a memory location. The data is later read from the same location and read data value is compared with the original value written into the location. The test flags a fail of pass based on the comparison.

Input : *data*, *address*

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

1 Initialization: Select memory bank by setting *ControlRegister*

2 *RegA* ⟵ *data*

3 *RegB* ⟵ *address*

4 Memory [*RegB*] ⟵ *RegA*

5 *RegC* ⟵ *data*

6 *RegD* ⟵ Memory [*RegB*]

7 **if** *RegC == RegD* **then**

8 | report *pass*

9 **else**

10 | report *fail*

11 **end**

Algorithm 1: Memory Read-Write

The test verifies 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 situation where the comparison fails. This can happen due to many reasons. Following are a few scenarios which can lead to a failure:

Case 1 : If due to some external process the control bit for bank selection is changed in between the execution, the data read will be from wrong memory location leading to failure.

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 chosen is read only. This will cause the wrong data to be updated into the memory and comparison fails.

ANALYSIS

Case 1: The memory bank selected should stay same throughout the program execution. This change in memory bank will cause the read operation to take value from wrong memory block. This will lead to self-test failure.

Now to understand when and where the actual error occurred, we need to keep track of the control register value and see where the value changed from expected value.

Case 2: Each memory block has a fixed size. The base value will be selected on setting the control bit. The offset value is provided by the test. A valid address value will be within the range of memory block that is between the lowest and highest offset. Any attempt to access a value which doesn't lie within this range will evidently lead to an error.

If such a situation occurs, the user needs to be aware of the particularities of each memory write operation. Details on instance of memory write, actual physical as well as logical address value, instruction cycle number etc are required to figure out if this was the cause of test failure.

Case 3: Certain bits of some specific registers are set as read-only. Consider a case where a 32 bit register A has its lower byte set as read-only and has the value XXXXXX00h. If the test is trying to set this register to a certain value, for example FFFFFFFFh, chances are that the value that is actually set might not be a FFFFFFFFh but some other value possibly FFFFFF00h. To catch such an error, the verification engineer needs to know the value stored in each register used during test at all cycle.

4.2.2 TEST B

Consider a string conversion program. The code reads an input string in upper case and converts it into lower case and display. Consider ASCII character coding and for converting lower case to lower case add decimal value to upper case value.

Input : *string*

  1 Initialize stack memory to 0

  2 Store input string into stack memory

  3 For stack[*base*] **to** stack[*maxOffset*]

Algorithm 2: Memory Read-Write

The string conversion program converts each character from upper case to lower case on by one by using a loop. This program can fail due to many reasons. Let us consider a couple of scenarios.

Case 1 : If the loop initialization is wrong. Suppose the base address of the stalk is from 1 and the program loops from 0th location. This will lead to an invalid conversion.

Case 2 : If the looping misses out on final character. In many scenarios if so happens that code exit from the loop before branch is executed.

ANALYSIS

Case 1: The base address of stalk declared for storing characters of the string is the starting point of the loop. If while looping the looping variable is initialized incorrectly then the code will point to invalid location. In case of such situation, information on the stack pointer register value is required.

Case 2: This situation arises when looping is terminated at wrong instant. The execution takes the wrong branch and misses out of last loop. Such situation requires knowledge of branching during program execution.

From the analysis of both the test and the selected error cases, we can conclude that to understand the cause of error some specific set of information are very important such as:

TEST A

Case1 : Control registers value at each instance of time

Case2 : Memory write/read information

Case3 : General purpose registers details

TEST B

Case1 : Stack base address information

Case2 : Branch Target information

In general for different test and error possibilities, all information regarding the processors execution flow is required. From the point of failure, a trace back through the execution log details and corresponding comparison with the expected assembly code action must be made. The above cases show that main aspect of failure analysis is co-relating the execution information with the test code and tracing the execution flow.

4.3   Co-relating information

Correlation of list file and log file is done by comparing the address values. 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 the cases considered in the previous section, this comparison at each stage and verification of processor action and states are required to understand how and why the test failed.

4.4   Tracing the log files

As explained earlier, processor execution logs are generated during test simulation. These files hold almost all details that can be possibly identified during simulation.

In SoC level verification, each simulation will have a collection of many tests verifying intimate details of the design. The test plans will have many small tests and will be a big collection and have thousands of code lines. In such cases it is rather obvious that the execution log file for such tests stimulus will be huge as it covers cycle by cycle

processor details. This vast wealth of information will be in generated in a simulation time based manner with details of various threads mixed together.

During debug the verification engineer need to trace the log file contents to find out the cause of failure. This navigation through such huge file can be a very tedious process if done manually by calculating linear address and then using string search to find the address in log file and asm file to correlate. In fact the data traversal can move up and down for verifying different aspects. This will consume a lot of verification time.

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 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.

Chapter 5

INTERFACE IMPLEMENTATION

The GUI provides user friendly and easy data navigation which will reduce debug efforts. All relevant processor execution log information and asm test details are provided to the user through a web interface. The interface uses data visualization JavaScript features to develop graphical representation of the log information. Features of the GUI are designed such that, traversing through the log information and comparison with the asm list file lines are made easy. Major processor operations are categorized and extracted to have a clear view of processor actions. From analysis done in Chapter 3, the following sets of features are expected to help the user and are our main design objective while implementing the interface:

- Visualizing the processor execution flow in each thread

- Information regarding each Memory write/read, I/O write/read, Branching etc

- Interrupt and exception happening during execution

- Easy traversal to the asm instruction line.

- Register values at each instances

- Comparison of register values between two cycles

- All the cycle information provided by the cycle in execution log for detailed reference.

Once the simulation is completed and failure is reported the debug phase starts. This is where the role of GUI comes. From the vast information provided by the logs and test files, interfaces have to capture and represent relevant information to the user. The following section details the implementation and features of the proposed GUI.

5.1   INTERFACE IMPLEMENTATION

GUI is implemented as a HTML web page. Each test case will have its one set of log files and asm list files. The interfaces implementation starts by taking these files are input. 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.

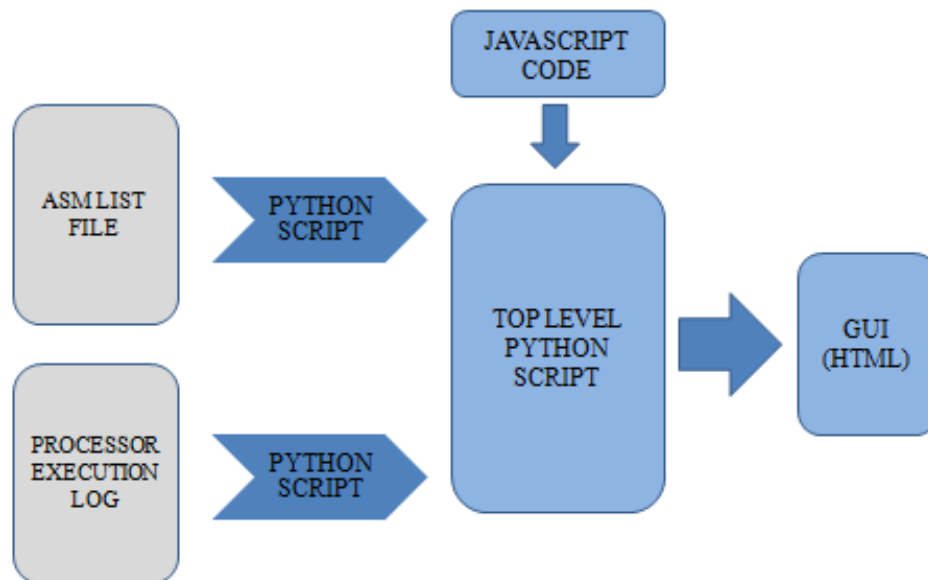Figure x shows the implementation of GUI.

Figure 5.1: GUI Implementation

Major implementation steps

- The asm list file and processor execution file informations are extracted by python script.

- A top level python program will manipulate these information are convert it to JavaScript objects.

- JavaScript code for developing user interactive features is written.

- The top level python program will combine the data extracted and javascriot code and generate a single HTML page.

Inputs to the implementation are Execution log file and Asm list files. To develop the GUI the user has to execute the top level python script with these two files as input argument. The script will generate an output file which is the interface web page. Major processes involved in implementation are detailed below.

5.1.1 EXTRACTING ASM FILE INFORMATION

The test code is written in x-86 assembly language. While debugging the instructions and values in the asm file is the expected action or value. Each cycle in processor execution log corresponds to a particular code line in asm. However the asm code written by the verification engineer is the unscheduled code without any memory address details. For cycle comparison with execution log, the asm test file is complied first. This is done by an assembler. Figure x shows how an asm test file is converted to a list file which hold an expanded, loop unrolled, scheduled code with memory details included.
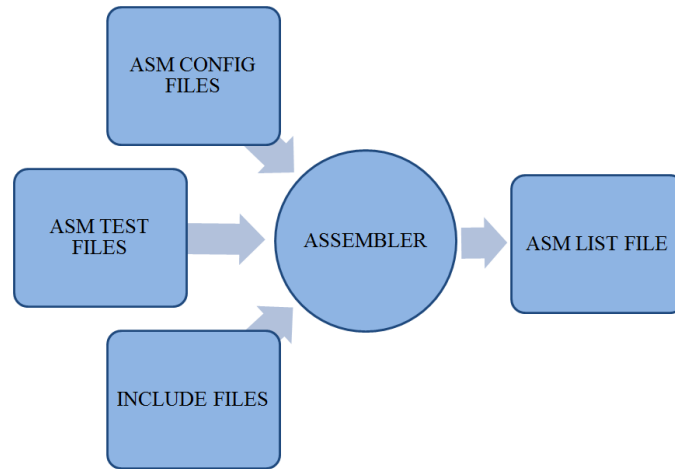
Figure 5.2: Assembler

A 64-bit assembler assembles asm test file, configuration file (that define the random operands, segmentation, gdt, ldt, page tables, etc,) and the include files together to generate an assembly list file [1].

Figure 5.3: Asm List File Extraction

List file holds details of instructions; opcodes and operand, linear address, module/register configuration details etc. For the purpose of correlating informations, a python script will extract relevant information corresponding to each instruction line in

the list file. For each line number the script will create an object which will have the following parameters.

- Base and offset address value

- Instruction line number

- The assembly code

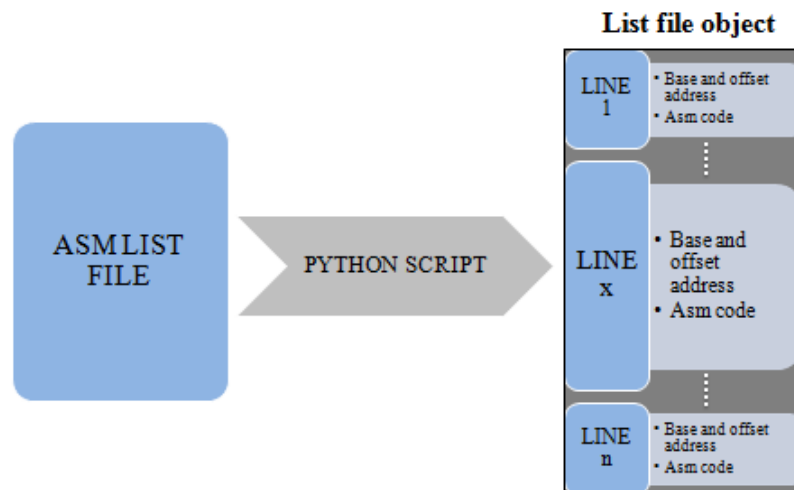Figure x shows how python script will read the input asm list file and generate a data structure of list objects corresponding to each line.

5.1.2 EXTRACTING EXECUTION LOG INFORMATION

The second input to the implementation is the execution log file. As explained in previous chapters, debugging require detail traversal through this log file as it holds instruction by instruction execution details which include register states, thread details, flags information etc and other data which help in tracing out the cause of failure.

All details given in the log file in reference to each cycle are to be extracted. The python code will extract this as well are generate a data base of log file object corresponding to each cycle of execution. These are classes with properties corresponding to major operations and register values. Each thread will have a separate handle with objects corresponding to each cycle of operation in that thread. Figure x shows how the script will read the input execution log file and generate log file objects. Each object in the log file will have the following properties associated with it:

- Thread number

- Mode of operation

- Cycle number

- Linear address

- Memory write, Memory read, I/O write/read, Code read

- Branch target (linear address)

- Registers updated with new value

5.1.3   Correlating asm file objects and log file objects

Once both the input files are processed, next step is correlating the asm file object and log file object. The top level python script will combine these these to in pairs based on the address mapping. As x86 architecture follows segmented memory model along with paging, address translation is required for generating the linear/physical address [2]. For each object in the generated list file class, calculate the linear address as follows:

Linear address = Base address + Offset value

For each object in the log file class, find the corresponding list file object based on the linear address value and combine the objects.

    Input : list file objects→ *listOb j*[], log file object→ *logOb j*[]

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 → *log*.property

               `// log properties are lineNo, opcode and address`

13   count = 1; break loop

14   **end**

15   **end**

16   **if** *count == 0* **then**

17   assert: "*noaddressmatch*"

18   **end**

19 **end**

20 End

Algorithm 3: Combining List and Log File Information

Now we have a set of objects that hold log details and are linked to corresponding asm file line. Once this stage is completed we have completed all the data extraction and next phase is building the interface.

5.1.4 DEVELOP THE INTERFACE

The final step is developing the interface. As the GUI is web based, layout design is done using HTML (Hyper Text Markup Language). However for providing interactive features to the user a much more powerful language is need along with HTML and we use JavaScript.

JavaScript (JS) is an interpreted computer programming language. It is implemented as part of web browsers so that client-side scripts could interact with the user, control the browser, communicate asynchronously, and alter the document content that was displayed. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.

In addition a style sheet language called CSS (Cascading Style Sheets) is used for describing the presentation semantics (the look and formatting) of the interface page written in HTML.
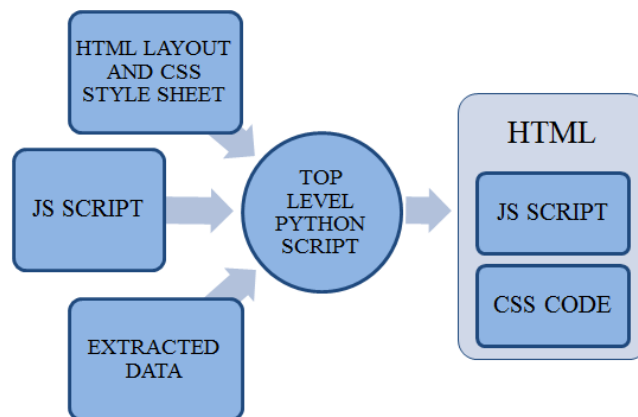

Figure 5.4: Web Page Generation

The JavScript program and the style sheet will we embedded inside the HTML code. This connection is done by the top level python script. Figure x shows the schematic of how the files are combined with the data.

CREATING JS OBJECTS

All the dynamic features of the interface is handled by JS script. The data input to the JS is the extracted information from log files andlist files. The top level python script

will convert the extracted objects into JS data array "*dataArray*[]" ; where each array element coresponds to an active thread. Active threads refer to active core or processor in a multiprocessor system.

```
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 4: Creating JavaScript Object

Once the extracted information is converted to JS compatiable data, the interface features can utilize this. The layout for various GUI features called windows, are designed using HTML and CSS code. All the dynamic interactive featurers are handled by the JS script embedded in the final HTML web page. The following section introduces the differend windows developed for the GUI.

5.2 GUI FEATURES

5.2.1 EXECUTION FLOW GRAPH

Main feature of the web page is the graph showing the execution flow of the code. Here asm list file line numbers are plotted against the cycle during which it is executed. All the active threads have different graphs which are tabbed. Hovering the mouse over any point on the graph will display x and y axis values.

This zoom enabled data graphs also provides onclick selection of specific operations

e.g: Branching, Memory Write, Memory Read, Code Read etc, which will display the instances of selected operation. Each operation is distinguished by its color.

Implementation

For the developement of execution graph we are using Dygraph JavaScript Visualization Library []. This will provide inbuild functions that enable zooming, x-y axis value display and point onClick call backs. onClick CallBack() function is called when ever a point on Dygraph is clicke and this function actives many other window.

For building the graph, input to the Dygraph is independent axis values followed by depended axis values. For each thread, a different graph is generated. The x-axis is cycle number and y-axis is the list file line number. Algorithm x explains hw the graph is build.

```
 1  CreateGraph()

 2  begin

 3      for each i in activeThread do

 4

 5          for each element in dataArray[i] do

 6

 7              Dygraph[i] ← [element.cycleNo, element.lineNo]

 8          end

 9      end

10  end
```

Algorithm 5: Creating Execution Graph

5.3 REGISTER WATCH WINDOW

Register window capture and display updated register values at a specific instance. Each thread holds its own copy of registers/flags. A selection of point on the execution flow graph will update the register window with values at that instant in the selected thread. Values of following registers and flags are provided to the user:

- 64 bit general purpose registers (RAX, RBX, RCX etc)

- RFLAG (64 bit)

- Instruction Pointer (RIP)

- Stack Pointer (RSP)

Another feature provided by register window is comparison between register values at two different instances that is between a reference point set by Set Marker button and current selection.

Implementation

This window is activated by the point onClick CallBack function evoked when a point is clicked on Dygraph. The function will update the register window rows set by the HTML code eg. regRow[RFLAG], regRow[RBX] etc. Also there will be a comparison with the reference point set.

```
1 pointonClickCallBack(element)
2 begin
3     for each reg in RegisterSet do
4         regRow[reg] ← element.[reg]
5         if (element.[reg] != referenceRow.[reg]) then
6
7             Highlight regRow
8         end
9     end
10 end
```

Algorithm 6: Creating Register Window

5.3.1   INSTRUCTION WINDOW

Instruction window give the asm file lines. A selection in execution graph will be reflected in this window by highlighting the asm file instruction corresponding to the selected point. Also the context of the selected line that is its preceding and succeeding instructions are also available in this window.

Implementation

This window is also activated by the onClick CallBack function evoked when a point is clicked on Dygraph. The function will update the instruction window set by the HTML code.

```
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

5.3.2 EXECUTION LOG WINDOW

In addition to the instruction and register information, all the processor execution log information regarding the selected instruction is also provided through execution log window. This collected information is stored as a log file object propert "*l*ogInfo".

Implementation of Execution log window is given in algorithm x along with instruction window.

5.3.3 SET AND CLEAR MARKER

These two options allow setting or removing a reference point with which current register values are compared against.

Implementation

Set and Clear buttons are set using HTML form options. These elements have onClick call backs. On clicking Setbutton call back will set add values to reference register section and on click clear will remove the values in reference section.

Chapter 6

RESULTS: GRAPHIC USER INTERFACE

The final GUI is a HTML page generated by the master phyton script. This file is generated by running master script with the asm list files and execution log files as input. The generated .html or .htm file can be viewed by any standard web browser like Internet Explorer, Firefox etc that support JavaScript. As this is just like any other html page with out any dependences, the user can access the page from web as a stand-alone file with out requiring the list files and log files. The following figures shows various windows of final GUI.
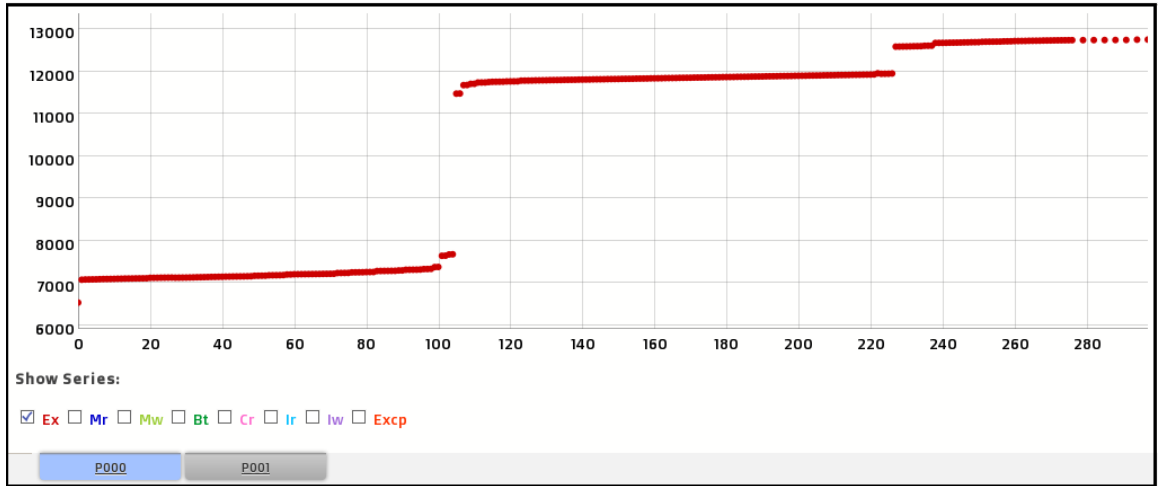
6.1   EXECUTION FLOW GRAPH

Figure 6.1: Execution Flow Graph

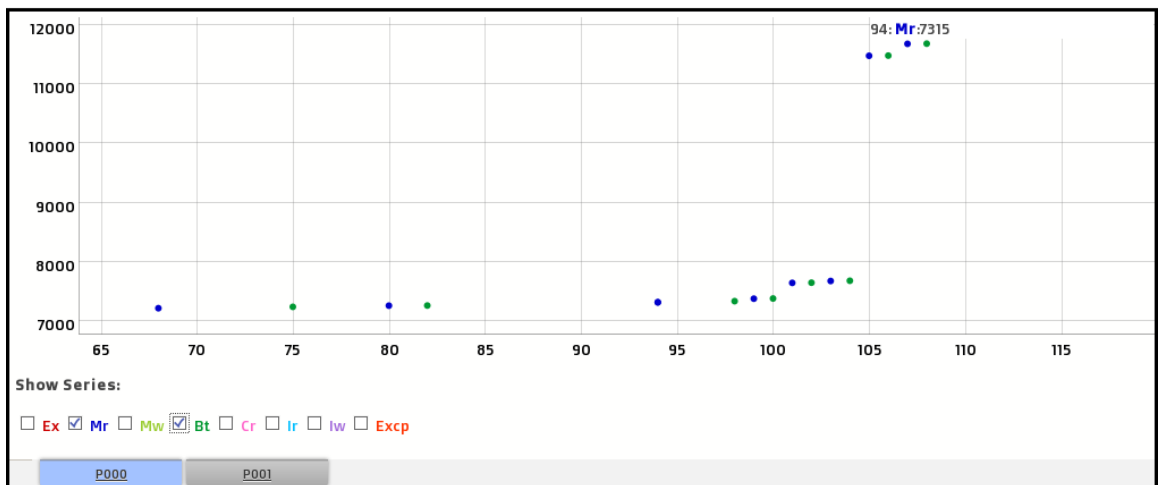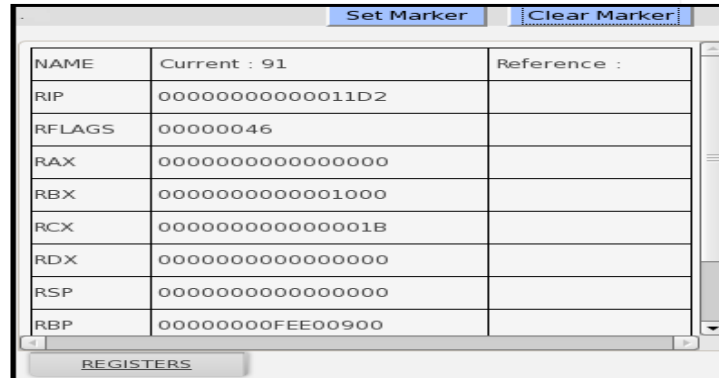Figure 6.1shows the main execution flow graph for selected active thread.

Figure 6.2: Execution Graph With Branching and Memory Writes

Figure 6.2 shows only memory write and branch operation happening in the selected thread.
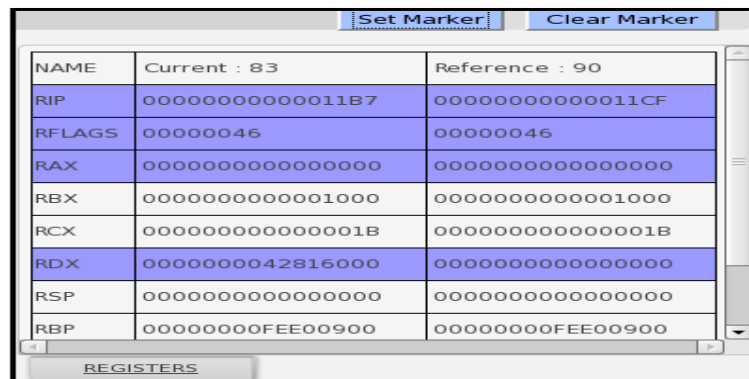
6.2   REGISTER WATCH WINDOW

Figure 6.3: Register Watch Window

Figure 6.3 shows register window showing updated register values of current selection.
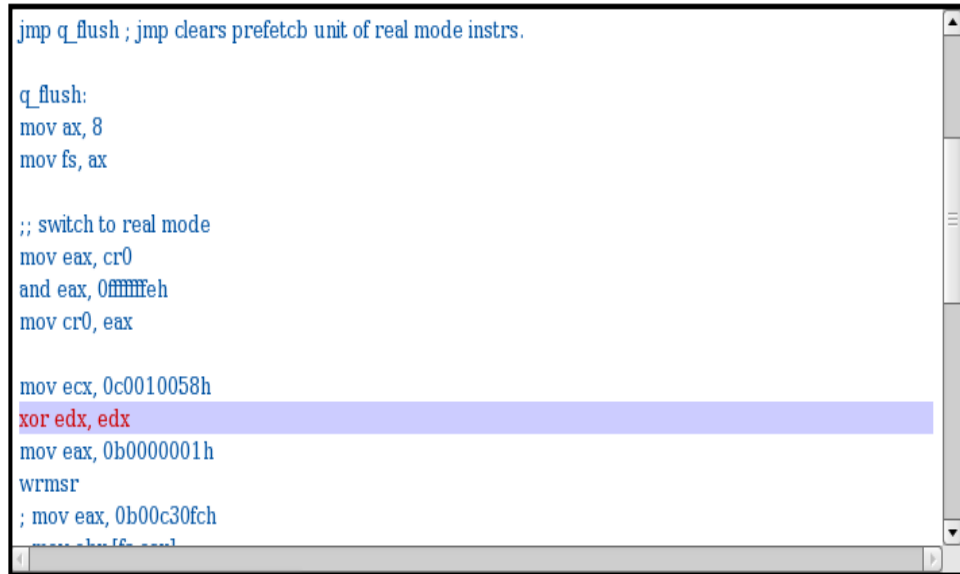
Figure 6.4: Register Value Comparison

Figure 6.4 shows the comparison of register states at two different points. Differences are highlighted.

6.3   INSTRUCTION WINDOW


```
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, 0fffffffeh
mov cr0, eax

mov ecx, 0c0010058h
xor edx, edx
mov eax, 0b0000001h
wrmsr
; mov eax, 0b00c30fch
```

Figure 6.5: Instruction Window

Figure 6.5 shows the instruction window which highlights the current selection and also display the context of selected instruction.

Chapter 7

CONCLUSION

An interactive off-line graphical user interface was designed and developed. The interface is a web based html page having all the relevant information from processor execution log and asm test files. The execution graph helps in analysis of various operations like branching, exception, read/write etc. This graph sort instruction execution information into corresponding thread making tracing rather easy. The register window provides comparison between registers and flag values at two different execution cycle which helps in tracking a wrong value written into register or flag. Instruction window and execution log window provide all the detailed information that could be obtained from list file and execution log file regarding the cycle selected from the graph.

Potential for future work: The interface tries to gather and represent data for easy debugging. There is scope for introducing more debug features like break points.

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 availabilty 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 primities 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 layed-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 pos-layout netlist as it becomes available. Figure 9.1 depicts progress of gatesim with respect to other design flows.
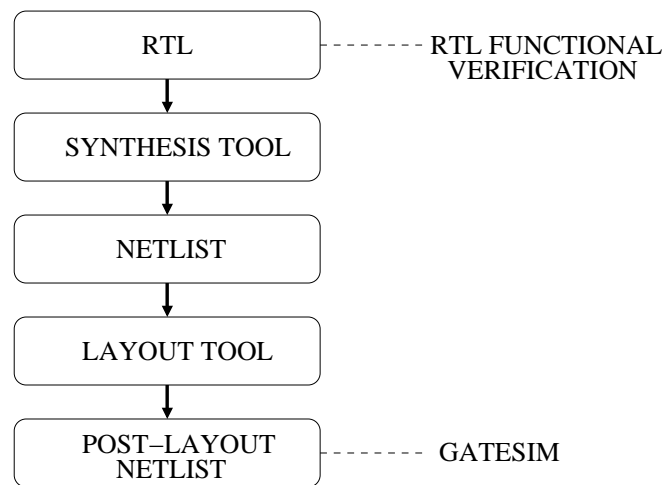
Figure 9.1: Design Flow

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 with gates rather than at RTL abstraction. Verification with gates is a huge confidence booster before manufacturing of actual silicon as they are also thought to complement and cover gaps left by 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 late design, methodology or process issues.

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 Checking (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 descriptions. Although LEC uses sophisticated formal algorithms to identify, map, and compare nodes in the netlists, the complexity is hidden from the user.

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 aspects of required verification on netlist. 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-handling:** STA deals only with logic domain of logic-0 and logic-1. An X in the design cause undetermined values to propagate through the design. This cannot be checked with STA.

- **Interfaces between analog and digital blocks:** STA does not deal with analog blocks. And hence cannot verify connectivity between digital and analog blocks whereous Gatesims can.

- **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 correct clock synchronizers are being used.

9.3 ISSUES CAUGHT BY GATESIM

The following are design issues missed initially but caught by gatesims:

1. **X Squashing**

   X-Squashing is a term used when X-es get squashed in a simulation and don't propagate anymore through the logic. In one case there was an X-Squashing issue in behavioral RTL which led to a valid value being present in one of the RTL outputs during simulations, where as in gates X wasn't squashed and it propagated to the corresponding output resulting in a simulation mismatch between RTL and gates.

2. **Reset X problem**

   Some of the un-initialized flops resulting in X issues were easily found during GLS. After identifying such scenarios appropriate forces were added as part of Gate simulation flow.

3. **Wrong connectivity during block level mapping**

   During integration, sub-blocks at top level may get connected incorrectly due to naming issues at the sub block level. Sub-block level LEC would not catch this issue, whereas gatesim flags the wrong connectivity.

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. The first step would be to obtain the test vector stimulus to the netlist. One approach could be to reuse the RTL testbench around the netlist. Another approach could be to replace only a portion of circuit with netlist in the existing RTL verification environment.

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

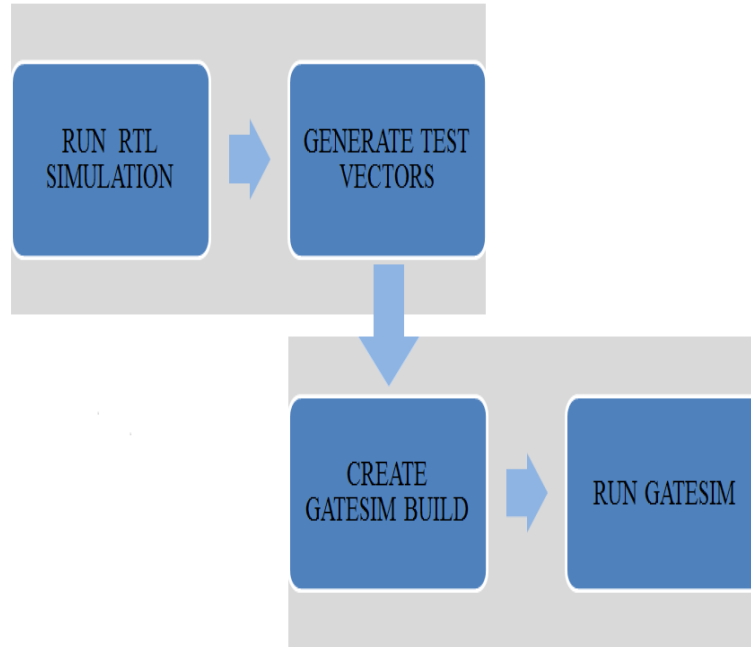as stimulus and comparison was done with the RTL output vectors. Figure 10.1 shows the simulation flow.

Figure 10.1: Early Dual-Sim Flow

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

Issues: 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 information of stimulus. These files were large and simulation performance was also affected by disk input/output accesses. Another shortcoming of this methodology was that when stimulus was converted to cycle based information, inherent sampling errors were introduced, which were themselves causing simulation mismatches when compared with RTL simulations.

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

adopted instead.

10.2   CO-SIM BASED GATESIM METHODOLOGY

Cosim-based approach was conceived to solve some problems that existed with dual-sim approach by allowing easy debug while maintaining common input vectors and output results for comparison. Figure 10.2 show how stimulus is appiled to netlist and comparison of output is done in co-sim method.
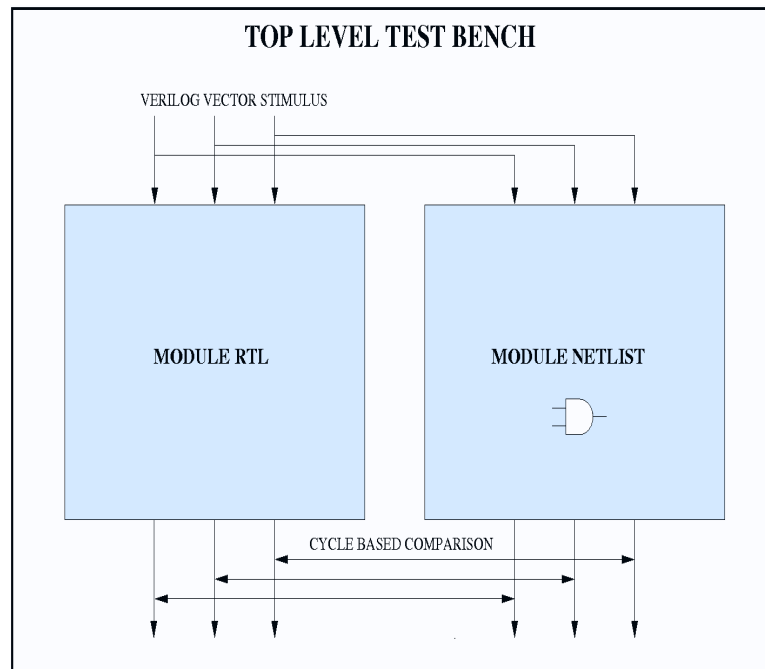

Figure 10.2: Co-sim Based Gatesim

A single combined executable containing the behavioral RTL design, the gate design, and all of the required simulation files is compiled. The behavioral RTL and gate models are run in lock-step with their inputs tied and driven by the testbench and the comparison of the behavioral RTL and gate outputs is done "on the fly". Figure 10.3 shows co-sim flow.
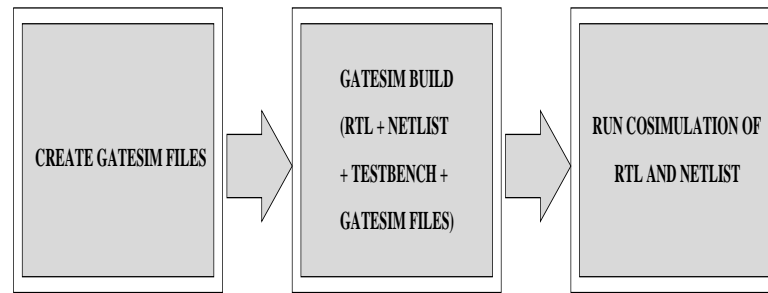
Figure 10.3: Co-sim Based Gatesim Flow

Major steps involved in this flow are:

1. **Getting gatesim files**

   Input to gatesims are files obtained from LEC tools. These file include 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 called gatesim files and these verilog files are:

   - File which contain top level module that connects the behavioral RTL signals to the corresponding gate signals.

   - Test bench compare files that contain the verilog compare code used to compare the outputs and register mappings.

   - Force files that contain all the force/release/assign commands for the gates corresponding to RTL force/release/assign statements.

2. **Getting gatesim build**

   Next stage is to enable a build structure supporting the co simulation of RTL and Netlist. The build infrastructure will include the following:

   - Netlist

   - RTL

   - Test bench

   - Gatesim files

3. **Run co-simulation of RTL and netlist**

   Finally, run co-simulation. Output files include <testname>.out which contains the simulation log of the entire test and <test name>.fsdb (if wave form dump enabled).

As the netlist stimulus is obtained from a live RTL instead if from storage files, co-sim based gatesim overcame the biggest limitation associated with dual-sim. Along with some good set of scripts aiding testbench generation and force generation, this method became the standard method for gatesims ever since.

10.2.1   ISSUES WITH CURRENT CO-SIM METHOD

Co-sim based Gatesim overcame all the known limitation associated with early approach but brought in new set of limitations with ever growing design complexity. Main limitations are:

- Larger turn-around time (run, debug cycle).

- Limitation on size of netlist (indirect cause due to larger build and run time).

Simulation performance analysis

Experiments showed that the simulation performance of gatesims was affected sometimes as low as 10% with respect to its counterpart RTL simulations. This indicates that:

- RTL Simulations contribute major to CPS than netlist.

- Simulator spends more time in simulating RTL and verification components than netlist.

Main causes for this poor simulation performance are:

- Test vectors are applied from a live RTL simulation, which runs redundantly for the sole purpose of generating test vectors.

- Verification components in modern designs (C, C++, SVTB, OVA, SVA), slow down simulation performance.

Evidently it was not an appropriate use of compute resources by having live RTL simulation every time just for the 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 the cause was bulky test-bench components associated with RTL simulation(C, C++, SVTB, OVA, SVA), which slows down simulation performance considerably. 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 source of test vector. The proposed solution should also improve on simulation performance by:

- Ensuring a compact method to store test vectors, as it directly affects the storage requirements.

- Avoiding re-build for different test vectors, as it affects the turn-around times and storage requirements.

FSDB or Fast Signal Database is signal data file similar to VCD 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 Verdi[3] tool set for FSDB format could be used to retrieve values from FSDB. PLI/VPI[1] could be used to drive stimulus onto netlist.

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. Following second simulation has 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.
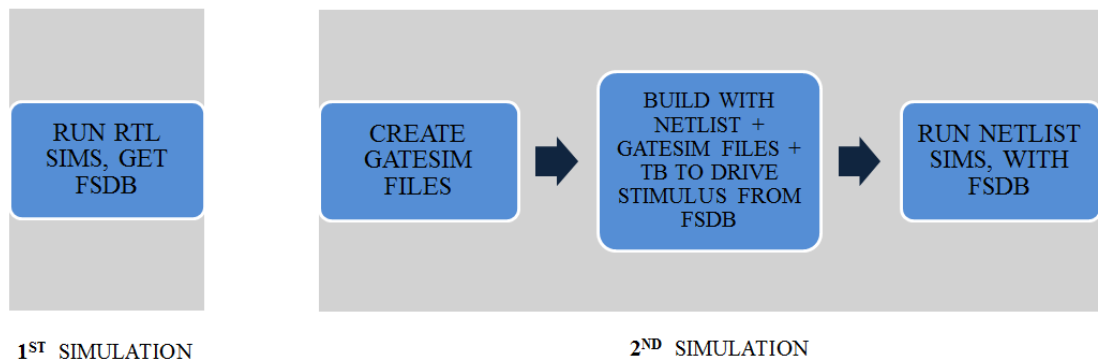

Figure 11.1: Dual Sim

Major steps involved in this flow are:

1. Run RTL simulation

   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. So the first step is to perform a standard RTL simulation with testbench components. 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, high-capacity 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 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 VCShave command to enable "*d*ump" during simulation run. This will dump all the *s*ignals involved in RTL simulation and these FSDB dump files are used by waveform viewers as signal source.
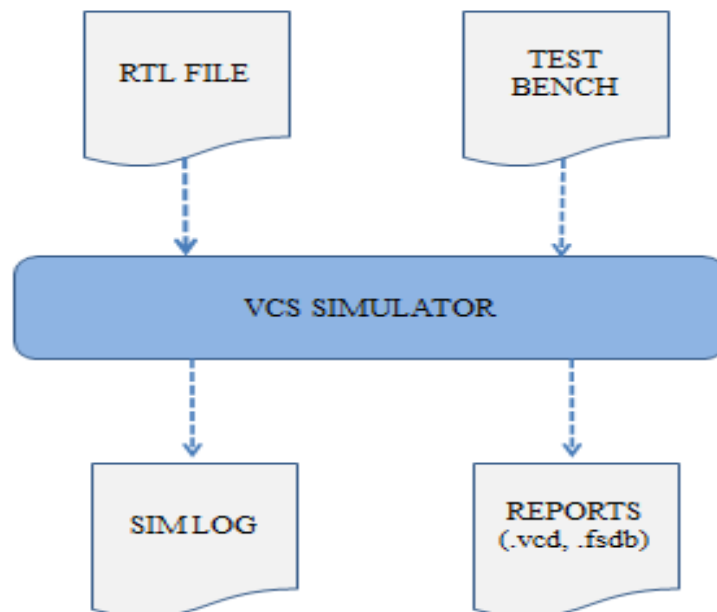

Figure 11.2: RTL Simulation

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 before 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 x shows the layout of programming infrastructure developed for this.
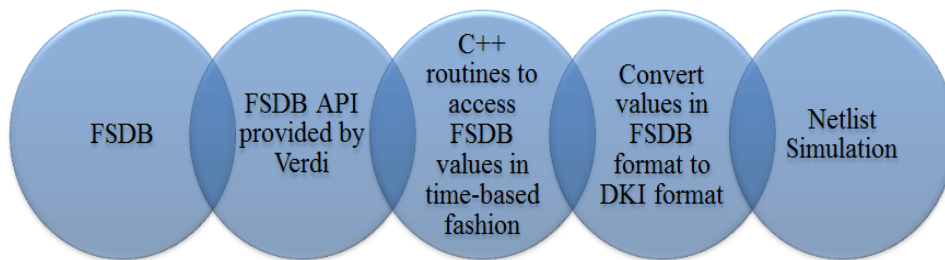

Figure 11.3: Accessing FSDB Signals

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.
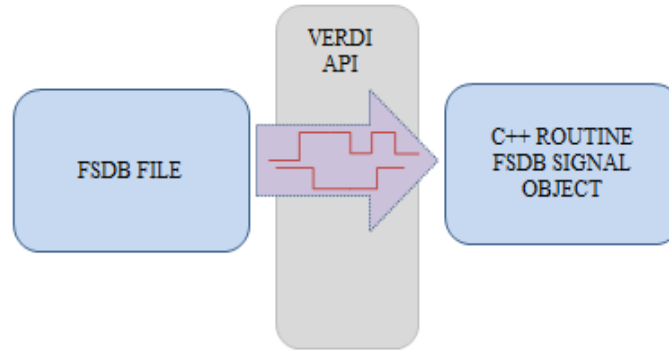
Figure 11.4: API For Accessing FSDB Signals

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 "*A*ttach()" routines are provided for signal attach with objects.

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.
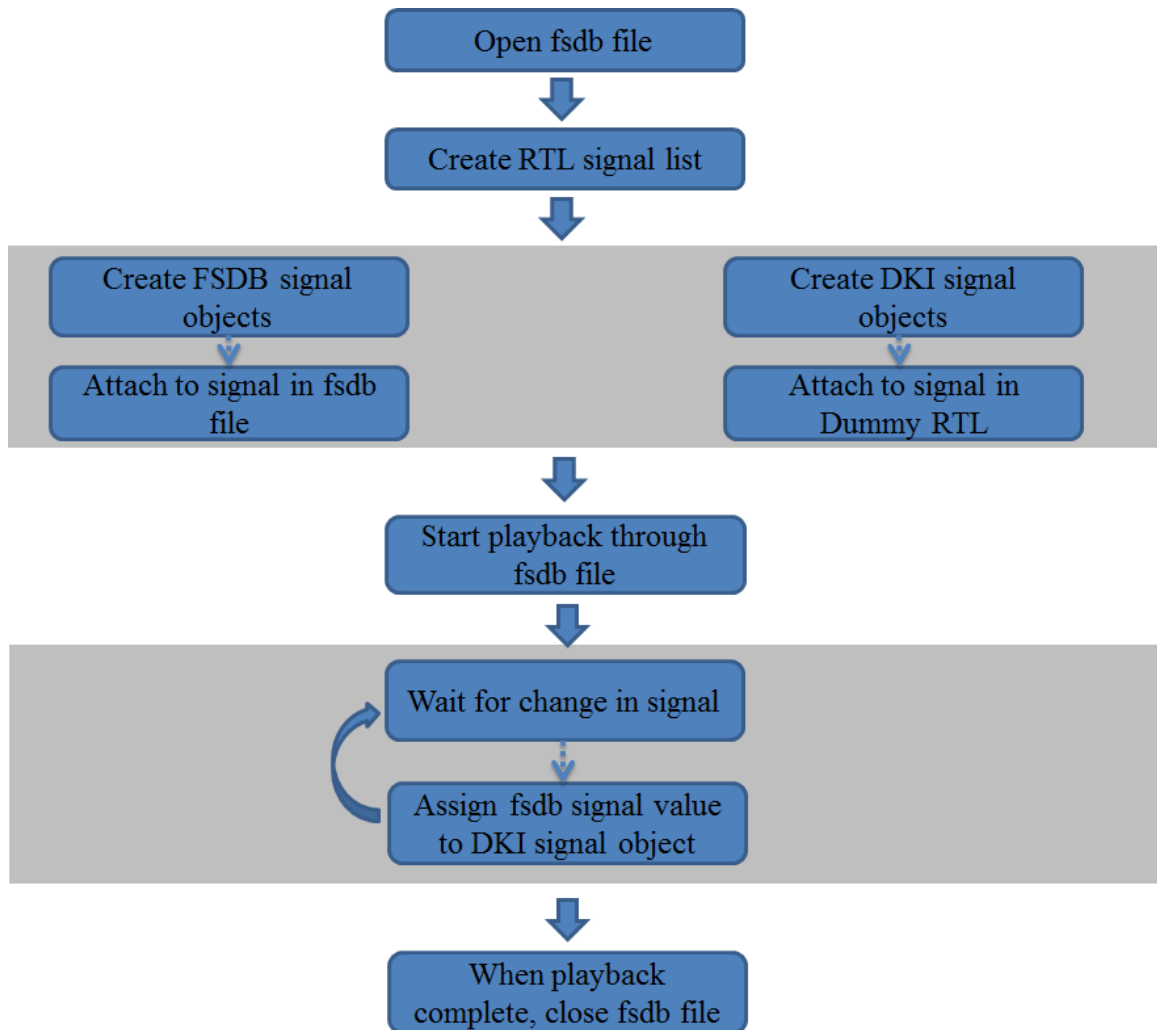
Figure 11.5: C++ Routine for FSDB Signal Extraction

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.

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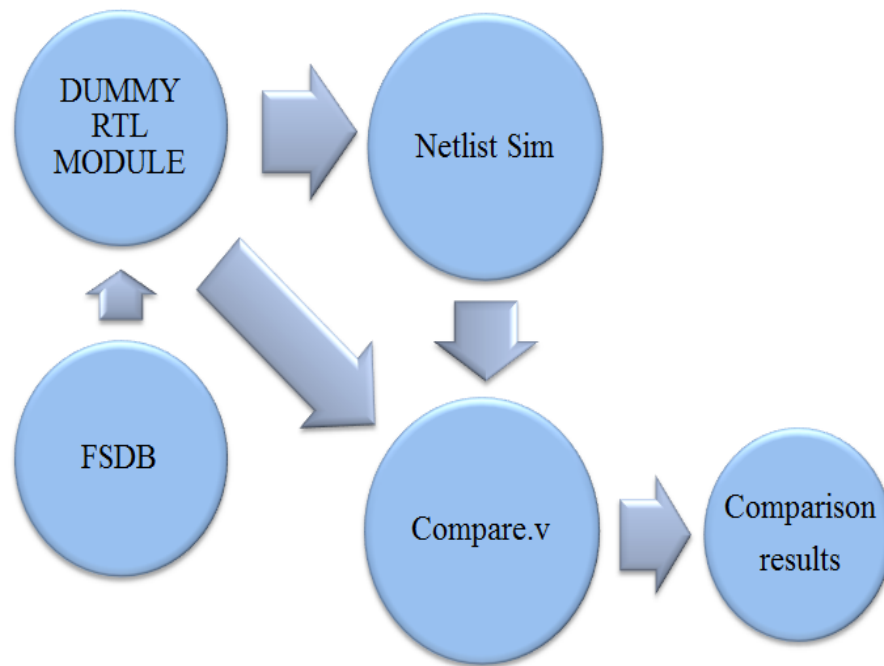
11.2.3 NETLIST SIMULATION FLOW

Figure 11.6: Netlist Simulation

Figure **??** show how test vectors are applied onto the netlist and how final comparison is done. Features of Improved gatesim methodology are:

1. Obtain test vectors from FSDB; access it in time-based fashion.

2. Drive stimulus onto a dummy RTL. This dummy RTL does not have any logic

other than Input/output. All the input and output ports of the module netlist are included in this dummy RTL module.

3. Dummy RTL drives netlist stimulus and other stimulus points inside design (like fuses, un init flops).

4. Apply appropriate force signals from FSDB directly on to the dummy RTL. The dummy RTL signal forces the netlist signals.

5. Accomplish verification goals by cycle-comparing netlist behavior to that of its counterpart RTL (as available in the FSDB).

It should be noted that the improved methodology is still dependent on vectors from RTL simulations and is not independent by itself. But gains obtained in turn-around times were of great benefit.

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.

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] SpringSoft. (2013, May.) Verdi Automated Debug System. [Online]. Available: http://www.springsoft.com/products/debug-automation/verdi

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