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	**Prof. M. S. Bhat**	**Prof. Muralidhar Kulkarni**
Project Guide	**Project Guide**	**Chairman DPGC**
(External)	**(Internal)**	

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. Moreover, manual tracing is error-prone and time consuming. This project aims to implement a GUI enabling effortless and faster 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.

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 gatesim methodologies are in use across the industry. In co-simulation methodology of gatesim verification, 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 itself. The thesis proposes a new dual simulation approach to Gatesim where RTL and gate level simulations are performed separately, by exporting test vectors from RTL simulation to netlist simulation. This improved method attempts to overcome performance issues with current co-simulation methodology.

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

Contents

List of Figures

List of Tables

List of Algorithms

Listings

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 pre-designed blocks called semiconductor Intellectual Property (IP). These hardware IP blocks can be mix-and-matched, thereby providing design reuse in SoC and thus reducing time-to-market (Brackenbury, Plana, and Pepper, 2010).

Over the past few years, major challenges faced by semiconductor industry has been 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 (Kuhn, Oppold, Winterholer, Rosenstiel, Edwards, a 2001). 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 has been widely accepted that verification consumes more than 70 percent of design effort (Zhang, 2005). 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. Even then 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 methodoloogy, the verification engineer develops a set of tests based on design specifications. Design correctness is then established through simulations. On the other hand formal verification is a mathematical proof method of ensuring that a design's implementation matches its specification (Segev, Goldshlager, Miller, Shua, Sher, and Greenberg, 2004). The most prominent distinction between these two methods is that while stimulus based method requires input vectors, formal methods do 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. Here, user need not generate stimulus but specification of invalid stimulus would be necessary. However formal methodology tools have large memory and long run time requirements. When memory capacity is exceeded, these tools often shed little light on what went wrong, or give little guidance to fix the problem. As a result, formal verification softwares are best suited only to circuits of moderate size, such as blocks or modules. Since SoC level designs are huge, these are typically beyond the capacities of automatic formal tools (**?**).

1.2 SoC VERIFICATION

Most SoCs are built around one or more processing cores (multiprocessor) 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 verification engineer 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 Environment* discusses existing verification methodologies and their shortfalls.

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

Chapter 5-*Interface Implementation* gives implementation details of proposed enhancement.

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

Chapter 13 discusses the various conclusion drawn from the results and the scope for future work.

Chapter 2

AMD64 ARCHITECTURE OVERVIEW

Typically a SoC brings 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 SoCs are very complex designs which includes programming elements, hardware elements, software elements, bus architecture, clock and power distribution, test structures etc.

The heart of any SoC design is a 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. Majority 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) (AMD, 2011). 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.

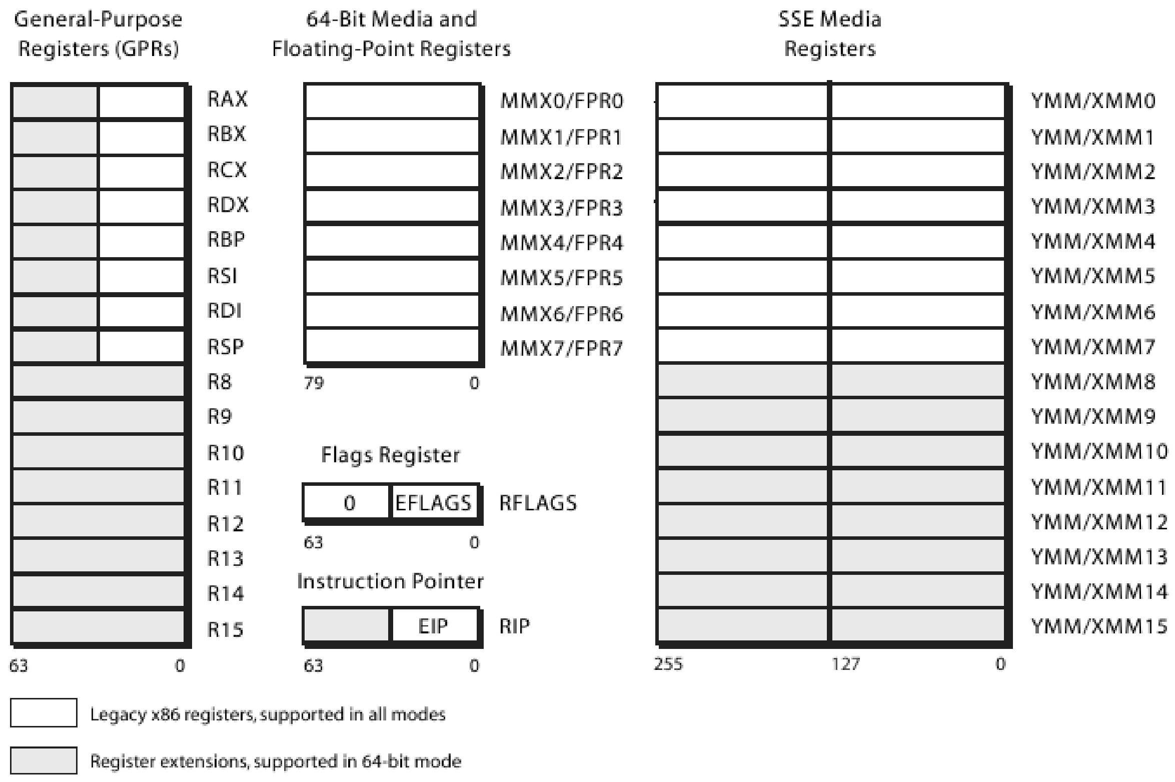


Figure 2.1: AMD64 Registers

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 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 (Corporation, 2011). 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 2.2shows segmented virtual memory.

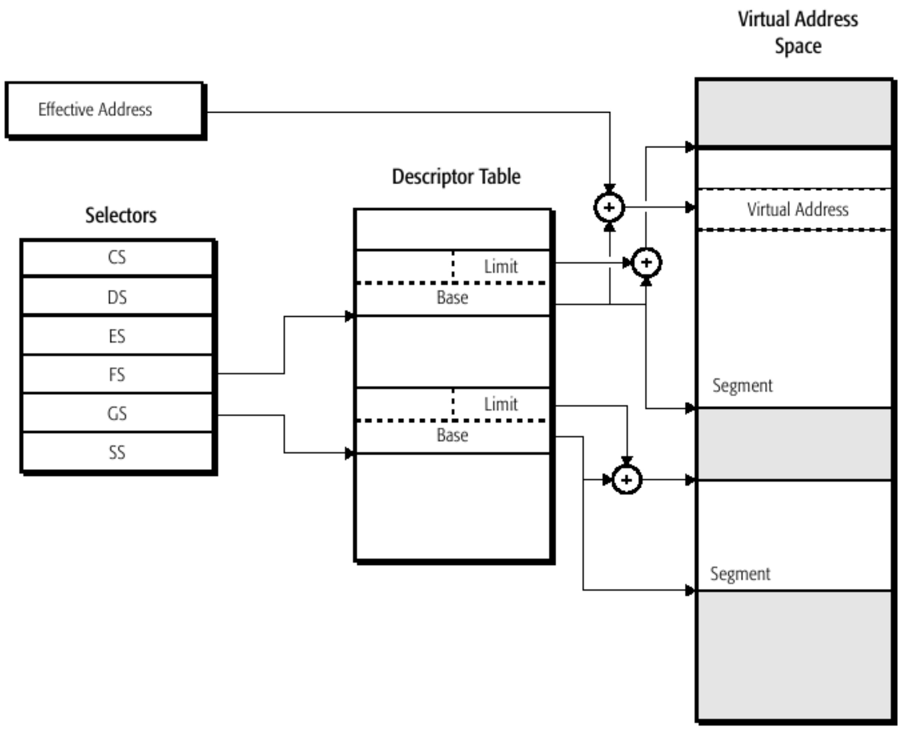


Figure 2.2: Segmented Memory Model

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.

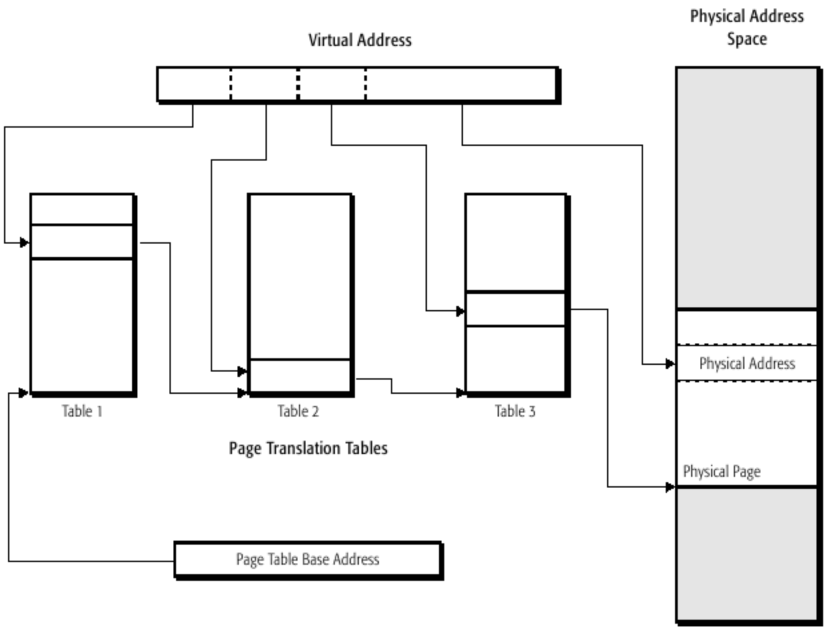


Figure 2.3: Paged Memory Model

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 (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. System level verifications are 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.

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 the IPs and functional verification of top level modules are done. Test conditions are written in x86 assembly or 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, the engineer 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 manly 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 testbench 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”.

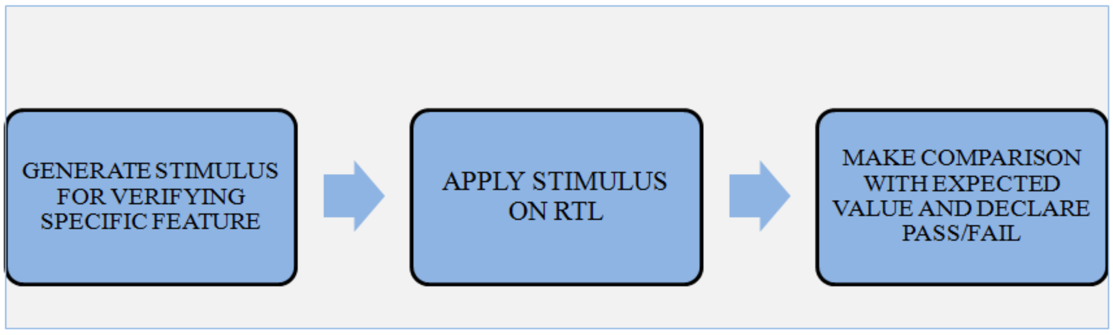

GENERATE STIMULUS FOR VERIFYING SPECIFIC FEATURE
APPLY STIMULUS ON RTL
MAKE COMPARISON WITH EXPECTED VALUE AND DECLARE PASS/FAIL

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 cosimulation and produces a log of processor activity.

3.3.1 COSIMULATION

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.

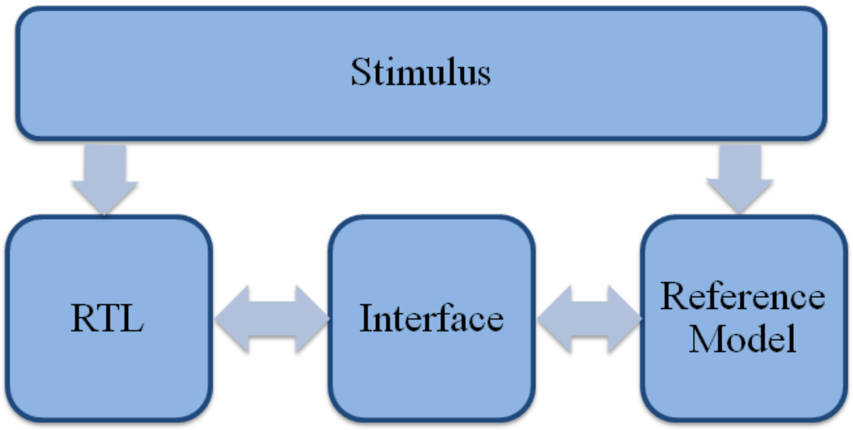


Figure 3.2: RTL-Reference Model Cosimulation

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 multithreaded code to simulate multiprocessor and multicore 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 *memory mismatches* and upon any *mismatch*, the cosimulation 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.

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.

The *processor execution logs* generated by ILS, during the course of simulation, contains cycle by cycle information regarding register states, memory values, flags, threads etc. All the comparisons and debugging requires these information.

Once a failure is reported, simulation terminates and and debugging is required. 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 *processor execution log* and the assembled test files needs to be analyzed and correlated 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. Correlated information is spread across in different files.

As the design itself is very complex, these reasons makes manual tracing too time consuming. This stretches 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 having 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

```
   Input  : $data, address$
   Output: Test result: $pass$ or $fail$
1  Initialization: Select memory bank by setting $ControlRegister$
2  $RegA \longleftarrow data$
3  $RegB \longleftarrow address$
4  Memory $[RegB] \longleftarrow RegA$
5  $RegD \longleftarrow$ Memory $[RegB]$
6  if $RegA == RegD$ then
7      report $pass$
8  else
9      report $fail$
10 end
```

Algorithm 1: Memory Read-Write

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 would be from wrong memory location, leading to a 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 *RegD* chosen is unavailable in this mode of processor operation. This causes the wrong data to be updated into the memory and comparison fails.

Debug Process

In Case 1, the memory bank selected should stay same throughout the program execution. Any change in memory bank selects would cause the read operation to take value from wrong memory block, leading to a self-test failure. In Case 2, 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, 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.

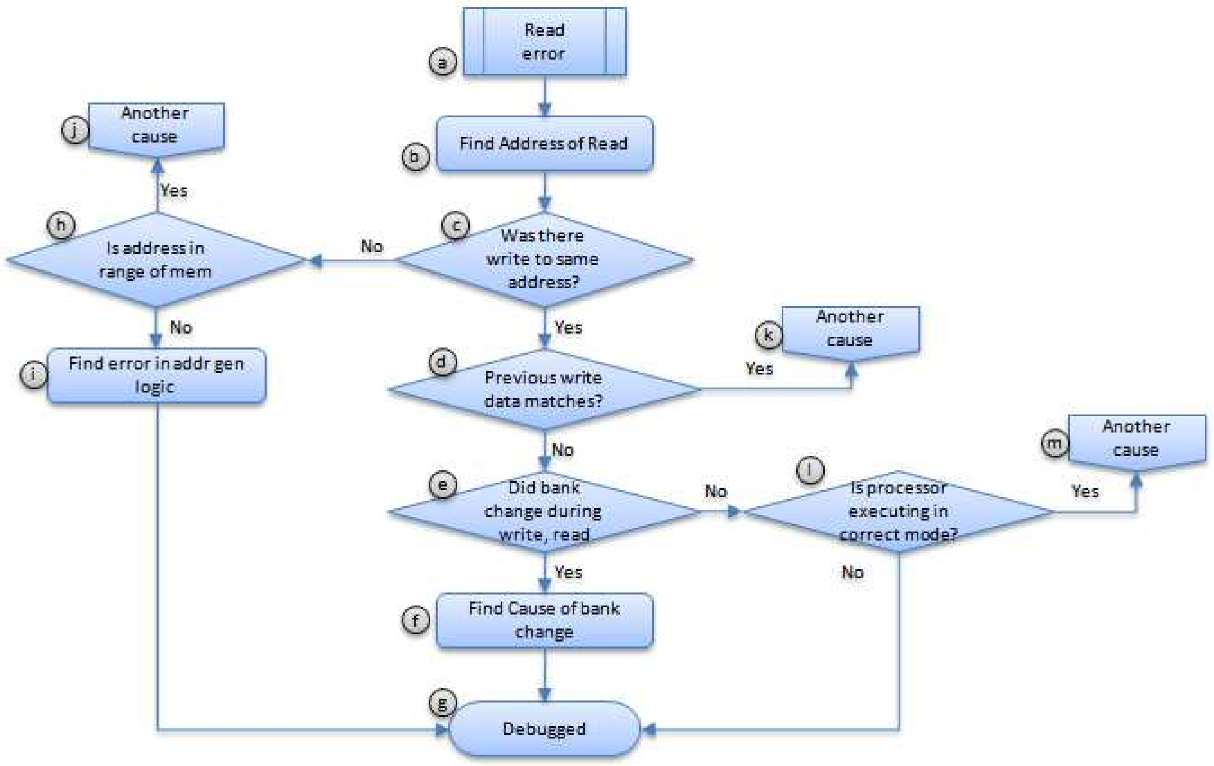
Figure4.1 depicts a typical debug process. Note that there could be other causes for failure as depicted in *states*ⓙ, ⓚ, &ⓜ. During debug, the engineer needs access to certain information as listed here:

1. *state* ⓑ: Examine Read to address, examine data obtained.

2. *state* ⓒ: Obtain all accesses occurred to same address, find the latest write to same address.

3. *state* ⓓ: Examine data written to latest write to same address.

4. *state* ⓔ: What is the bank select location? Did it change between write and read?



a
Read error
b
Find Address of Read
c
Was there write to same address?
No
h
Is address in range of mem
Yes
j
Another cause
No
i
Find error in addr gen logic
Yes
d
Previous write data matches?
Yes
k
Another cause
No
e
Did bank change during write, read
No
l
Is processor executing in correct mode?
Yes
m
Another cause
No
Yes
f
Find Cause of bank change
g
Debugged

Figure 4.1: Illustration of debug process : Test A

5. *state* ⓕ: Which program code changed bank select?

6. *state* ⓛ: What is processor mode?

7. *state* ⓘ: 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     $c \longleftarrow c + CONST$
3 end
```

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 would lead to an invalid conversion.

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

Debug Process

In Case 1, 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, the resulting string after loop exit would normally have characters unmodified towards the end. This case could be identified with such symptom.

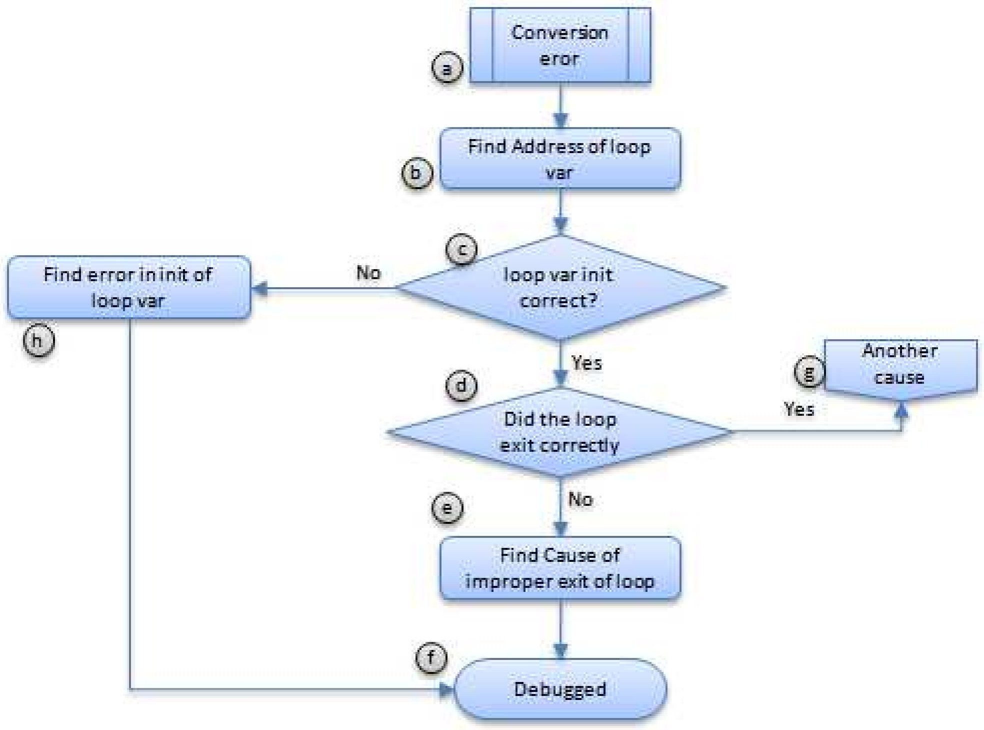


Figure 4.2: Illustration of debug process : Test B

Figure4.2 depicts a typical debug process for this case. Note that there could be other causes for failure as depicted in *state* ⓖ. During debug, the engineer needs access to certain information as listed here

1. *state* ⓑ: Find address of loop var.
2. *state* ⓒ: At start of loop, check initial value of loop var.
3. *state* ⓓ: When the loop exited, what was the value of loop var.
4. *state* ⓔ: Need computation values of loop exit condition.
5. *state* ⓕ: 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 evident that availabitily of correlated information during debug process would enhance it in a big way. In this thesis, we are proposing a Graphical User Interface (GUI) which would 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 helps to get rid of traditional method of comparing Instruction Pointer Register (RIP) values and string search by providing graphs that would 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. In this case, if there exists a visual representation of execution control flow, it would be obvious if a loop was 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 etc are readily accessible.

Processor State Information Change: It would aid debug if it is 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
- 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 accommodate any relevant assembly test, accompanied with execution logs.

5.1.1 CHOOSING GUI

There are multitude of languages providing GUI capability, the project does not need a very sophisticated GUI implementation. Moreover the interface should aid remote debug and if possible there 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 *JavaScrips* and that makes it customisable. The design is also scalable and aids introduction of new features for debug with ease.

Each stimulus has 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.

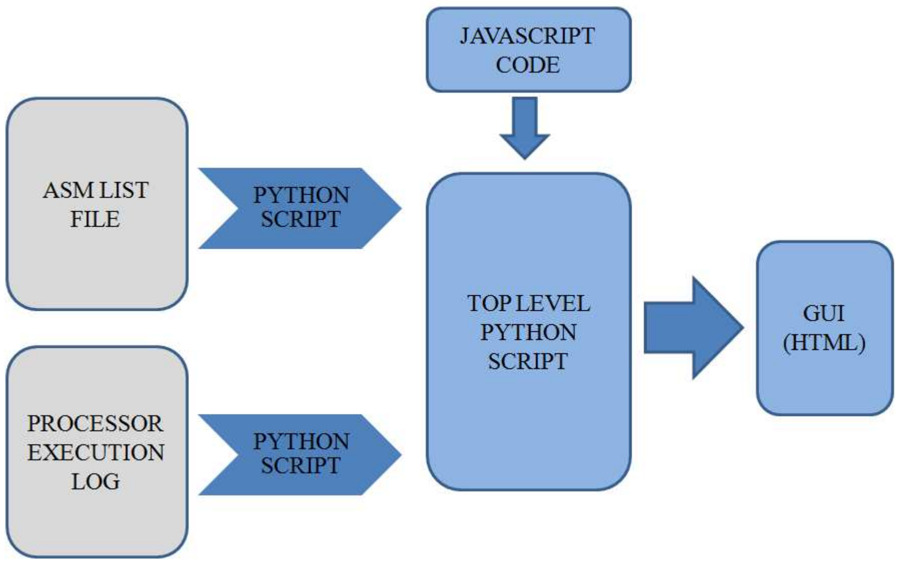


Figure 5.1: GUI Implementation

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.
- Top level *Python* program to manipulate available information and to convert it to *JavaScript* information.
- The top level *Python* program combines 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 targeted 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, 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*.

Resultant *HTML* page embeds *JavaScript* routines and style sheets. This is done by the top level *Python* script. Figure5.2 shows a representation of information flow.

5.1.3 JS DATA STRUCTURE

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 “$dataArray[]$” (Algorithm 3); where each array

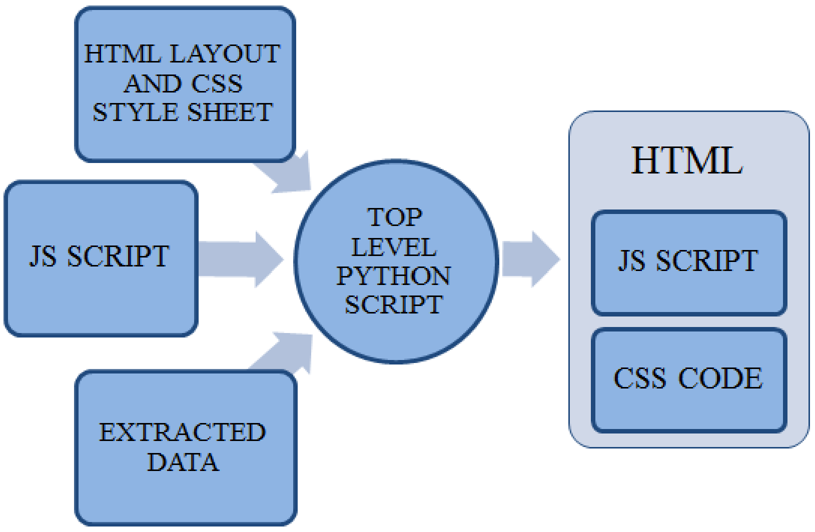


Figure 5.2: Web Page Generation

element corresponds to an active thread. Active threads refer to active core or processor in a multiprocessor system.

```
CreateDataArray()
begin
    for $i$ in $activeThreads$ do
        for $log$ in $logObj$ do
            if $log.Id == i$ then
                Append $log \rightarrow DataArray[i]$
            end
        end
    end
end
```

Algorithm 3: Creating JavaScript Object

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.

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 the object code, and in that process each instruction is also mapped to a particular address. Figure5.3 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 stimuli.

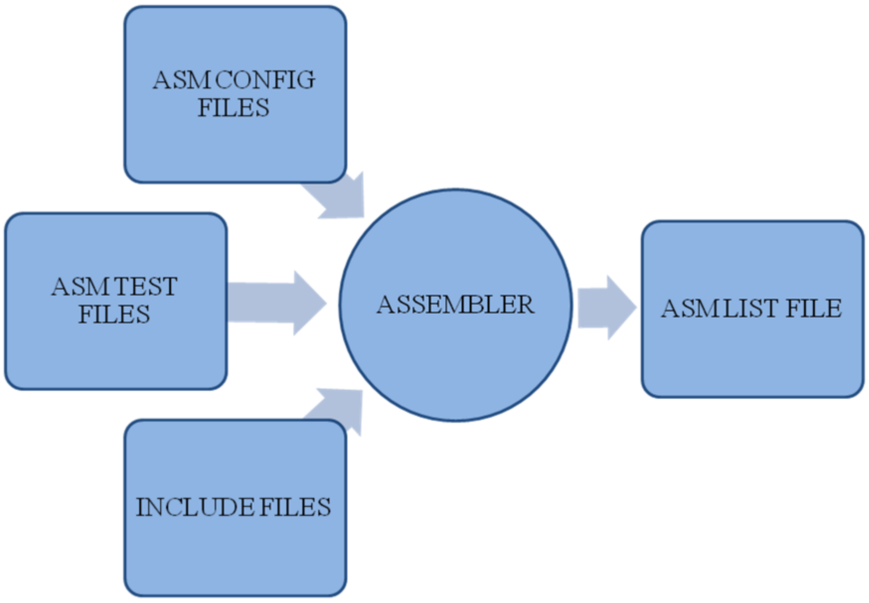


Figure 5.3: Assembler

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 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

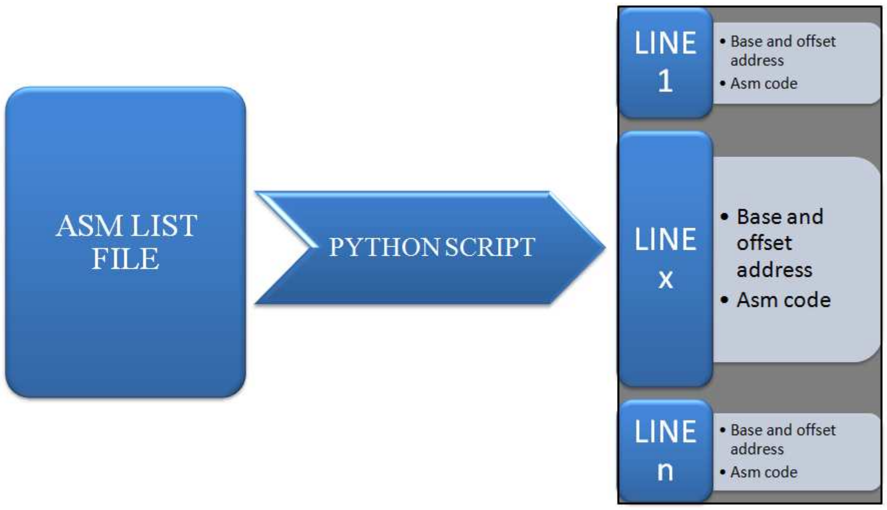

ASM LIST FILE
PYTHON SCRIPT
LINE 1
• Base and offset address
• Asm code
LINE x
• Base and offset address
• Asm code
LINE n
• Base and offset address
• Asm code

Figure 5.4: Asm List File Extraction

- 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 separate data structures as it aids 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 based on linear addresses. As x86 architecture follows segmented memory model along with paging, address translation is required for generating the linear/physical address (Corporation, 2011). Linear address is calculated as:

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

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 displays *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 would select the point and syncronizes 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.

```
Input : list file objects$\rightarrow$ $listObj[]$, log file object$\rightarrow$ $logObj[]$
Start:
for each object list in listObj do
    $list.address = list.Base + list.Offset$
end
for each object log in logObj do
    set count = 0
    for each object list in listObj do
        if $log.address == list.address$ then
            Append each $list$.property $\rightarrow$ $log$.property
            // log properties are lineNo, opcode and address
            count = 1; break loop
        end
    end
    if count == 0 then
        assert: ”$no_address_match$”
    end
end
End
```

Algorithm 4: Combining List and Log File Information

IMPLEMENTATION

Execution flow graph is created with Dygraph JavaScript Visualization Library (Dygraph, 2013). 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 pseudocode used in building the graph in our application.

```
CreateGraph()
begin
    for each i in activeThread do
        for each element in dataArray[i] do
            Dygraph[i] ← $[element.cycleNo, element.lineNo]$
        end
    end
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 difference 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.

```
pointonClickCallBack(element)
begin
    for each reg in RegisterSet do
        regRow[reg] ← element.[reg]
        if (element.[reg] != referenceRow.[reg]) then
            $Highlight_r egRow$
        end
    end
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.

```
onClickCallBack(element);
begin
    for each item from element-50 to element+50 do
        Add $item.opcode$ → InstructionWindow
    end
    Add $item.logInfo$ → ExecutionLogWindow
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*".

IMPLEMENTATION

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

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 *Phython* 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 *.html* 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

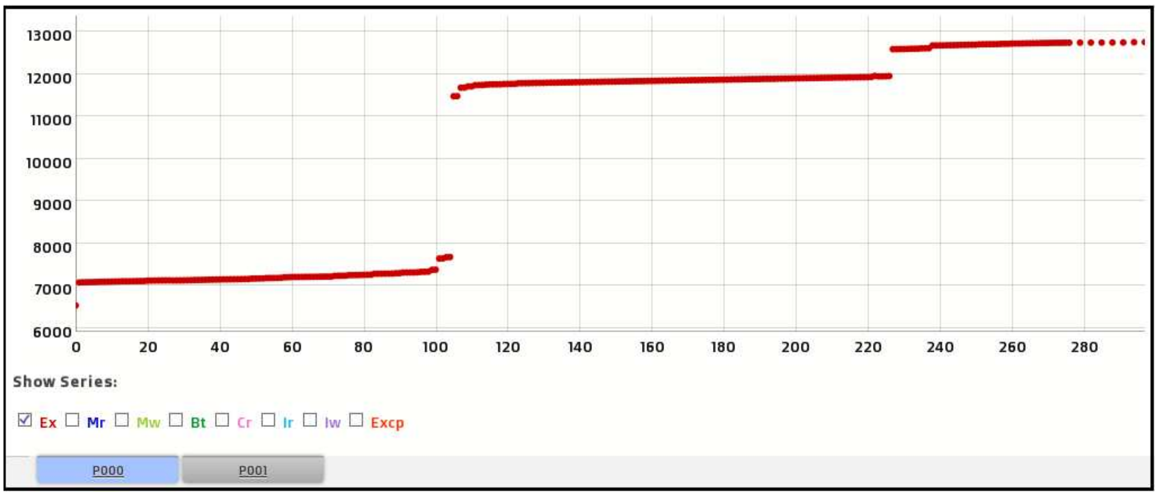


Figure 6.1: Execution Flow Graph

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

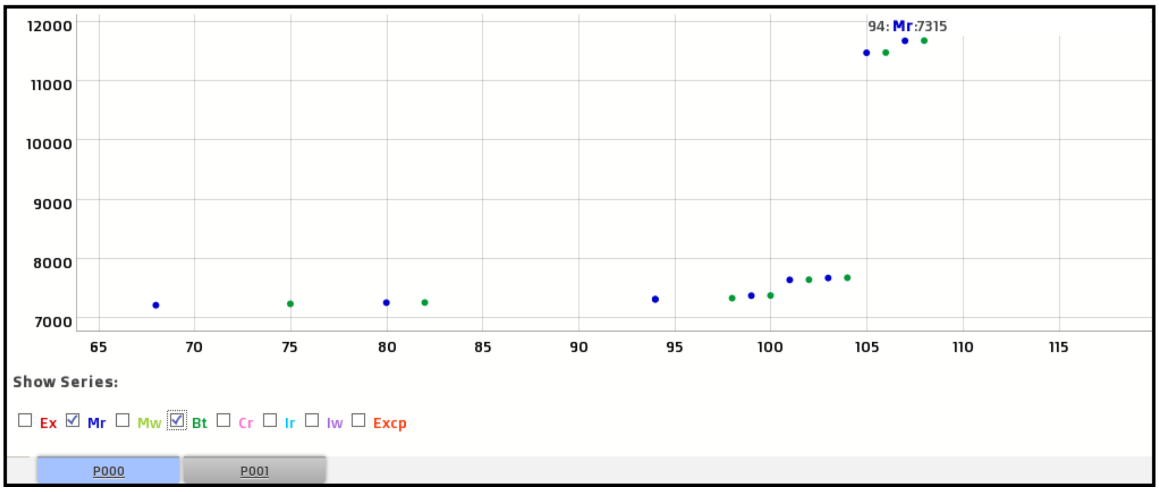
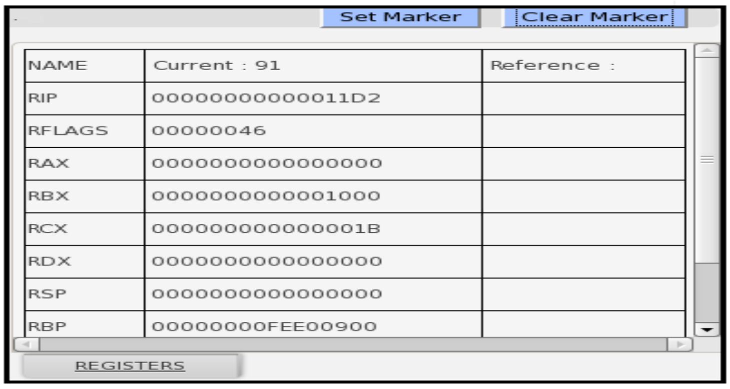


Figure 6.2: Execution Graph With Branching and Memory Writes

Figure 6.2 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.

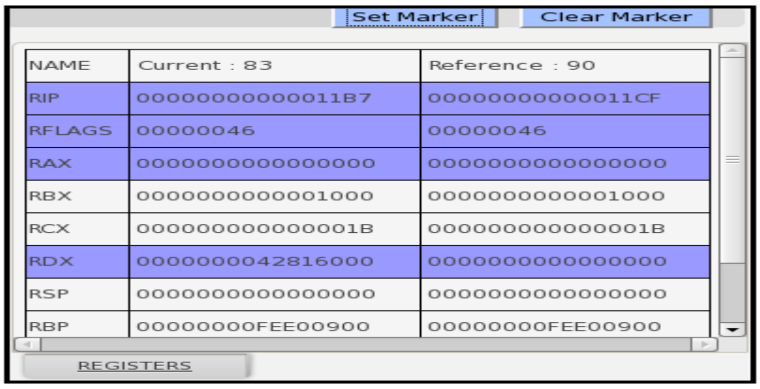
6.2 REGISTER WATCH WINDOW



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

Figure 6.3: Register Watch Window

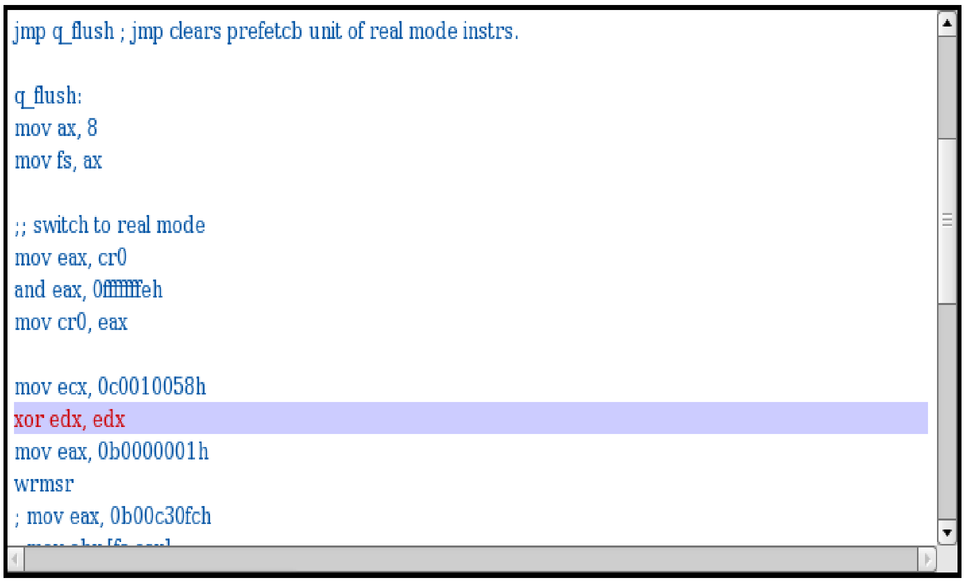
Figure 6.3 is the register window. It could be observed that register values corresponding to selected execution point *cycle 91* is shown in *current selection* column. *Set Marker* and *Clear Marker* buttons could also be observed in the picture. Figure 6.4 shows the comparison of register states at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



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

Figure 6.4: Register Value Comparison

6.3 INSTRUCTION WINDOW



```
jmp q_flush ; jmp clears prefetcb 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 is the instruction window. The instruction window could be seen highlighting the assembly along with its context, corresponding to the selected execution point.

Chapter 7

CONCLUSION

An interactive offline graphical user interface was designed and developed. The interface is completely web based and achieves primary goals of offline 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 separately, 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 features 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 futureZhang (2005).

Verification is done at different abstraction levels. Two main abstraction levels from verification point of view are RTL and gate levelZhang (2005). 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 methods 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 and static verification techniques like LEC and STA Segev *et al.* (2004). 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 cosimulation 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 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 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 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.

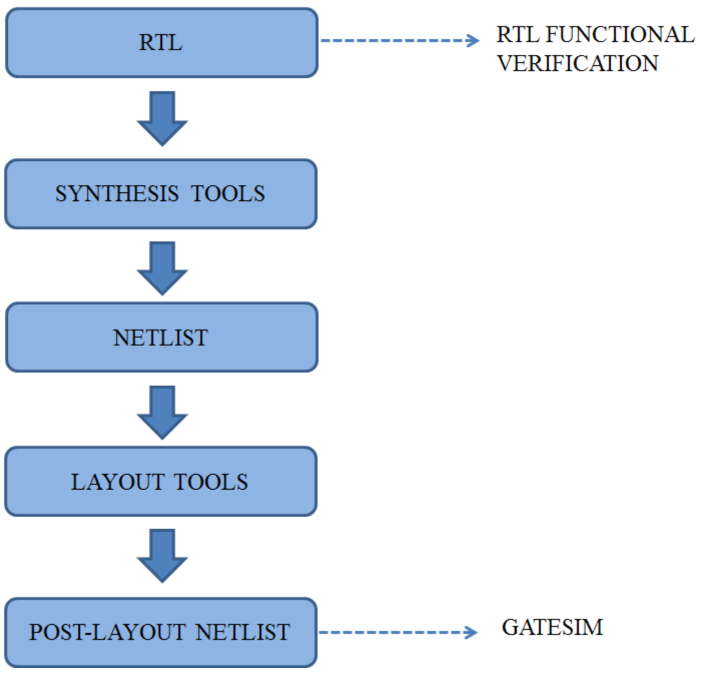


Figure 9.1: Design Flow

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 targeted 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 descriptionsDoe (2009). Although LEC uses sophisticated formal algorithms to identify, map, and compare nodes in the netlists, the complexity is hidden from the userCarlson (2011).

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 where as 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 modeling 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.

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.

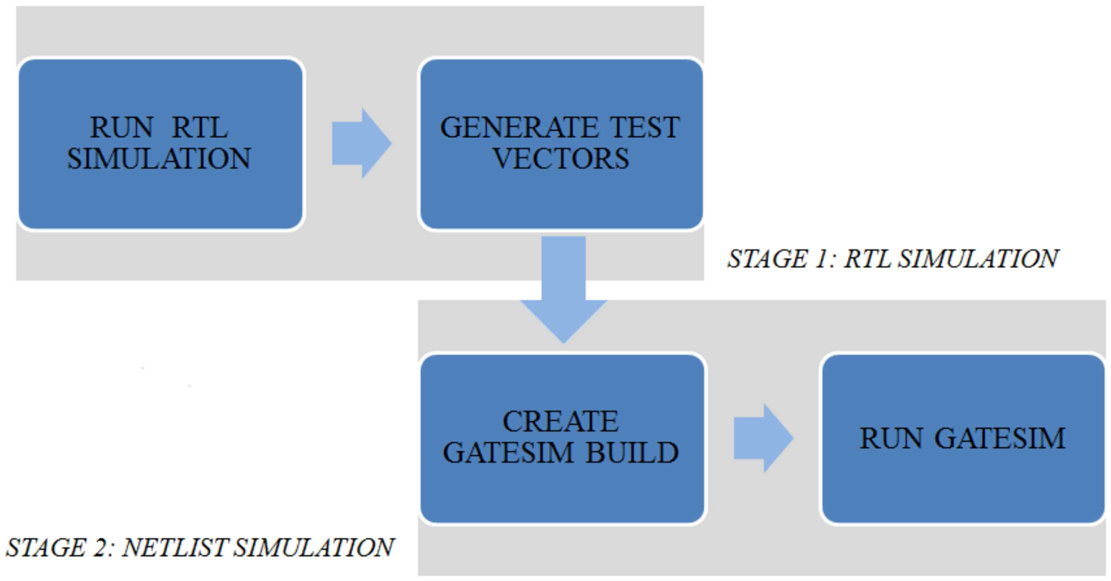


Figure 10.1: Early Dual-Sim Flow

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 methodology 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 comparision and debug easier. Figure 10.2 shows how stimulus is applied to netlist and comparison of output is done in cosim method.

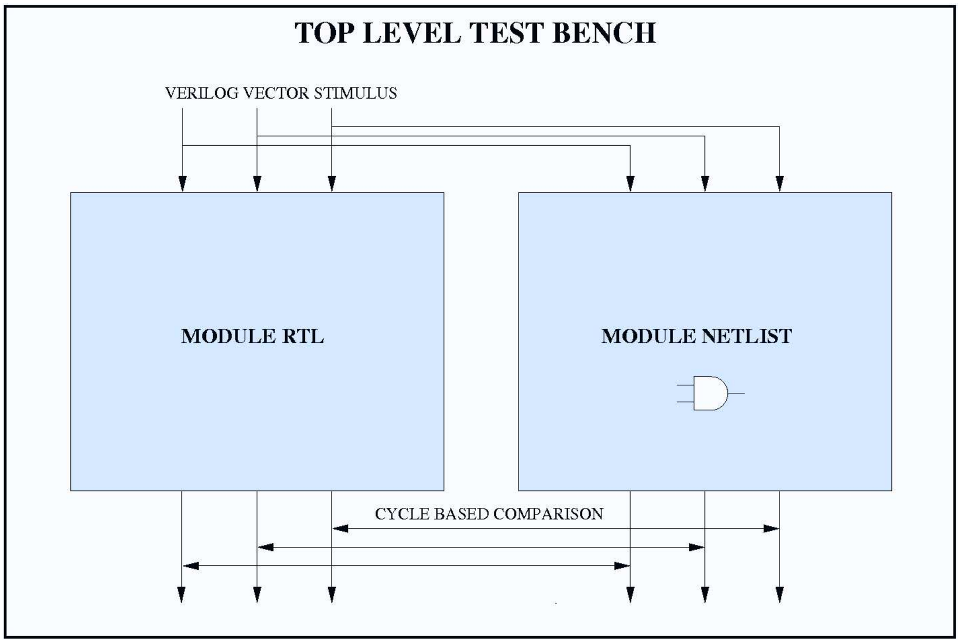


Figure 10.2: Co-sim Based Gatesim

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.

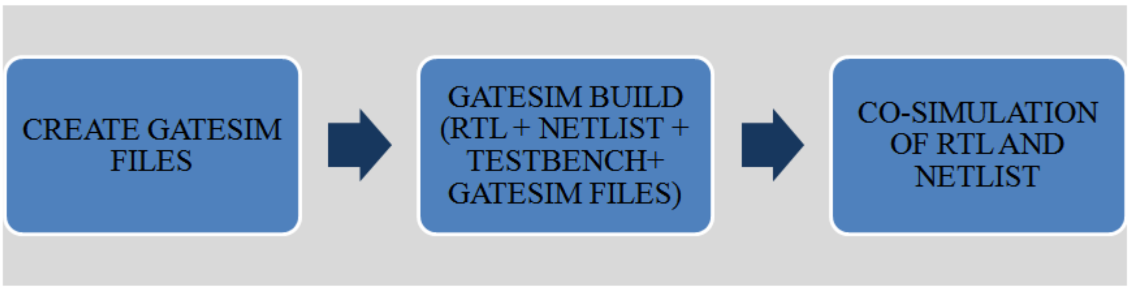

CREATE GATESIM FILES
GATESIM BUILD (RTL + NETLIST + TESTBENCH+ GATESIM FILES)
CO-SIMULATION OF RTL AND NETLIST

Figure 10.3: Co-sim Based Gatesim Flow

Major steps involved in this flow are:

1. **Getting gatesim files**

   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 LIMITATIONS OF CO-SIM METHODOLOGY

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.

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

After analyzing limitations associated with *early dual-sim approach* it could be inferred that the main cause of inefficiency was the method used to capture, store, and apply test-vectors onto the netlist. Analyzing limitations associated with *co-sim approach*, it could be 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

FSDB**?** or Fast Signal Database is a 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**?** tool set for FSDB format could be used to retrieve values from FSDB. PLI/VPI**?** 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. Second simulation with only gatesim components with capability to apply test-vectors from FSDB directly.

11.1 IMPROVED METHODOLOGY

In dual simulation methodology the idea is to have two seperate simulations unlike co-sim. Both these simulations are completely independet except for the stimulus vector file. In the first unmodified RTL simulation (together with test bench components) dump is enabled in FSDB format. This FSDB dump is used as test-vectors for the following netlist simulation. In co-simulation methodology the same test bench will provide stimulus to RTL and netlist. Figure 11.1 shows a flow diagram of proposed dual-sim methodology.

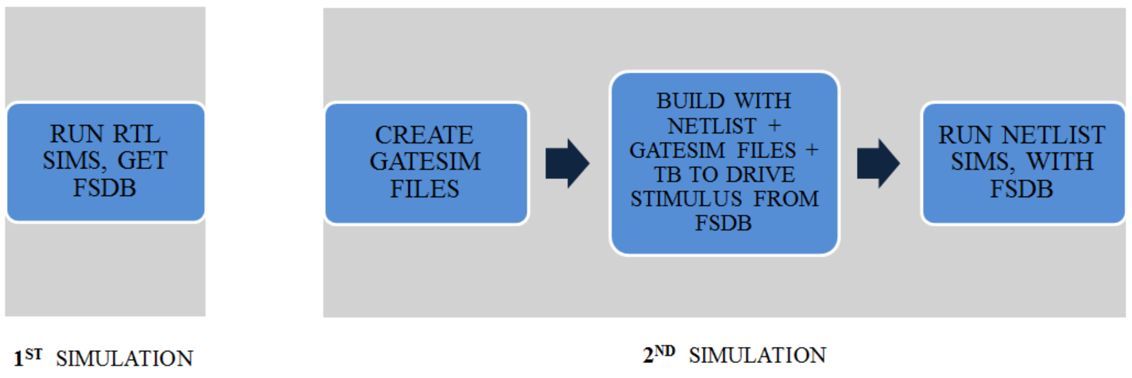

RUN RTL SIMS, GET FSDB
CREATE GATESIM FILES
BUILD WITH NETLIST + GATESIM FILES + TB TO DRIVE STIMULUS FROM FSDB
RUN NETLIST SIMS, WITH FSDB
1ST SIMULATION
2ND SIMULATION

Figure 11.1: Dual Sim

Stages associated with this methodology are

RTL simulation RTL simulation along with test bench components is already being done as part of functional verification. For gatesims, no changes are required except that the simulation signals needs to be dumped into FSDB, through runtime switches. Such FSDB stimulus vectors have to be generated for every stimulus envisioned to be applied onto the netlist. Note that once generated, the stimulus could be reused for repetitive netlist simulations.

Gatesim infrastructure As in the case of co-simulation, gatesim files need to be generated from files obtained from LEC flow. Same infrastructure used in co-sim is used for this and hence no additional effort is required when the methodology is modified. Note that this step needed to be done only once.

Gatesim Build Gatesim build could be obtained after compiling and elaborating the design netlist and infrastructure files with a simulator. The build would be devoid of RTL design and non-gatesim verification infrastructure. Note that this step needed to be done only once.

Gatesim simulation Run all envisioned stimulus vectors onto the netlist.

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 Verilog language as defined in the *IEEE Standard Hardware Description Language* based on *Verilog Hardware Description Language (IEEE Std 1364-1995)* and *Standard Verilog Hardware Description Language (IEEE Std 1364-2001)*. It supports almost all verification, design and assertion constructs of *System Verilog*. It also provides native interfaces like DKI etc. VCS is a compiled code simulator and is widely considered as one of the advanced simulators avialable in the industry.

For this project we had used *VCS Verilog simulator* developed by *Synopsys Inc.* for RTL as well as netlist simulations. Figure11.2 is a self explainatory depction of RTL simulation. The implementation details of this methodology is detailed in following sections.

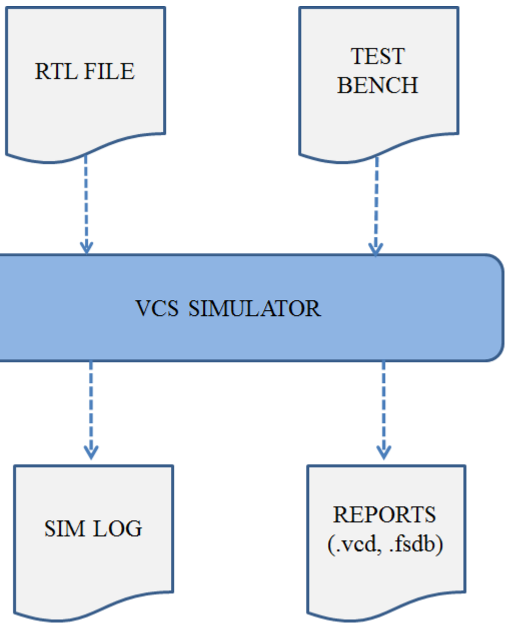


Figure 11.2: RTL Simulation

11.1.1 NETLIST SIMULATION FLOW

Figure11.3 shows netlist simulation flow. Process ⓐ is generation of FSDB stimulus, which is already explained in Section 11.1. Process ⓑ is intricated and is detailed in Section 11.2. Remaining process stages are explained here.

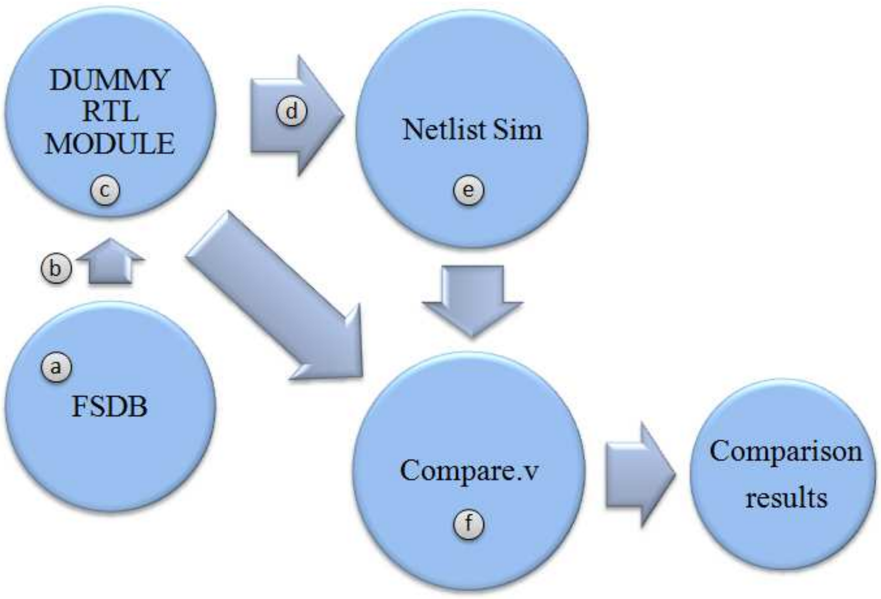


Figure 11.3: Netlist Simulation

Process stages of netlist simulation flow are:

1. Obtain test-vectors in FSDB format
2. Drive stimulus onto a dummy-RTL
3. Dummy-RTL drives stimulus onto netlist
4. Apply appropriate force signals from FSDB on to the dummy-RTL, which then forces it onto appropriate netlist nodes.
5. Accomplish verification goals by clock-cycle-based-comparators connected between golden RTL reference and corresponding netlist nodes.

11.1.2 DUMMY-RTL

When test-vectors were applied from FSDB to *Verilog* design, it was found that VPI/DKI method of stimulus application is not suitable when driving *Verilog* nets. Nets have special behaviour that they needs to be continuously dirven, which in the case of VPI/DKI is through forcing the net. But it was found to have flaky behaviour. Hence it was decided to create a “dummy-RTL” module that would contain all the signals those need to be export out of FSDB. These signals would have same size as their actual counterparts but be of type *Verilog* “reg”. Test-vectors from FSDB would be applied to these signals inside dummy-RTL. Listing 11.1 shows a code snippet of dummy-RTL module (Process © in Figure11.3). All wires, IO signals are of type “reg” in dummy-RTL module.

Listing 11.1: Dummy RTL Module

```
module dummy_RTL();
        reg input_signal_1;
        reg input_signal_2;
         _
         _
        reg input_signal_n;

        reg output_signal_1;
        reg output_signal_2;
         _
         _
        reg output_signal_m;
endmodule
```

11.1.3 SIMULUS APPLICATOR

Stimulus from dummy-RTL module is applied onto netlist by *Verilog* by connecting ports of netlist while instantiation at gatesim test-bench. Listing 11.2 gives a snippet of this Verilog instantiation of netlist module. There could be certain forces required in certain logic inside netlist, which is accomplished through *Verilog*’s “force” statement. These forces could be for initialisation, fuses or for hastening reset release.

Listing 11.2: Verilog Instantiation

```
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();
    );
```

11.1.4 NETLIST SIMULATION

Process ⓔ is simulation with netlist testbench components. These components are reused form of co-simulation methodology's netlist components (Section 10.2), with references of actual RTL paths changed to *Dummy-RTL*'s paths. Waveform dump needs to be enabled during simulation to enable debug in case of failure. Reference RTL waveform dump file is already available for the user.

11.1.5 FUNCTION COMPARATORS

Process ⓕ are a set of comparators those keep checking the behaviour of netlist with respect to RTL on every clock-cycle. Generally the comparision is made on respective active clocks, well after all signal changes have died down. The reference RTL signal is available in the dummy-RTL. Comparators may also consider certain important internal nodes, hence those have to be available in dummy-RTL as well. Listing 11.3 is a code snippet representing a single comparator. Whenever compare fails, a “miscompare” information is written on to simulation transcript, that would aid debug. When test completes without any “miscompare” report, then test is said to have completed without failures.

Listing 11.3: Cycle Based Comparison

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

11.2 STIMULUS APPLICATOR

Process ⓑ is a layered software infrastructure for the purpose of stimulus application onto the dummy-RTL. Figure 11.4 shows four different layers of the applicator. Together they accomplish transporting vector stimulus from FSDB file and finally delivering it to the netlist simulation. The first layer used to access FSDB file is provided by *Verdi* tool itself. It is available as a pre-built object**?**, which could be linked with. The remaining three-layers of the infrastructure has been developed as part of this project and implemented as *C* and *C++* code.

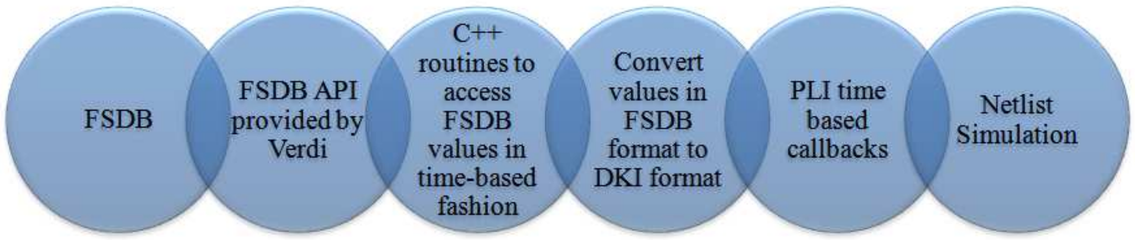

FSDB
FSDB API provided by Verdi
C++ routines to access FSDB values in time-based fashion
Convert values in FSDB format to DKI format
PLI time based callbacks
Netlist Simulation

Figure 11.4: Accessing FSDB Signals

11.2.1 EXTRACTING STIMULUS

FSDB is a binary file format for storing signal value change information. This format commonly used across the *Verdi* toolset and no other program has enough information to decode data from those files. But *Verdi* toolset, provides APIs**?** to be interfaced with. This project make use of those APIs, to accomplish desired task of accessing design signals at any point in time.

In order to successfully extract signal values from stored FSDB file the following needs to be done

Handle FSDB file: In this process approprite FSDB file needs to be “opened” and be “closed” after all operations are completed. *Verdi* provides routines `ffrOpen()/ ffrOpen2()` and `ffrClose()` for these purposes respectively.

Obtain signal handle: In this process handle to signals of interest needs to be obtained through APIs. Sample routine `MyTreeCBFunc`(**?**, p. 3) is already provided by *Verdi*.

Access signal values: Once signal handle is obtained, the signal’s value at this point in time could be obtained by routine `ffrGetVC()`.

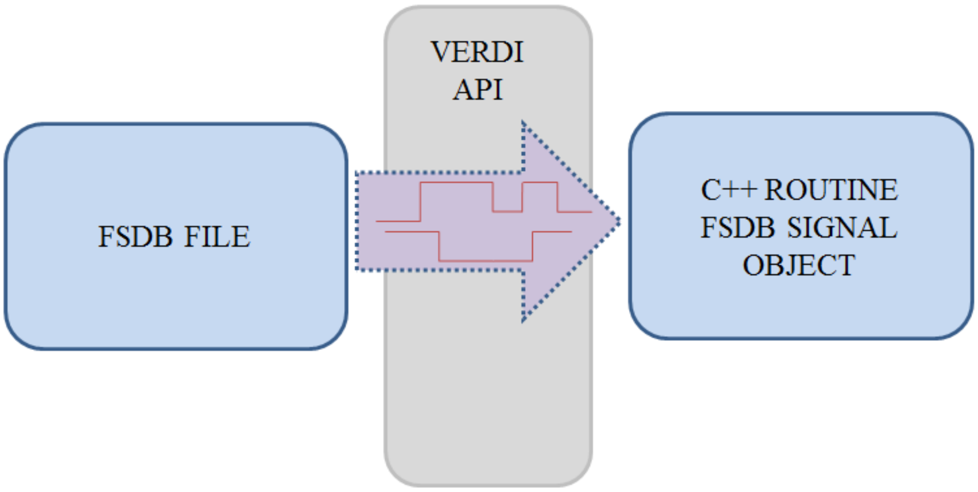


Figure 11.5: API For Accessing FSDB Signals

11.2.2 TIME BASED TRAVERSAL

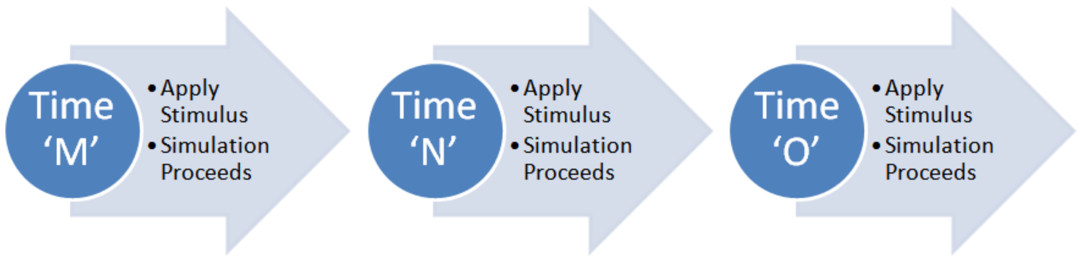


Figure 11.6: Time Based Application of Stimulus

Figure11.6 shows sequences of actions that would occur during simulation. At any simulation time 'M', it would be required to apply the stimulus to all signals of interest. After this, control is transferred to simulator which computes the states values at time 'M'. The event driven simulation would proceed in this fashion for all subsequent event times until it reaches the next stimulus point 'N'. Series of actions that took place for time 'M' would be repeated for time 'N' and so forth.

List of RTL signals those need to be extracted would be refined over time, as netlist becomes stable. In this particular application, it would be required to obtain values of all signals of interest at the desired instant. This type of traversal is called as time-based-traversal. Sample routine `tb.cpp`(**?**, p. 28) is already provided by *Verdi*. In this application, a larger C++ infrastructure is required to be built since

1. Number of RTL signals those needs to be extracted is huge. Hence a seperate data structure is required.

2. Accessing of signal values in time-based fashion could be abstracted out.

3. Signal sizes vary from 1 to hundredes, so handling of different sized signals could be made uniform by $\widehat{C}$++ routine abstraction.

11.2.3 VALUE CONVERSION TO DKI

FSDB exports signals values in its custom 8-bit format, but that can't be used directly by any simulator. These values needs to get converted to a format that could be understood by the simulator. *Verilog* language**?** provides many standard interfaces such as PLI, VPI (or PLI 2.0) for this purpose. The drawback with these standard interfaces are that they are too slow when huge vectors are in consideration (like gatesims). Conversions to and from string based format is very timeconsuming as it was observed.

In order to overcome these limitations, VCS provides a non-standard interface called as DKI . This interface has the advantage of less simulation overhead and smaller memory footprint compared to VPI interface. Hence DKI was choosen for this project.

Signal values could be retrived through *Verdi* APIs, as listed in speudo-code in Listing 11.4.

Listing 11.4: Retrieving Values from FSDB

```
byte_T *vc_ptr;
/* vc_ptr should be assigned to particular signals value change */

// convert verilog bits to unsigned int
uint retVal=0;
uint8 bit;
for (int i=0; i<=(msb-lsb); i++) {
  bit = vc_ptr[i+LBit-msb];
  retVal = (retVal<<1)
           | (bit==FSDB_BT_VCD_1 || (bit==FSDB_BT_VCD_X));
}
```

DKI provides direct interface to internal representation of simulator object. Signal values retrieved from FSDB could be assigned to VCS's object as per instructions in *VCS User Guide*(**?**, Section C Language Interface¿ Direct C).

11.2.4 INTERACTION WITH SIMULATOR

Having obtained vector values and having the capability to apply it to the simulator, the final part of stimulus application is to have the ability to stop simulator at will and trigger new vector application. Since advanced simulators like VCS are event based, the interruption on simulation has to be done on need basis rather than on every unit dealy. This could be accomplished with the use of standard simulator interfaces such as VPI callbacks. Enough documentation on this subject could be found in standard texts**? ?** and online**? ?**. Listing 11.5 shows a snippet of code that could be used to schedule a callback from simulator by specifying the delta-delay.

Code 11.5: Scheduling a VPI callback

```
vpiHandle   cbHandle;
s_vpi_time  vpiTimeObj;
s_cb_data   cbData;

vpiTimeObj.type  = vpiSimTime;
vpiTimeObj.low   = delay & 0xffffffff;
vpiTimeObj.high  = delay >> 32;

cbData.reason     = cbAfterDelay;
cbData.cb_rtn     = GlobalCallbackSyncFunc;
cbData.time       = &vpiTimeObj;
cbData.value      = NULL;
cbData.obj        = NULL;
cbData.user_data  = (PLI_BYTE8*) vpiCallbackPtr;

cbHandle = vpi_register_cb(&cbData);
vpi_free_object(cbHandle);
```

Delta-delay is the simulation time difference between next stimulus point with respect to the current simulation time. Current simulation time could be obtained through VPI `vpi_get_time()` routine. The next stimulus point could be obtained through fsdb routines. Stimulus application is acheived by continously progressing to next stimulus point followed by application of stimulus, as shown in snippet Listing 11.6.

Code 11.6: Stimulus Applicator Routine

```
static fsdbXTag cur_time, nxt_time;

/* Find the next time instant in FSDB and cause a callback at that time */
```

```
while (FSDB_RC_SUCCESS == tb_trvs_hdl->ffrGotoNextVC()) {
  tb_trvs_hdl->ffrGetXTag(&nxt_time);
  if ((cur_time.hltag.L != nxt_time.hltag.L) || (cur_time.hltag.H !=
      nxt_time.hltag.H)) break;
}
if ((cur_time.hltag.L != nxt_time.hltag.L) || (cur_time.hltag.H != nxt_time
    .hltag.H)) {
  uint64 delta = (((uint64)(nxt_time.hltag.H - cur_time.hltag.H))<<32) | (
      nxt_time.hltag.L - cur_time.hltag.L);
  ScheduleAfterDelayCallback(delta, this); /* This routine would be called
      back again after the delta */
}

/* Do assignments */
for (uint32 ui=0; ui<num_sigs; ui++) {
  /* Assign FSDB value of signal to DKI object */
}

/* Finish if no more stimulus */
if ((cur_time.hltag.L == nxt_time.hltag.L) && (cur_time.hltag.H == nxt_time
    .hltag.H)) {
  /* Finish simulation */
}

/* Will be used during recursive call */
cur_time = nxt_time;
```

Chapter 12

RESULTS

A new gate simulation flow based on separate simulation of RTL and netlist was developed. Performance analysis was done with respect to co-sim methodology. A set of standard test cases were considered for the purpose. An exclusive machine was used for benchmarking and hence tests were run in sequence one after the other in both the methodologies. The machine features were:

- 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 compares simulation performances whereous Table 12.2 compares simulation memory requirement.

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. Simulation performance when benchmaked and compared against current co-simulation method on a dedicated machine, shows a consistent improvement of around 10. Use of FSDB as the source of test vectors helps to keep the disk requrirement minimal and to get rid of bulky time hogging test bench components. The methodology also enables quicker turn-around time. This 10 times improvement in simulation performance will help in aiding time-to-market of cutting-edge processors. The method still requires a one time RTL simulation runs for generating stimulus (as FSDB). The proposed methodology needs minor modifications to existing co-simulation methodology and hence could be adopted to new projects with ease.

Though the improved methodology is dependent on vectors from RTL simulations and is not independent by itself, gains obtained in turn-around times were considerable, and is a good return-on-investment when complete gatesim verification is considered.

Potential for future work: Observed performance of improved dual-sim methodology could be further increased by optimizing glue code used for reading FSDB and applying that inside design. Infrastructure created during the course of this project could be packaged into libraries to simplify porting between projects.

Chapter 14

REFERENCES

Bibliography

AMD, C. (2011).

Brackenbury, L. E. M., **L. A. Plana**, and **J. Pepper** (2010). System-on-chip design and implementation. *IEEE Trans. Education.*, **53**(2), 272–281.

Carlson, N. (2011). How Many Users Does Twitter REALLY Have? URL `www.businessinsider.com/chart-of-the-day-how-many-users-does-twitter-really-have-2`

Corporation, A. (2011).

Doe, R. (2009). This is a test entry of type @ONLINE. URL `http://www.test.org/doe/`.

Dygraph (2013).

Kuhn, T., **T. Oppold**, **M. Winterholer**, **W. Rosenstiel**, **M. Edwards**, and **Y. Kashai** (2001). A framework for object oriented hardware specification, verification, and synthesis. *In Proceedings of the 38th annual Design Automation Conference*. ACM, 2001.

Segev, E., **S. Goldshlager**, **H. Miller**, **O. Shua**, **O. Sher**, and **S. Greenberg** (2004). Evaluating and comparing simulation verification vs. formal verification approach on block level design. *In Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems, ICECS.*. IEEE, 2004.

Zhang, L. (2005). *Design Verification for Sequential Systems at Various Abstraction Levels*. Ph.D. thesis, Citeseer.

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