Develop Coverage Framework and Architecture Profilers

Major Project Report

Submitted in fulfillment of the requirements for the degree of

**Master of Technology
in
Electronics & Communication Engineering
(Embedded Systems)**

By

Dhruv Dave

(16MECE28)

**Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481**

May 2018

Develop Coverage Framework and Architecture Profilers

Major Project Report

Submitted in fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering

By

Dhruv Dave
(16MECE28)

Under the guidance of

External Project Guide:	**Internal Project Guide:**
Mr. Gopalakrishnan Chidambaram	**Dr. N.P. Gajjar**
Staff Engineer	PG Coordinator, Embedded Systems,
Arm Embedded Technologies Pvt. Ltd.,	Institute of Technology,
Bangalore.	Nirma University, Ahmedabad.

Electronics & Communication Engineering Department

Institute of Technology-Nirma University

Ahmedabad-382 481

May 2018

Declaration

This is to certify that

a. The thesis comprises my original work towards the degree of Master of Technology in Embedded Systems at Nirma University and has not been submitted elsewhere for a degree.

b. Due acknowledgment has been made in the text to all other material used.

- Dhruv Dave
16MECE28

NIRMA UNIVERSITY
INSTITUTE OF TECHNOLOGY
NAAC ACCREDITED 'A' GRADE

Certificate

This is to certify that the Major Project entitled **"Develop Coverage Framework and Architecture Profilers"** submitted by **Dhruv Dave (16MECE28)**, towards the fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of our knowledge, haven't been submitted to any other university or institution for award of any degree or diploma. Date: Place: Ahmedabad

Dr. N.P. Gajjar

Internal Guide

Program Coordinator

Dr. D. K. Kothari

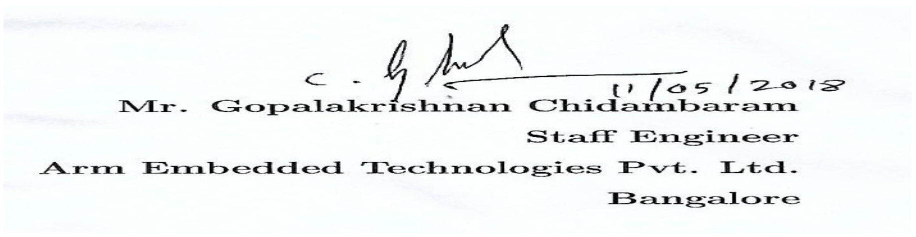
Section Head, EC

Dr. Alka Mahajan

Director, IT

Certificate

This is to certify that the Major Project titled **"Develop Coverage Framework and Architecture Profilers"** submitted by **Dhruv Dave(16MECE28)**, towards the fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.



[illegible] 11/05/2018

Mr. Gopalakrishnan Chidambaram
Staff Engineer
Arm Embedded Technologies Pvt. Ltd.
Bangalore

Acknowledgements

With immense pleasure, I would like to present the report on the work **"Develop Coverage Framework and Architecture Profilers"**. I am very thankful to all those who helped for the successful completion of the project work and for providing valuable guidance throughout. I express my gratitude and sincere thanks to **Dr. N.P. Gajjar**, PG Coordinator of M.Tech Embedded Systems and **Dr. Sachin Gajjar** for guidelines during the review process.

I take this opportunity to express my profound gratitude and deep regards to **Dr. N.P. Gajjar**, guide of my internship project for his exemplary guidance, monitoring and constant encouragement.

I would also like to thank **Mr. Gopalakrishnan Chidambaram**, external guide of my internship project from **Arm Embedded Technologies Pvt. Ltd.**, for guidance, monitoring and encouragement regarding the project.

- Dhruv Dave
16MECE28

Contents

List of Figures

Abstract

Arm Fast Model is a software simulation platform which simulates the Arm architecture functioning. Arm Fast Model support generation of traces that consistently track the execution and related activities. By creating a plug-in in the form of DLLs, we use these trace sources. It is very difficult to verify whether the entire architecture is fully compliant with given specification as we need to make sure every architectural and micro-architectural feature has been exercised. For verifying the functional correctness of the microprocessor core, a wrapper is built around the microprocessor core to give some input to it and monitor the output. We developed tools as a part of test case methodology to observe input stimuli quality. We worked on the development of Coverage and Profiler tool. The coverage tool determines how is the input stimuli quality and helps in picking out the corner cases where the architecture is not sufficiently stressed. We were able to obtain coverage data and pick out corner cases which were not sufficiently stressed. The Profiler tool helps to obtain more information about the test cases. Based on the feature enabled by user, Profiler checks for the feature across test suites and gives information. Main work of coverage and profiler tool is to provide feedback about the Test Suite and in turn help to improve the test suites quality for the architecture verification.

Abbreviation Notation and Nomenclature

GTI	Generic Trace Interface
API	Application Programming Interface
FVP	Fixed Virtual Platform
CADI	Component Architecture Debug Interface
IP	Intellectual property
RTL	Register Transfer Level
AEM	Architecture Envelope Model
GIC	Generic Interrupt Controller
CSV	Comma Separated Value
ELF	Executable Link Format
AXF	Arm Executable Format

Chapter 1

Introduction

1.1 Motivation

With the increase in the complexity of the CPU design, it gets very important to test them on their timing and functionality and every of its feature needs to be tested exhaustively. Modern CPU are complex enough that it isnt realistic to test them exhaustively.

There are two focus areas for a processor:
To verify that the architecture design under test (DUT) is compliant with the architecture specifications and every instruction and mode are compliant with ISA.
To verify that the architecture design under test is working correct functionally by random testing using random instruction stream generators and find bugs.

For this purpose, a normal thing to do is to exhaustively give the input all the possible combinations under all the states possible to give the input. But considering the modern complex CPUs, it gets difficult to implement the above statement. Most CPU verification teams use a form of biased random testing where algorithms designed by a separate team would throw structured random groups of instruction intended to test out various functionality of the CPU.

There are metrics called as Monitors, Checkers, Profilers and Coverage which are used to measure how much of the functionality of the architecture (CPU) has been tested through the monitoring and measuring the functionality covered using these random instruction groups as well as directed test cases. Verification involves a mix of random and directed tests. Each feature must be exercised at the architecture level as well at the micro-architecture level to cover focus areas as mentioned above. There is a certain pass rate criterion, which when achieved the verification of the architecture is said to be complete.

Arm Fast Models enable the development and validation of the OS, Firmware, Software and other applications for the Arm architecture well ahead of the target silicon arrival. It is very important to determine how well our tool /model /application is performing in terms of coverage of all the micro-architectural and architectural events of our core. It can be difficult to ensure that each specific implementation of a processor is entirely compliant with the specification. This is due to very large total stimuli space of the architecture. Every architectural and micro architectural feature should be exercised. To test the correctness of the CPU, Typical approaches involve collecting large test-suites of real Software Simulator and running it on the architecture along with verification tools and collecting results and analyzing them.

1.2 Problem Statement

To make sure that the architecture and micro-architecture is verified thoroughly, there are test suites. The quality of the test cases needs to be validated as well to make sure they are verifying the architecture completely. The question is that for each of the specific implementation of architecture and micro-architecture, we have bunch of test cases. But how to make sure that the test really stresses on all events?

1.3 Approach

As a part of the architecture testbench environment, tools need to be developed to verify on the two focus areas. Functionality is needed which tracks the behaviour and quality of the test suite along with monitoring what all features it stresses upon. There is an interface provided as a part of Arm Fast models which has the feature of tracing the entire architecture execution. Using it as a base framework, functionality can be developed on the top of the interface and do profiling and coverage.

1.4 Scope of Work

Architecture tools to be developed as a part of the testbench methodology using the existing interface framework for Arm Fast Models. The plugin framework involves extracting the API data fields into our program variables and the planning out the functionality or check required. Both the profiler and coverage tool have a different use case. The profiler tool helps in segregating the test cases based on which feature is exercised. The coverage tool helps in determining the quality of the input stimulus and in turn helps in improving the arm test generation tools.

Chapter 2

Literature Survey

2.1 Overview

ARM fast model is a simulation model which is generally used pre-RTL and is very useful for architecture trace, debug and profiling. The ARM fast model is provided to the partner companies along with the test cases as a part of architecture validation suite (AVS). The Fast Model is made available 12-18 months before the actual release of the Arm CPU. It helps the developers, firmware engineers, OS creators and application developers to start developing the applications way before the actual silicon arrival. Fast model is a dynamic binary translator. It converts all the arm instructions to be run on the host CPU I.e. x86 architecture. So, at the run time, to speed up the arm architecture simulation, dynamic binary translation from arm instruction set to x86 instruction set is done. By writing high level C program functionality as well, this can be achieved but then the simulation speed would be very slow. Fast model is generally the functionally accurate version and Fast Model ESL is the cycle accurate version.

The Arm Fast Model is Programmers View (PV) models of processors and devices work at a level where functional behaviour is equivalent to what a programmer would see using the hardware. It also helps for the architecture debug, trace and profiling which is generally used to verify the architecture functionality at the early stage of the development. The partner companies use the fast model to have early development cycle. Also, the partner companies do the implementation of the design based on the architecture specification. The architecture test case methodology is prepared as a part of the architecture compliance based on the Arm fast model. The Arm Fast Models provide an interface where we can trace the execution of the architecture. The architecture can be traced by passing a query at run time to the fast model and that query is responded back. The tools are prepared for verification of the architecture validation suite which uses the generic trace interface framework. But the use cases for the tools will be different even though they are based on the single framework.

2.2 Generic trace Interface

The Fast Models provide a functionality to trace the execution of the architecture. This functionality is called as Generic Trace Interface. The trace interface can be used to trace the architecture execution through which we can plan targeted events to verify architecture implementation. The generic trace interface model is prepared using the object-oriented principles. The framework allows creating plugin and binding it with the architecture at the run time. This is a base class for plugin from where plugin instance is created. Once the plugin instance is created using virtual functions, the derived classes are allocated memory at run time and the plugin is loaded with the fast model.

A plugin framework can be prepared using the generic trace interface which would bind with the execution of the fast models at the run time and it will monitor the random instruction sequence group as it will trace the architecture execution. The entire interface model is developed using object-oriented principles and using the functionality of the virtual function, late binding is done i.e. at run time with the fast model. At run time, the plugin framework gets loaded and unloaded with the fast model. The generic trace interface also provides the option of architecture debug interface where input configurable parameters can be set for example: debug information of the architecture instruction by instruction can be seen, which coverage events to be enabled or disabled so that we are not stressing on all the events but only on the one required. Entire infrastructure is to be adopted in the form of a template and only certain things as a part of the architecture is supposed to be changed.

The generic trace interface also provides the option of architecture debug interface where input configurable parameters can be set for example: debug information of the architecture instruction by instruction can be seen, which coverage events to be enabled or disabled so that we are not stressing on all the events but only on the one required. Entire infrastructure is to be adopted in the form of a template and only certain things as a part of the architecture is supposed to be changed.

2.3 Trace Source API's

The Arm fast models provide functionality through which architecture execution can be traced at instruction level. There is a separate plugin which when attached to the Fast Model gives out the entire textual trace of the architecture execution data fields. Basically, an API is a trace source event. For example: an API with the name "Loads" would be focusing on all the load instructions which are happening as a part of the architecture execution. There are more than 300 API fields and each API consisting of more than 5 data fields which can be utilized.

As a part of the API, there is a set of data fields which are supposed to be called when the architecture executes. So, consider an example for a "Loads" API, the corresponding data fields within it would be: physical address, virtual address, access size and so on. These data fields are supposed to be captured within our variables in framework and further functionality for coverage can be planned out of it. These data fields have specific data type along with the specific number of bytes to be accessed. There are more than 250 such API trace sources as a part of the events within the architecture execution. It includes both the architecture as well as micro architectural event. Not all the API source important and not every API may give reliable results.

As a part of literature, the API trace sources are important in coverage functionality planning as they are to be planned from the available data fields present within the API's. There are enough API and data fields within them to plan out architecture functionality. Consider the below structure as a part of the fields where API data fields are extracted and stored.

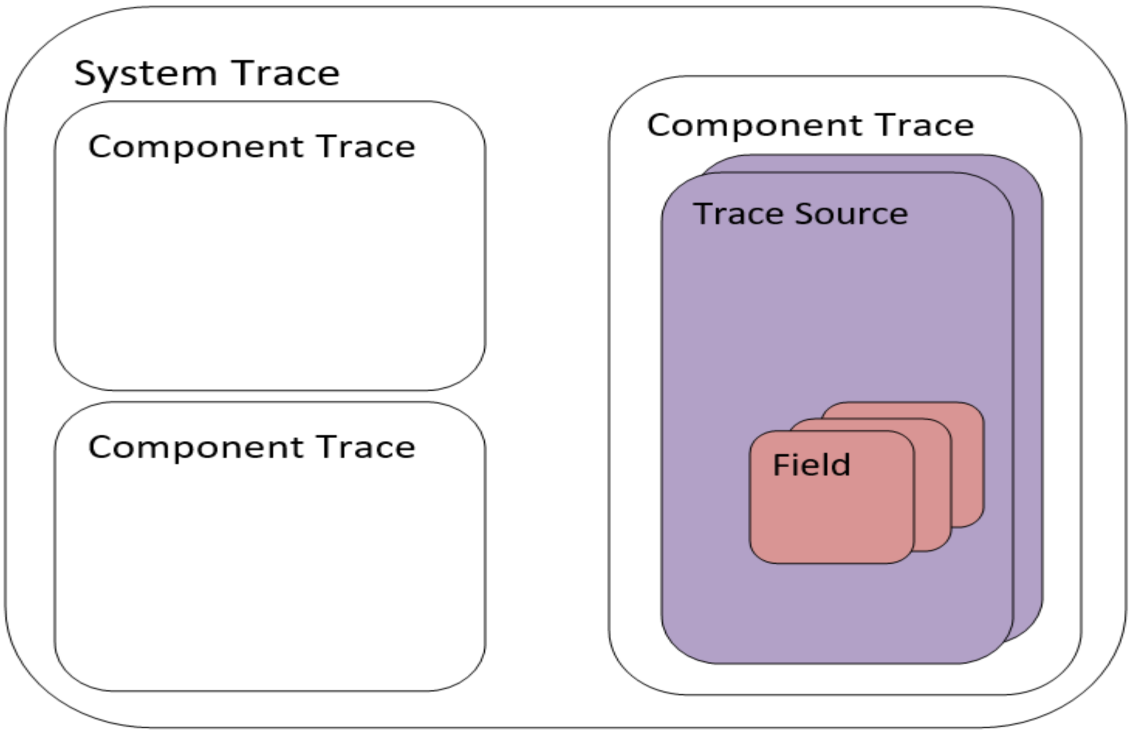
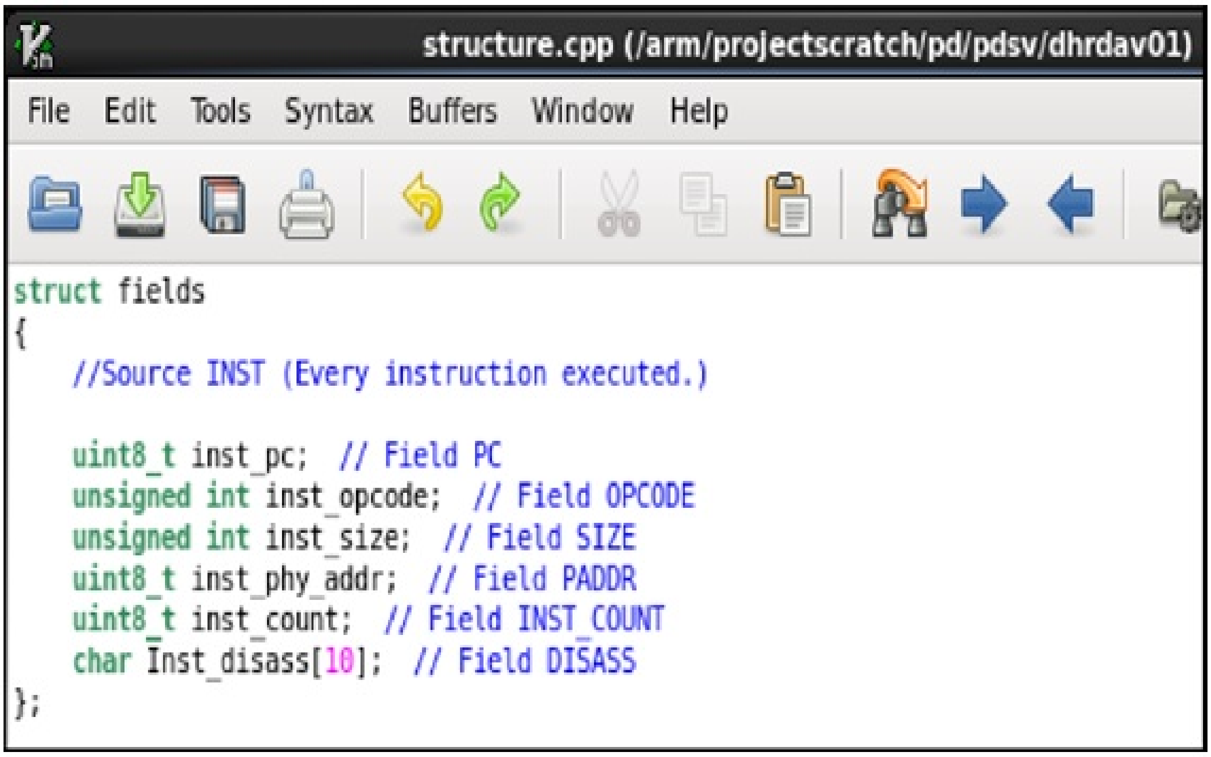


Figure 2.1: AEM Generic Trace Interface

The above diagram shows traces for the system. The system trace has many components called as component traces i.e. APIs. Each of the component trace sources has a set of data fields corresponding to that API. The data field is the location where the architecture fields being updated is registered every time a call to the event source happens. The trace source data field is important to plan out tools development by extracting the architecture data. Multiple plugins can be loaded with the architecture at the same time.


structure.cpp (/arm/projectscratch/pd/pdsv/dhrdav01)
File Edit Tools Syntax Buffers Window Help

```
struct fields
{
    //Source INST (Every instruction executed.)

    uint8_t inst_pc;  // Field PC
    unsigned int inst_opcode;  // Field OPCODE
    unsigned int inst_size;  // Field SIZE
    uint8_t inst_phy_addr;  // Field PADDR
    uint8_t inst_count;  // Field INST_COUNT
    char Inst_disass[10];  // Field DISASS
};
```

Figure 2.2: Structure Fields

The above structure is like the API trace source API along with its data fields. The fields have specific data type and it needs to be extracted from the model in a similar way otherwise wrong data will be read. There are enumerated types as well which has a set of possible values for a data field.

A framework is prepared and laid out on the top of which the architecture events can be planned and implemented to make sure the coverage is hit or not. The coverage data obtained in turn helps to analyze how rigorous are we testing architecture events. Which can be utilized as a feedback to the test generation tools and in turn improve them. The coverage functionality can be prepared on the top of the CPU software framework.

Chapter 3

Coverage Plugin Framework

3.1 Code Flow for Creating a Test Executable

The below figure shows the code flow which is followed to make an arm executable test case which is to be run on the Arm Fast Models.

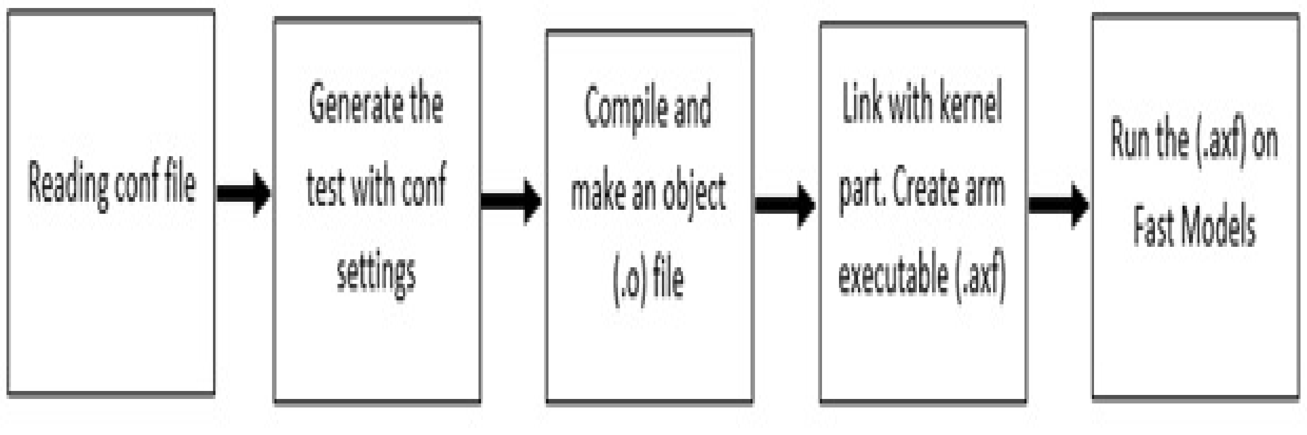

Reading conf file
Generate the test with conf settings
Compile and make an object (.o) file
Link with kernel part. Create arm executable (.axf)
Run the (.axf) on Fast Models

Figure 3.1: Code Flow

The conf files are the files which contains all the features a user wants to exercise or stress upon. So, there is a list of many parameters within a conf file which is actually a knob setting. Conf file is basically a rule-based file based on the architecture specifications. So, if we want the test case to target upon some unique architecture parameter e.g. atomic operations we can add more weight to the particular event and the test would be built in such a way that while execution it stresses particularly on atomic operations.

As a part of the tool, the reading of the conf file which actually is a configuration file is done. Once the conf file is read, task is to generate the test case with the conf settings selected. Once the test file has been created with a unique conf setting, it is further compiled, and an object file is made out of it. The object file is still left to be linked with some important files. The test file which is in the form of binary object is linked with the kernel part and once linked, it creates an arm executable i.e. axf extension file. The axf file is the arm executable which is ready to be run on the fast models. This is the general code flow for creating a test executable as a part of architecture verification.

3.2 Block Diagram: Coverage plugin framework

By creating the plugin functionality, we can load our plugin framework with the fast model at run time. The block diagram of loading plugin framework with with fast models along with arm executable is shown in below figure:

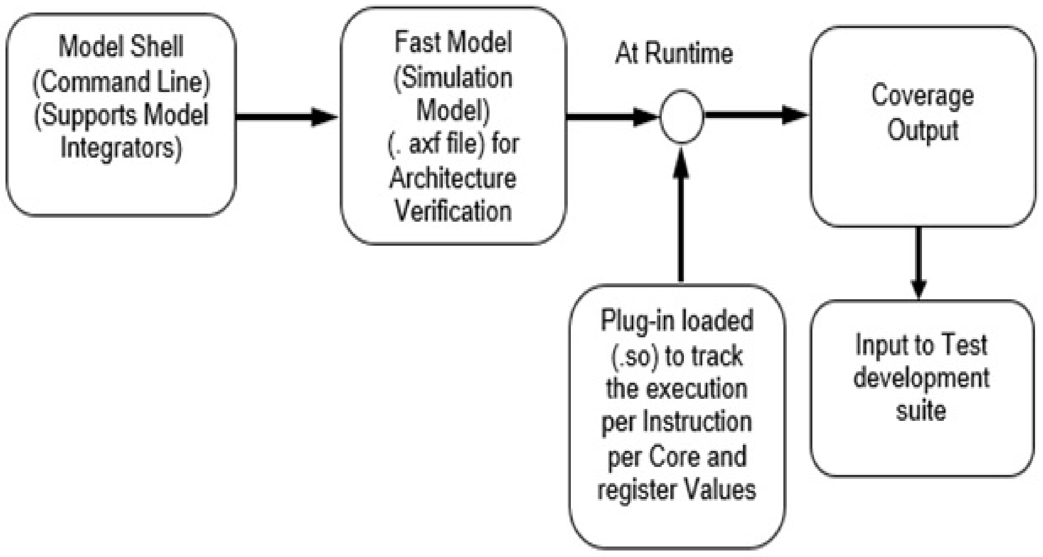

Model Shell (Command Line) (Supports Model Integrators)
Fast Model (Simulation Model) (. axf file) for Architecture Verification
At Runtime
Coverage Output
Plug-in loaded (.so) to track the execution per Instruction per Core and register Values
Input to Test development suite

Figure 3.2: Block Diagram: Coverage Framework

As shown in the above block diagram, the model shell is our shell which has been configured to support the model integrators and model parameters. Over here, the fast model which is in the form of a shared object (.so) is run along with the test file i.e Arm executable test file for architecture verification.

Additionally, at the run time, plugin framework is loaded with the fast model and the test file. The plugin is added as a part of the command within the command line interface. At the end of the execution when the run completes, the coverage output would get into the csv file. The coverage output would give the results that how many times the events have been coverage hit.

Once we receive the coverage output data, we can either run multiple test case and have a coverage data collection which is more rigorous in terms of the data. Once the output coverage data is received, it can be analyzed and further can be used to give input or automatic feedback to the test generation tools. This helps us in improving the test generation tools and making better random instruction sequence for verifying the architecture.

The main work involved here is to develop the coverage plugin framework. Direct events can also be added by creating separate plugin each time. But then every time the infrastructure needs to be created for loading and unloading of the plugin within the fast model. Here, the better option is to rather create a common intermediate platform over which more than single coverage event can be targeted.

So, by using the concept of data structures, an intermediary layer can be prepared where all the data fields within more than one API are pushed up on to the structure and structure maintains a set of data fields. Idea is to plan up the coverage events by then extracting the data fields from the data structure into our own variables. But data structure serves as a middle layer due to which multiple coverage events can be added as a part of the same plugin framework.

3.3 Flowchart: Coverage Plugin Framework

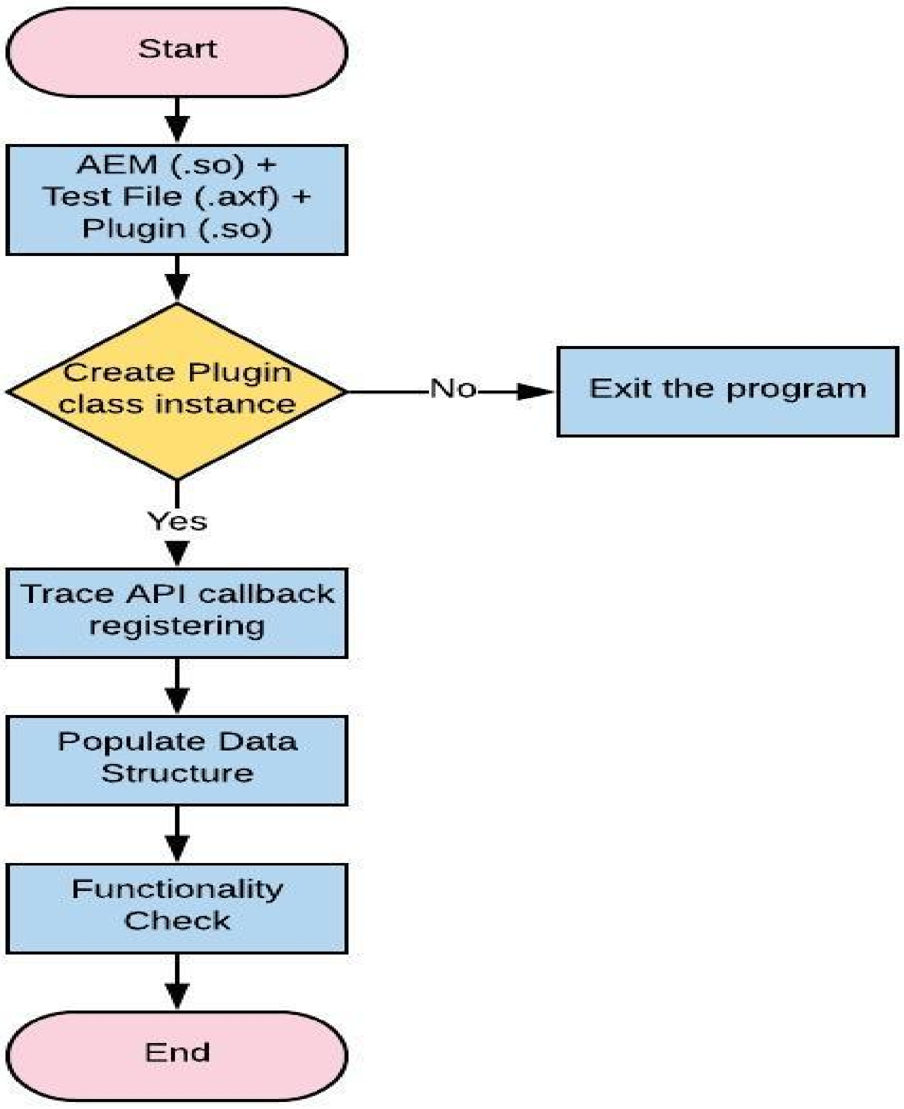

Start
AEM (.so) + Test File (.axf) + Plugin (.so)
Create Plugin class instance
No
Exit the program
Yes
Trace API callback registering
Populate Data Structure
Functionality Check
End

Figure 3.3: Flowchart: Coverage Framework

The entire flow is such that it starts with attaching the plugin along with the fast model (so) and the test file (axf). At the run time, the first task is to attach the plugin I.e. load the plugin with the model. For it, we have to create a plugin class instance in order to load the plugin with the fast model. Once a simulation instance of the class has been created, next task is to associate the plugin with the fast model and allocate the required amount of memory at the run time for all the API trace source data fields.

Once the memory is allocated, the entire initialization of getting the system trace interface for each API is done, where there is infrastructure involved for getting each of the data field. For getting the data field, we need to have an intermediary function called as thunk function which in turn calls our trace source function. To get the correct values for each of the data field within an API, we need to have a value index where we get each of the bit fields individually with respect to their value index which points to where the exact bit field is. In case due to some reason like incorrect field within an API or a model version which does not support the trace interface, we may encounter an error.

If the above part has been executed, we have successfully loaded the plugin with our fast model. The next task is to register for the call backs. The callback registering means getting the program control from the fast model to our plugin at the execution of our given API. If the API is executed per instruction, then we register the callbacks per instruction and we register all the data field values within our structure for API's. Then, again the control is returned back to the fast model and it then executes the next instruction and the process repeats. The process of callback registering starts and we would be getting all the value index for the bit fields within our API.

We declare variables and get the values from the index fields into our variables. Once we receive the values into our variables the process becomes simple and straightforward. From the variables, we populate the data structure will all the values obtained from data fields from a set of API's.

There is a functionality called as trace source function, where we add our coverage functionality for the particular API. Once we do the functionality within the trace source function during the entire test execution, we in the end execute the Display function i.e. a part of the release function. The release function is the last function to get executed before the plugin framework is unloaded from the fast model. So, within the Display function, the final print to file is added where total number of coverage hit count is mentioned.

At the end after the entire execution, check the output where the coverage hit count is being written. This is the entire flow of the entire CPU Software Coverage Plugin framework developed.

3.4 Data Structure implementation as Framework Base

3.4.1 Developing the Infrastructure

Before setting up the data structure, there needs to be the supporting infrastructure required for the loading of the plugin within the fast model. It is necessary to prepare the infrastructure first. The infrastructure requires creating the value indexes of the data fields within the set of the API inside of the Plugin class. The value indexes are used to refer to the index of where specific data fields are present. Apart from the value indexes, we need the declaration of the thunk functionality as a part of the infrastructure.

Thunk functions are intermediary functions which are first called by the model and the thunk function in turn calls our trace source functions. The thunk function declaration is considered from within the template and implemented similarly. There needs to be the declaration of each of the trace source API function where we are getting the field along with their value indexes along with necessary error checks. One we receive the data fields along with their value indexes, then adding the coverage event can be planned. There needs to be separate set of functions within our class for the loading and un-loading of the plugin.

3.4.2 Developing Data structure

The infrastructure is something that needs to be prepared whenever we have a plugin to be loaded with the fast model during the architecture execution and it is unloaded when the execution of the architecture is completed. One way to perform coverage is by making plugin for each of the architecture event I.e. a separate infrastructure needs to be developed every time for individual event-based plugins.

The better option is to prepare an infrastructure once, and then add multiple events on the top of it without repeating the infrastructure part. This can be done by setting up a data structure where multiple API are part of the data structure. With this, the infrastructure needs to be set up only once.

There are 2 data structures prepared. One data structure is implemented as 2-dimensional array where 1 dimension represents the per core details I.e. CoreNum field and 2nd dimension represents Instruction window where we maintain a set of instruction within our structure to plan out events from them. This data structure is specifically prepared from the API trace sources having CoreNum data field as using it, the structure as a 2-dimensional format is maintained. The examples of API with CoreNum data field are INST, CPSR, EXCEPTION, ExceptionReturn, CoreRegs64, etc.

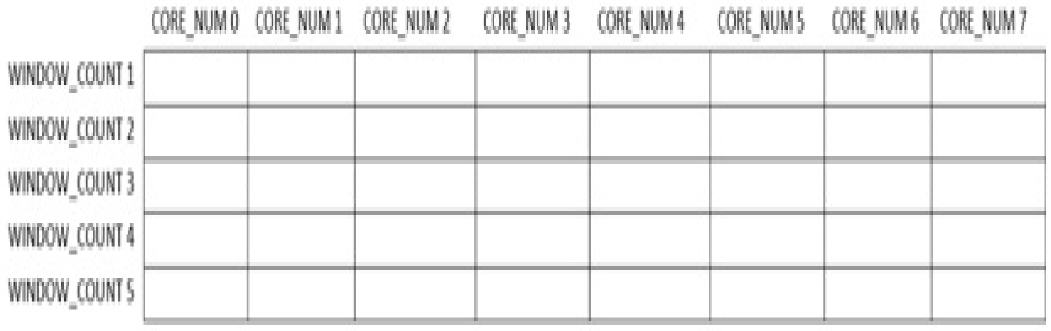


Figure 3.4: 2D Array Structure

Another data structure is such that it only has 1 dimension, where we only maintain the number of instructions and not the per core details. This is because the structure is prepared from the API trace sources which does not have CoreNum field. The examples of API without CoreNum data field are LOADS, STORES, MmuTlbFlush, etc.

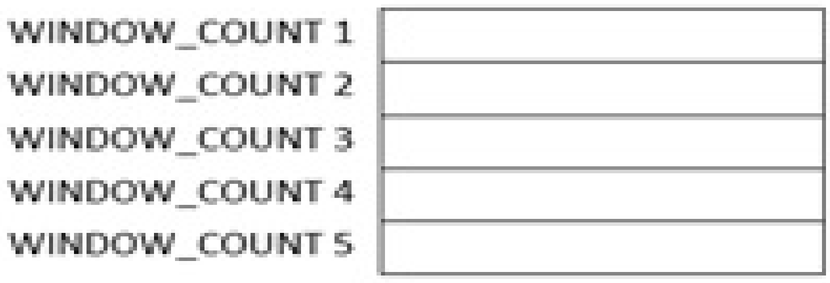


Figure 3.5: 1D Array Structure

Both the structures are implemented in the form of double ended queues I.e. deques. This is using the Standard Template Libraries as a part of cpp. As a part of a double ended queue, we maintain the structure by making sure we have a set of instructions within the queue. We set a Count limit of how many instructions can be a part of the deque at a time. This is an input configurable parameter. Let's say as a part of the software coverage framework, we have kept the Window Count to 5. This means API data would be push back to the structure for 5 times consecutively. As soon as the push back comes 6th time, as the deque can only contain 5 elements at a time, the first element inserted into the queue would popped from the front. So, at any point of time there are 5 elements as a part of the deque.

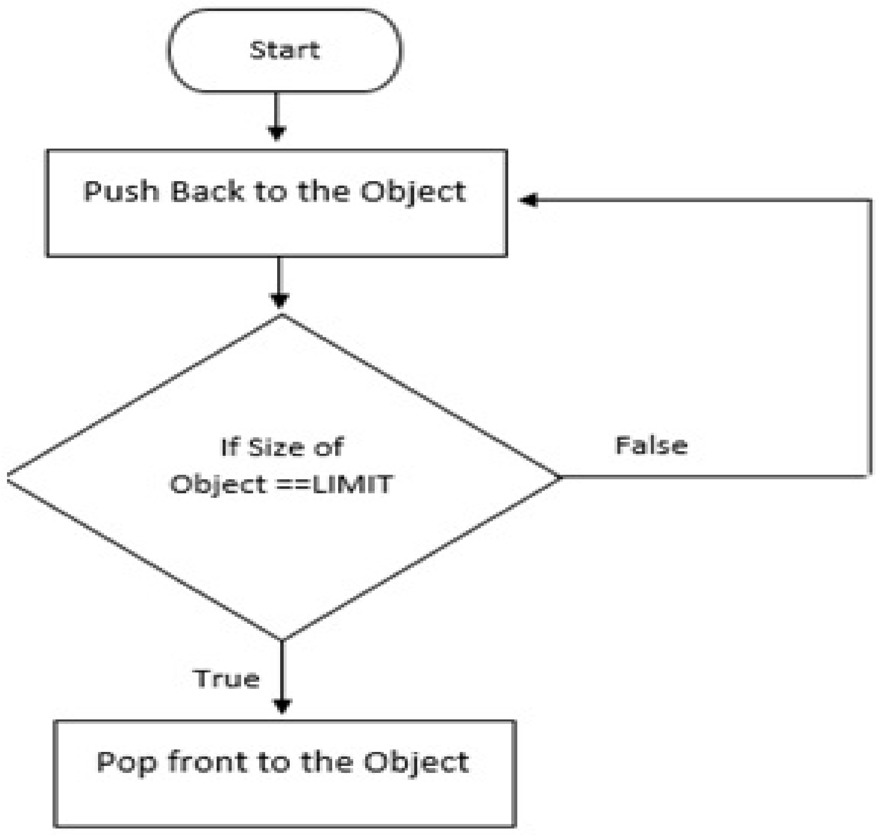


Figure 3.6: Structure: INST Source

As shown in the above figure, the structures are operated differently. For the structure with INST source, it is used to extract values from the variables element by element into the data structure fields. Once the values are inside the variables, push back of the entire object is done. This works for INST API trace source. For rest of the API trace sources, we only manage the last element being inserted which is as shown below:

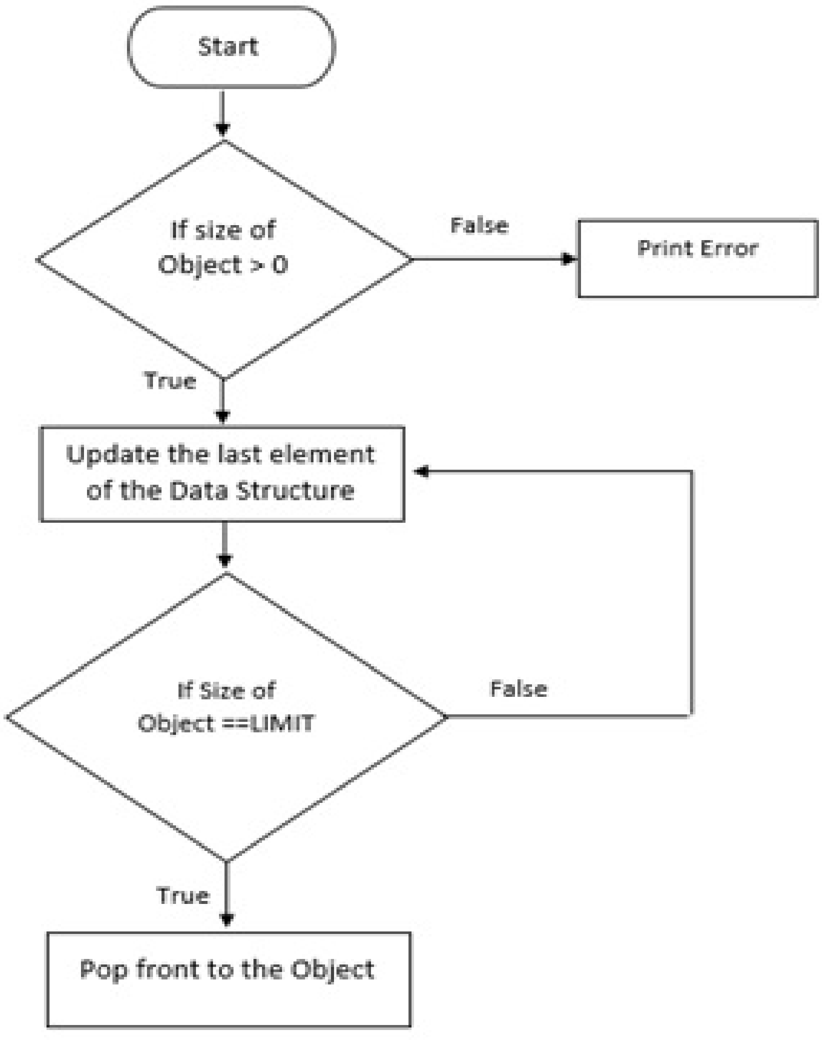


Figure 3.7: Structure: CPSR Source

For rest of the API sources for INST structure, we access the data structure as per the last element being entered into it. In case of planning out an event, conditional statements can be used specifying the size limit which when exceeded, then only execute the function. The data structure operation is separate for each API and is within trace source API functions of each respectively.

The data structure acts as an intermediate layer and event addition is done on the top of it. The event planning becomes simple as the infrastructure is already prepared for the data structure to work.

Chapter 4

Architecture events for coverage:

4.1 List of Architectural Events

Once the CPU software coverage framework is ready, architecture events can be planned using the framework. The set of events covered as a part of Coverage work are:

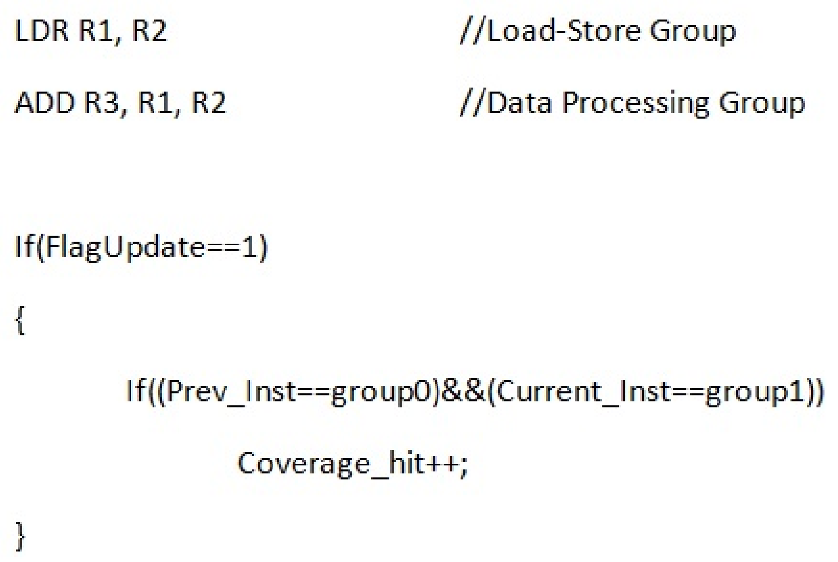
1. Flag Updates based on Instruction Grouping
2. Page Crossing for Load and Store Instruction
3. RAR Hazard for different access width
4. VA Aliasing
5. L1, L2 Bank access details
6. Global Monitor and Local Monitor details
7. WAR, RAW, WAW Hazards
8. Flag updates

4.1.1 Flag Updates based on Instruction Grouping

Dividing the entire instruction set in form of groups. As different instruction groups have different instruction queues as it takes different number of cycle depending on which group the instruction belongs to. There are instruction groups mentioned within the ARM architecture reference manual. The instructions have their own encoding based on which group it belongs to. The different groups can be considered majorly as SIMD, Floating Point, data processing, system instructions, branch group, Load, Stores, etc.

Now, if a group1 instruction gets executed after a group 1 instruction, there is no problem as the number of cycles taken are the same. The real scenario is when a group 1 instruction gets executed after an instruction not belonging to group1. This is because the number of cycles taken by both the groups is different. In this scenario, a stall may be required to make sure incorrect value is not being read. This is where the pipeline issue comes into picture and we use the register renaming functionality.

In this particular coverage functionality implementation, flag updates are checked for instructions belonging to two different groups and executed sequentially. For this event, first step is to make sure whether any of the individual flag is getting updated or not. If true, then different functions have been created according to the group it belongs to. For any instruction set which updates the flag- for the corresponding instructions as per their encoding, each of the different functions is checked. Once the group of the instruction is known, similarly based on index positioning, next instruction group is checked. On both the previous and current instruction indexing, it is made sure that which of the two-different group the instruction belonging to executes. Once known, coverage is hit for flag update based on the instruction grouping.



```
LDR R1, R2                      //Load-Store Group

ADD R3, R1, R2                  //Data Processing Group


If(FlagUpdate==1)

{

        If((Prev_Inst==group0)&&(Current_Inst==group1))

                Coverage_hit++;

}
```

Figure 4.1: Instruction Grouping

As shown in the example above, the first instruction belongs to the load store group. The next instruction I.e. ADD belongs to the data processing group. For this instruction group, if there is flag update, coverage is hit for the particular set of instruction.

4.1.2 Page Crossing for Load and Store Instruction

This event is planned based on the Load and Store API sources. The bit fields are access size as well as physical address and virtual address. Task is to determine whether for a given load or store instruction, page crossing is happening or not. The first thing to obtain is the page offset. Offset bits are the bits that represent each memory address in a page table. Let's say that the compiler is generating address of 32 bits for a 32-bit processor. That means 32-bit virtual address space. For this case, ideally 4 GB of RAM would be required. For a 4KB page size (a configurable option), either for the virtual address or physical address there is last 12 bits of page offset. The last 12 bits would be having one to one correspondence i.e. between the virtual address and the physical address. For a 32-bit address, the first most significant 20 bits of the virtual address correspond to a virtual page number which will be mapped to 18-bit physical page number by a page table.

From the virtual address, the offset bits can be obtained by doing modulo operation. Once the offset bits are received for a particular instruction, obtain the access size from the offset bits by subtracting from the page size i.e. 4k

Once the access size is received, from data field of size determine whether page crossing is happening or not for given instruction by comparing the size and the access value.

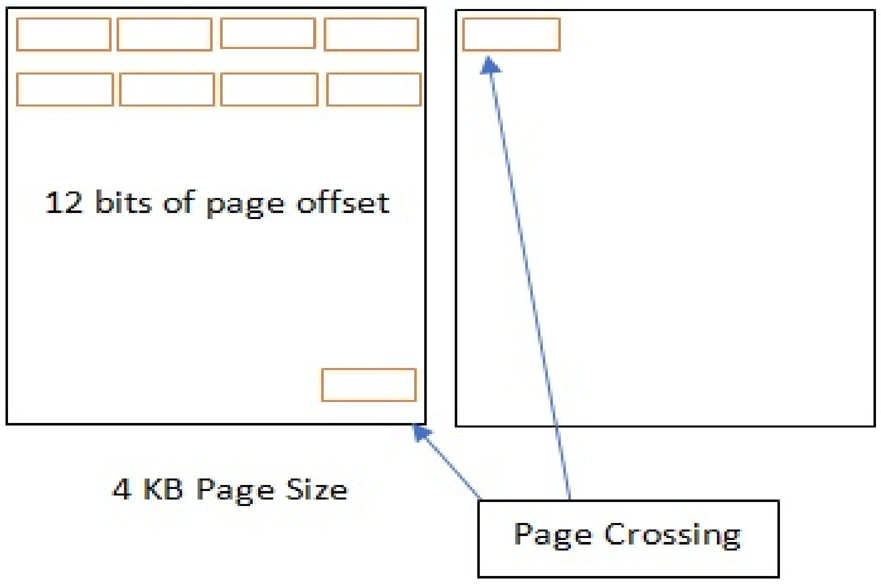


Figure 4.2: Page Crossing

4.1.3 RAR Hazard for different access width

The read after read hazard is a multi-CPU scenario. Different excess width means seeing page crossing for one instruction which means that particular instruction would be stalled but the next instruction would not be stalled which may be accessing the same address but a half word lying under the same page. In this case if a store happens in between from another CPU, there is a chance of reading the incorrect value.

Consider an example of loads instruction with different access width back to back. For the first load, in case a page crossing happens, stall is introduced and the page table needs an update from the TLB. In turn 2nd instruction gets executed and if it lies within the same page boundary, then this is a possible RAR hazard scenario.

4.1.4 VA Aliasing

There are mapping of every virtual address to the physical address. Virtual address aliasing means multiple virtual address mapping to the same physical address. This scenario is more useful when happen across multiple test cases. One thing to be made sure of is that the TLB invalidate should not happen otherwise all the entries would be removed. There are specific test cases where TLB invalidate happens at the end of test case.

So, whenever this scenario happens, increment the counter and in case a TLB invalidate happens then store the count value and check against the next instruction.

4.1.5 L1, L2 Bank access details

This is to identify which bank is being accessed by the set of instructions. This is a check for stream of back to back instructions accessing the same bank. This is

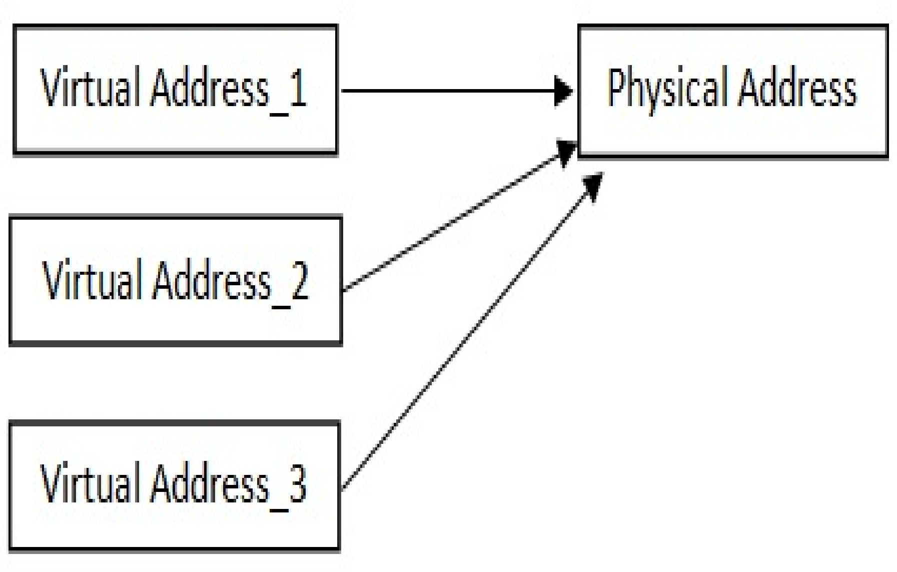


Figure 4.3: VA Aliasing

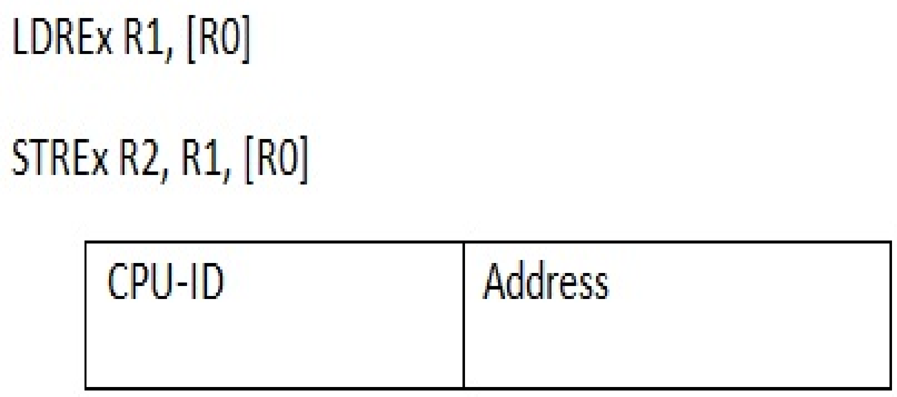
to make sure are we able to stress on a single particular bank or not. For this using the Physical address, out of the total 64 bits of the physical address, some bits give the details about which particular bank is being accessed.

For a set of instructions, from their respective physical addresses it can be determined that whether multiple physical addresses are accessing the same bank or not. This is a straightforward functionality to add and check for coverage.

4.1.6 Global and Local Monitor details

This is the state of the monitor. There are atomic operations which are exclusive I.e. for a single CPU a set of operations performed example: a load operation happens and i.e. followed by a store operation. But in a multi CPU scenario, multiple CPU will try to do a load and store on the same address. To avoid it, there is an exclusive state within a CPU to perform specific operations where no other CPU tries to interrupt on the same address.

For example:



```
LDREx R1, [R0]

STREx R2, R1, [R0]
```

CPU-ID	Address

Figure 4.4: Local and Global Monitor

Here, in above example, the LDREx instruction loads from the memory a word and it then initialize the monitor state to being exclusive. This is to synchronize the entire process. The above example shows that load exclusive is performed from the address in R0 and that value is then placed into R1 which then in turn updates the exclusive monitor.

The next instruction STREx instruction is forming a conditional store of a word to memory. If exclusive monitor perform a store the memory location gets updated and it will return a value 0 in the destination register. If the store is not permitted, then memory location is not updated and a value 1 is updated in the destination register. This indicates success or failure in R2.

4.1.7 WAR, RAW, WAW Hazards

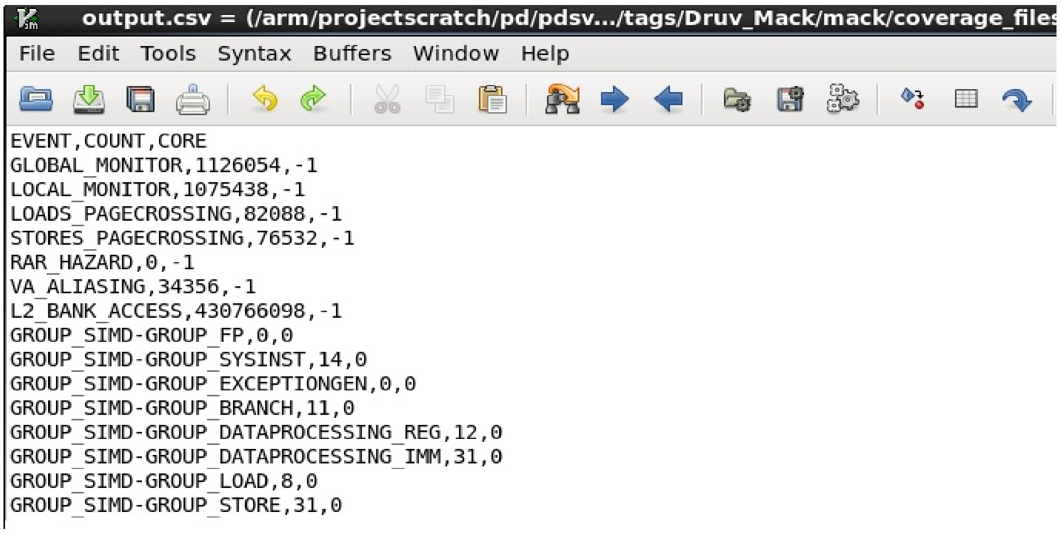
There is an API called as CoreRegs64 where there is a data field called as register Id. The register Id gives the details about the source register and the destination register being accessed. From a set of instructions within the structure, dependency between the register can be calculated based on the register Ids. The total number of count gives total number of hazards present within a given test file.

4.1.8 Flag updates

This is a straight forward coverage event where given a test, all the individual flag updates are calculated without any kind of conditions. The counter is incremented whenever any individual flag gets updated from 0 to 1.

4.2 Output Coverage Data

The output coverage data is written in the file as a part of the main source file. The events are added in the form of a csv file I.e. comma separated values. The count field mentions the number of times coverage event is hit within a particular test case. The idea is to make the output format in such a way that it is a part of the existing coverage flow.

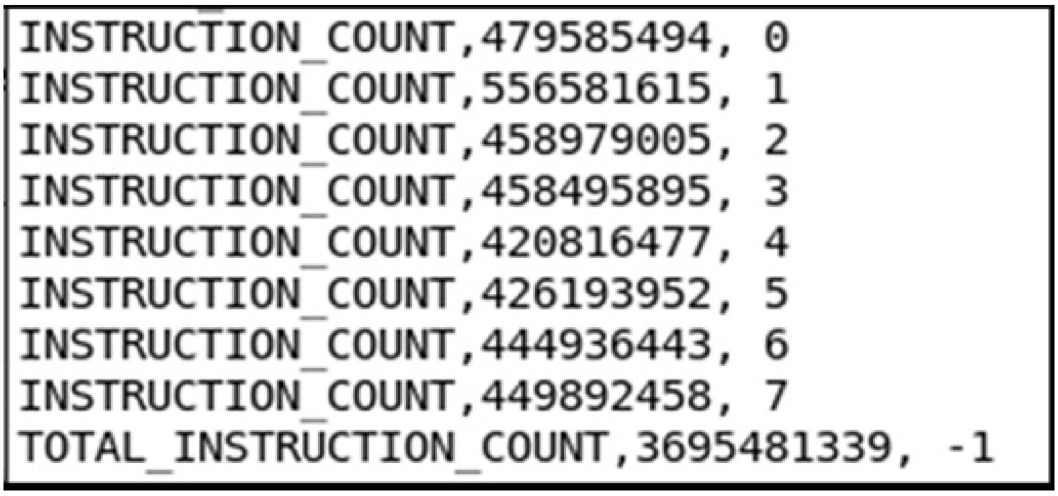

output.csv = (/arm/projectscratch/pd/pdsv.../tags/Druv_Mack/mack/coverage_files
File Edit Tools Syntax Buffers Window Help

```
EVENT,COUNT,CORE
GLOBAL_MONITOR,1126054,-1
LOCAL_MONITOR,1075438,-1
LOADS_PAGECROSSING,82088,-1
STORES_PAGECROSSING,76532,-1
RAR_HAZARD,0,-1
VA_ALIASING,34356,-1
L2_BANK_ACCESS,430766098,-1
GROUP_SIMD-GROUP_FP,0,0
GROUP_SIMD-GROUP_SYSINST,14,0
GROUP_SIMD-GROUP_EXCEPTIONGEN,0,0
GROUP_SIMD-GROUP_BRANCH,11,0
GROUP_SIMD-GROUP_DATAPROCESSING_REG,12,0
GROUP_SIMD-GROUP_DATAPROCESSING_IMM,31,0
GROUP_SIMD-GROUP_LOAD,8,0
GROUP_SIMD-GROUP_STORE,31,0
```

Figure 4.5: Output Coverage

As shown a sample output above, the first column lists out the set of events and the second column represent the count for each of the event respectively. The third column gives the core value. For all the events which are based out of structure not containing core number, we define the events as global and the value assigned to them is ”-1”.

Further to have proper coverage analysis, there is total instruction count to have a ratio of how many times the coverage is hit as compared to the total number of instructions being executed. This detail is per core as well as the total of all the cores. The example is shown below:



```
INSTRUCTION_COUNT,479585494, 0
INSTRUCTION_COUNT,556581615, 1
INSTRUCTION_COUNT,458979005, 2
INSTRUCTION_COUNT,458495895, 3
INSTRUCTION_COUNT,420816477, 4
INSTRUCTION_COUNT,426193952, 5
INSTRUCTION_COUNT,444936443, 6
INSTRUCTION_COUNT,449892458, 7
TOTAL_INSTRUCTION_COUNT,3695481339, -1
```

Figure 4.6: Instruction Count

4.3 Loop generation script

Using the loop generation script, the script continuously runs test cases unless interrupted. Every time a test case is run, a unique configuration setting is selected I.e. stress on some architectural event as compared to other events. With multiple test cases, coverage data across multiple files is collected which is better for doing analysis.

The data can either be merged as per the unique knob setting exercised for each of the test cases. The data can be merged using a merging script and can be analyzed by preparing a graph out of it as shown below:

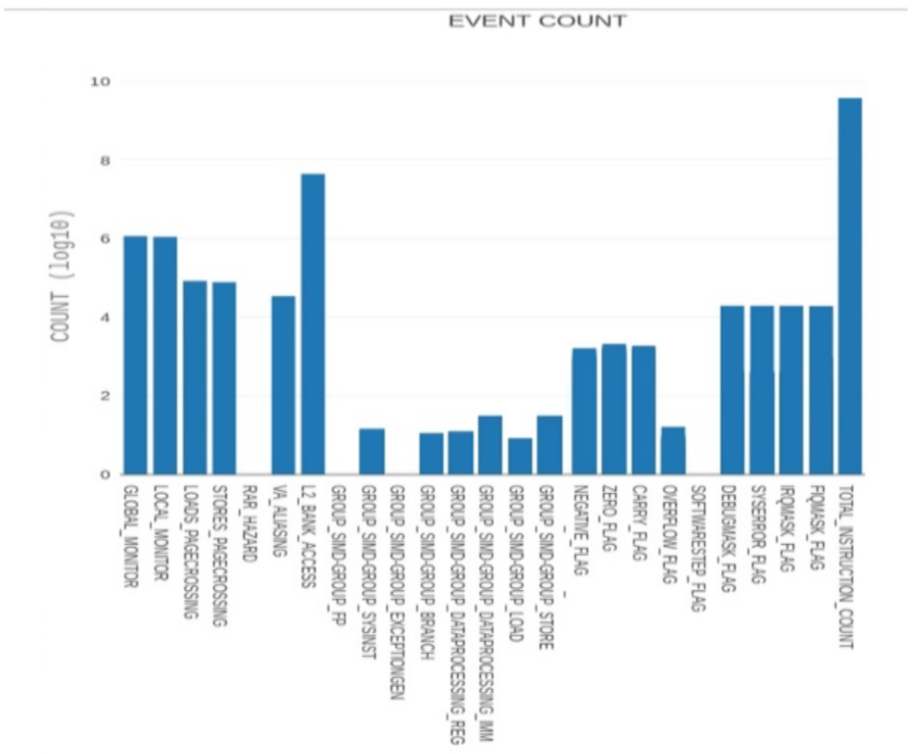


Figure 4.7: Data Coverage Graph

The above graph is prepared as a part of merged output that was run across all test cases. The graph is on a log base 10 scale as the total instruction count is very large in number if implemented on a linear scale. Here it is visible that the occurrence for the event RAR hazard is null for the given number of test cases whereas the event for L2 Bank access shows maximum number of coverage hit.

The graph is still not showing the details of the coverage event per core. There needs to be a three-dimensional graph over which one axis would represent the core and event occurrence can be checked per core. The core details are very important when considering the multi-threading environment where each core can be considered as a thread.

Chapter 5

Architecture Profilers

5.1 Overview

Architecture Profilers are analysis tools to obtain more information about the tests. As a part of the AVS i.e. Architecture Validation Suite, there are bunch of test cases to check whether Architecture is compliant to the specification or not. The test cases exercise on different features of the architecture for example: Processor core, memory, memory management unit and so on. Sometimes, due to some issue, test cases might fail, and, in such scenarios, it is very important to know the reason why the test case is failing. In both the above situation, a tool is required to know what all features a test case is touching upon.

Memory mapped I/O means mapping of the device registers to the fixed addresses in the normal memory space. There are reasons to keep registers or other special purpose inside the conventional memory space. One is that it is costly to have many registers separately. Secondly, it is not always feasible to have so much space dedicated for the registers and it increases complexity as well. With the increase in functionality need for having more registers arise.
Solution is to have memory mapped registers. The memory mapped registers means

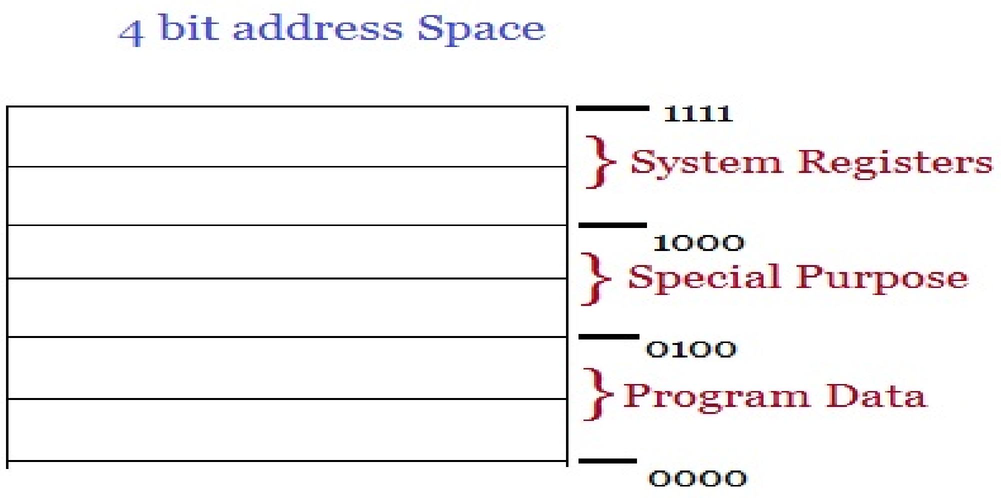


Figure 5.1: Sample Memory Mapped Region

region dedicated to function as a register in the normal memory space. The access to the register i.e. read and write occurs as same as for memory access like load and store operations. Whenever any kind of write operation happens to this memory region, based on which register region, functionality will be triggered. The memory mapped registers are dedicated to perform some functionality whenever a write happens to them.

Memory Map interface is the interface used to trigger the memory map registers. As a part of running the architecture test suite, interrupts are generated during the architecture execution by writing to some of the memory mapped regions. These memory mapped regions are triggered using the memory map interface. These memory mapped registers are called as configuration registers.

5.2 Block Diagram

The profiler development is done on the top of abstraction layer called as the Trace Interface layer. The functionalities are developed individually over the Profiler layer. It is done as shown in the below block diagram.

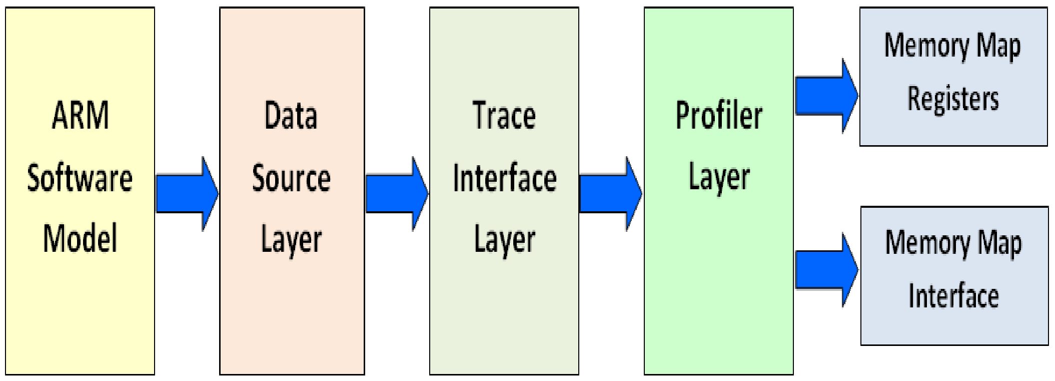


Figure 5.2: Block Diagram: Architecture Profiler

The data source layer is the access to the APIs that are exposed during the execution of the Architecture. Out of all the APIs, important APIs are extracted into an existing framework called as Trace Interface layer. This layer helps as a base layer, an abstraction over which many different tasks can be performed. One such task is of a profiler. The profiler layer is based on the top of the trace interface layer. Using the single profiler object, profiler components like memory map registers and memory map interface exist. The profiler layer is implemented as a Singleton design pattern. If the profiler tool is used, single profiler object is created at run time and all the components for profiling are either enabled or disabled by the user.

User can either enable profiler individually or can enable a master switch to exercise on all the profilers. Within each member function, iterate over and comparing the address obtained from the AEM to the address present in the structure. This is what is done for the memory map interface.

The Profiler information obtained is reported in the log file. The reporting is in such a way that specific keyword like: PROFILER is reported along with the component name.

5.3 Flowchart: Architecture Profiler

Profiler is enabled as a part of the input parameter before running the architecture execution. During the execution, Profiler object is created and along with that, the functionalities to be traced as a part of profiler is also enabled. During the architecture execution, the profiler checks on the events and reports it in the log file.

Further, a script file is written to extract all the statements with the keyword PROFILER. The keyword is searched and then parsed separating and dumping into another file.

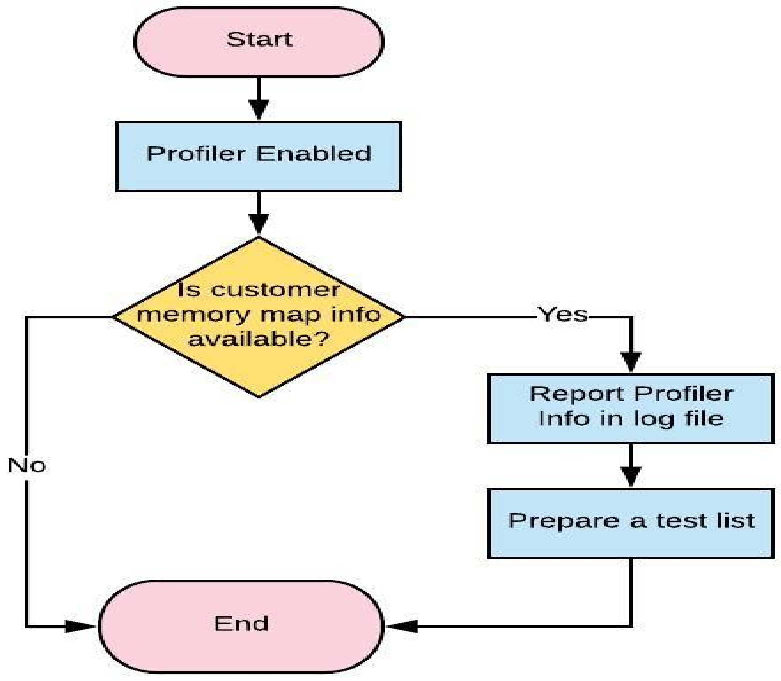

Start
Profiler Enabled
Is customer memory map info available?
Yes
No
Report Profiler Info in log file
Prepare a test list
End

Figure 5.3: Flowchart: Architecture Profiler

The above graph is prepared as a part of merged output that was run across all test cases. The graph is on a log base 10 scale as the total instruction count is very large in number if implemented on a linear scale. Here it is visible that the occurrence for the event RAR hazard is null for the given number of test cases whereas the event for L2 Bank access shows maximum number of coverage hit.

5.4 Unit Testing

Unit tests are written to prove that a feature of the code developed is working correctly. For profilers, it is checking of the logic and actual simulation is not necessarily required. What needs to be done is to mock up the stimulus. The interface is Model View Controller which conveys that view part can keep on changing, so the unit test environment is a view, AEM is another view. It can come from any source but the profiler code doesnt change. So, as long as the profiler code is working with the interfaces, it can be tested as a unit test. There is no dependency on AEM.

Chapter 6

Input configuration parameters

6.1 Segregating address range

We have multiple configurable input options to be specified within the command line. The options are specified by adding a plugin option as -C and then enable or disable it.

Check for pc limit

For separating out the test part and the kernel part for test file. The address range of kernel part is different as compared to range of test part. For example, kernel part lies from 0x0000 to 0x3FFF and test part lies within 0x4000 to 0x7FFF address range.

6.2 Segregating on specific functionality

6.2.1 Enable/Disable Events

For selecting which event to target on, we enable or disable the following variables with respect to the event it covers:

The reason is only if we want to exercise on a certain event, we do the entire process of populating the data structure and perform the functionality. In case the event is not enabled, the data structure would not be populated for a given API trace source. The above-mentioned variables need to be set to 1 to exercise on the event.

6.2.2 Enabling populating of data structure for specific API

The below API trace source functions are not populated as there are no events which are planned out of them in the existing framework. They can be enabled in case any event is planned out from the APIs or it can be removed as well from the source file if needed: basically, it is a conditional check before executing the trace source function.

Once the run is complete, the output file in csv (comma separated value) format with the name specified as a command would be produced by the plugin within the same directory. In case output file name is not explicitly entered as a part of the command, the plugin takes file name as SampleOutput.csv and dumps output into it.

Chapter 7

Conclusion

7.1 Conclusion

Test Suites are input stimuli given to the Architecture to verify whether it is working correctly or not. Testbench verification tools help to evaluate the quality of the input stimuli used for the verification of architecture. At micro-architecture level, functionality is important to verify which is achieved by using the Coverage framework tool and at architecture level, check for compliant against the specification is important which is achieved by the Architecture Profiler tool. As a result of using these tools, quality of input stimuli can be known and in turn more rigorous verification is achieved.

References

[1] “ARM Architecture Reference Manual v8 Profile A ”Beta, Arm limited, 2013

[2] “Programmers Guide for ARMv8 Cortex-A ”, Arm limited,2015

[3] “Computer Architecture: A Quantitative Approach ”, by John L. Hennessy and David A. Patterson (5th edition)

[4] “Containers Library”, http://en.cppreference.com, Date: 30/8/2017, Time:11:30

[5] “ARM Fast Models User Guide”, version 8.4, 2014

[6] ARM Internal Documents