Developing Arm Architecture Verification Tools and Solutions

Major Project Report

Submitted in fulfillment of the requirements
for the degree of

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

By

Khushnuma Jhunjhunwala

(16MECE06)

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

May 2018

Developing Arm Architecture Verification Tools and Solutions

Major Project Report

Submitted in fulfillment of the requirements
for the degree of

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

By

Khushnuma Jhunjhunwala

(16MECE06)

External Project Guide:
Mr. Saurov Kanti Shyam
Staff Engineer,
Arm embedded Technologies Pvt. Ltd.,
Bangalore.

Internal Project Guide:
Prof. Vijay Savani
EC Department,
Institute of Technology,
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.

- Khushnuma Jhunjhunwala
16MECE06

Certificate

This is to certify that the Major Project entitled **"Developing Arm architecture verification tools and solutions"** submitted by **Khushnuma Jhunjhunwala (16MECE06)**, towards the partial 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 her 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

Prof. Vijay Savani
Internal Guide

Dr. N.P. Gajjar
Program Coordinator

Dr. D. K. Kothari
Section Head, EC

Dr. Alka Mahajan
Director, IT

Certificate

This is to certify that the Major Project entitled **"Developing Arm architecture verification tools and solutions"** submitted by **Khushnuma Jhunjhunwala (16MECE06)**, 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 her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Mr. Saurov Kanti Shyam
Staff Engineer
Arm Embedded Technologies Pvt. Ltd.
Bangalore

Acknowledgements

I am profoundly grateful to **Mr. Saurov Kanti Shyam**, Staff Engineer at Arm, Bangalore for his expert guidance and continuous encouragement throughout the internship period in preparing me for the project of Developing Arm architecture verification tools and solutions.

I would like to express my sincere thanks to **Dr. N.P. Gajjar**, PG Coordinator of M.Tech Embedded Systems for providing me an opportunity to pursue my internship at Arm, Bangalore. Would also like to thank **Prof. Vijay Savani** for his exemplary guidance, mentoring and constant encouragement.

At last I must express my sincere heartfelt gratitude to all the employees of Arm who directly or indirectly helped me during the course of this work.

- Khushnuma Jhunjhunwala
16MECE06

Contents

List of Figures

Abstract

Processor architecture verification is the process of uncovering bugs in the design of the processor.The ultimate goal of architectural verification is to deliver the design to market as bug free as possible. It is one of the biggest challenge industry is facing today, as the verification effort is often more than the design effort.

"Random Instruction Sequencer (RIS)" tools are most commonly used across the processor design industry for verification and validation of processor design. Thus developing RIS tools would simplify the process of processor design verification which may prove helpful to considerably reduce time to product.

The work presented here introduces a unified target configuration generation tool, for auto generation of architecture configuration files used to configure RIS tool. It achieves significant reduction in resource utilization by automating the process of configuration file generation which is being hand coded otherwise. It also includes development of Pre-generation and Post-generation Quality Assurance Tools for the MP RIS Tool which is used for verification of multi-processor environment. The Pre-generation QA tool discards the invalid test cases before generation and hence increases the efficiency of the MP RIS Tool by saving time and resources being used to generate invalid test cases. Whereas the Post-generation QA tool does the analysis of the generated test case to subsequently reduce the percentage of overhead instructions added compared to actually required instructions in a generated test case.

Abbreviation Notation and Nomenclature

RIS ..... Random Instruction Sequencer
RTL ..... Register Transfer Level
ISA ..... Instruction Set Architecture
AEM ..... Architecture Envelop Model
ISG ..... Instruction Stream Generator
AVS ..... Architecture Validation Suites
DVS ..... Device Validation Suites
ACS ..... Architecture Compliance Suites
PEs ..... Processing Elements
ELF ..... Executable Linkable Format
GUI ..... Graphical User Interface
CSV ..... Comma Separated Value
SOC ..... System On Chip
IPS ..... Instructions Per Second
QA ..... Quality Assurance

Chapter 1

INTRODUCTION

1.1 Motivation

Verification is one of the biggest challenge industry is facing. The verification effort is often more than the design effort as the design complexity of the processor architecture is increasing these days [3]. "Random Instruction Sequencer" (RIS) tools are most commonly used across the processor design industry for functional verification and validation of processor design. These tools are used most of the times to find bugs in design for a stable RTL (Register Transfer Level) design that is not at a matured level. RIS tools effectively generates test scenarios that are hard to anticipate. These completely random sequences of instructions prove to be useful for exploring corner cases in the RTL design. These sequences are more effective when the bugs are related to a sequence of events that happen in a short time span [1].

The verification results depends upon the quality of results produced by the instruction sequence generator, the availability of time and also the number of computational resources available. The time and resources are limited most of the times so we need to tune the instruction sequence generator i.e. the RIS tool to target specific areas in the design [1].

1.2 Objective

RIS tools are popularly recognized as an efficient approach for verifying corner cases for processor architecture verification environment that are hard to anticipate. To cover most of corner cases RIS tools uses random test libraries.Thus improvements in randomization that the RIS tools take would help us to increases the hit rate of corner cases which are hard to acknowledge by writing directed tests for different scenarios.

The objective of this project "Developing Arm architecture verification tools and solutions" can be divided into various phases:

a. **Unified Target Configuration generation:** With Unified Target Configuration we look forward at developing a common interface which auto-generates different standard files in RIS tool acceptable format.

b. **Pre-generation Recipe QA:** With this approach we plan to improve the coverage of the test sequences generated by RIS tool by adding some directed aspects to the recipe specific knobs taking random values from a previously defined range of acceptable values.And also increasing the efficiency of the RIS tool used for verification of multi processor design by discarding the invalid test cases before their generation.

c. **Post-generation Recipe QA:** This approach is used to verify the length of the generated test cases against the anticipated length to ensure that the overhead added is at an optimum level to not decrease the efficiency of the random test generator. Thus the percentage of overhead added is first analyzed and then the instruction count is given such that we get higher number of useful instructions then the overhead instructions.

1.3 Outline of Thesis

This Thesis is put together in five chapters, a brief information about each of them are discussed below:

- **Chapter 2 : Arm architecture verification tools**
  An introduction to the Random Instruction Sequencer Tool and also description of how it is more effective than the Deterministic approach to cover corner cases.

- **Chapter 3 : Unified Target Configuration**
  Detailed description of the unified approach of generating the architectural configuration yaml file in RIS tool compatible format. Also includes the flow of taking specifications from the partner and storing it in a database and querying it to get appropriate values.

- **Chapter 4 : MP RIS Tool**
  This chapter includes the detailed description of the MP RIS tool and its working. It also includes detailed procedure of generating the test cases using various Arm tools and procedures. It defins various components and plugins used in the tool like the recipes, knobs, zones, the wrapper script tool and much more.

- **Chapter 5 : Pre-generation QA - Recipe QA**
  It describes in detail a newly developed plugin for Quality Assurance of the generated test cases from the MP RIS Tool. Also includes the level of randomization achieved by the architectural parameters and how to improve it to achieve better coverage. This chapter describes how this Pre-generation Tool helps to increase the efficiency of the MP RIS Tool.

- **Chapter 6 : Post-generation QA - Test Length Analysis**
  This chapter describes the Post-generation tool made for the MP RIS tool and its working. It gives a detailed explanation of why we require to analyze a test case after it is generated. Also includes the study of optimum number of instruction count to make the generated test case efficient despite of all the overhead instructions that are added to the test cases that are generated.

- **Chapter 7 : Conclusion and Future Work**
  Describes concluding remarks on how the Arm verification tools i.e. the RIS Tools are developed in this project to enhance their functionality. It also describes the proposed work that can be done in future.

Chapter 2

Arm ARCHITECTURE VERIFICATION TOOLS

2.1 Architectural Verification

Verification here refers to the entire process of uncovering bugs in the architectural design of the processor.The ultimate goal of architectural verification is to deliver the design to market as bug free as possible. It is the process of ensuring that a design of processor meets specified requirements. It reduces the cost of late bugs in a design by ensuring proper functional coverage of the entire design that is being tested. Thus it saves us from loss of revenue invested in respin of production samples.

The verification methodology at Arm verifies the RTL or the AEM (Architecture Envelop Model) against its ISA (Instruction Set Architecture). Here the AEM is a highly configurable fast model of Arm processor. The verification process has to fill in the gap between the architecture specification given by the ISA and the detailed processor implementation [4]. Thus there needs to be a mapping between the architecture level and the micro architecture level of the processor which is useful in the process of verification of the processor design.

The verification of microprocessor architecture design relies majorly on simulation based techniques. These stimuli are then executed on RTL design model of the processor architecture or on AEM and then the results are analyzed to check the correctness of the design. The stimulus are the sequence of instructions which may be generated using automated tools which are commonly known as Instruction Stream Generators (ISG) [5].

The stimulus for verification of arbitrary architecture design can be deterministic, random or constraint-based stimuli [5].

- **Deterministic simulation**
  This methodology is the most common way of generating test stimulus.The two ways for generating test cases for deterministic simulation are:

  1 AVS (Architecture Validation Suites) and
  2 DVS (Device Validation Suites)

  AVS focuses on checks for architectural functionality whereas DVS focuses on behaviour of a specific core [3]. Thus AVS does the checks on ISA and the exception model that is implemented. Its working is as shown in figure 2.1. On the other hand DVS checks:

  1 IMPLEMENTATION DEFINED features
  2 Scenarios dependent on timings
  3 Scenarios related to stress testing and
  4 Scenarios which are not covered fully by AVS

  These test suits are the Arm "Architecture Compliance Suites" (ACS). The limitation that they have is due to the effort that is required to generate the test cases and the performance that is achieved by the simulation tool [3].

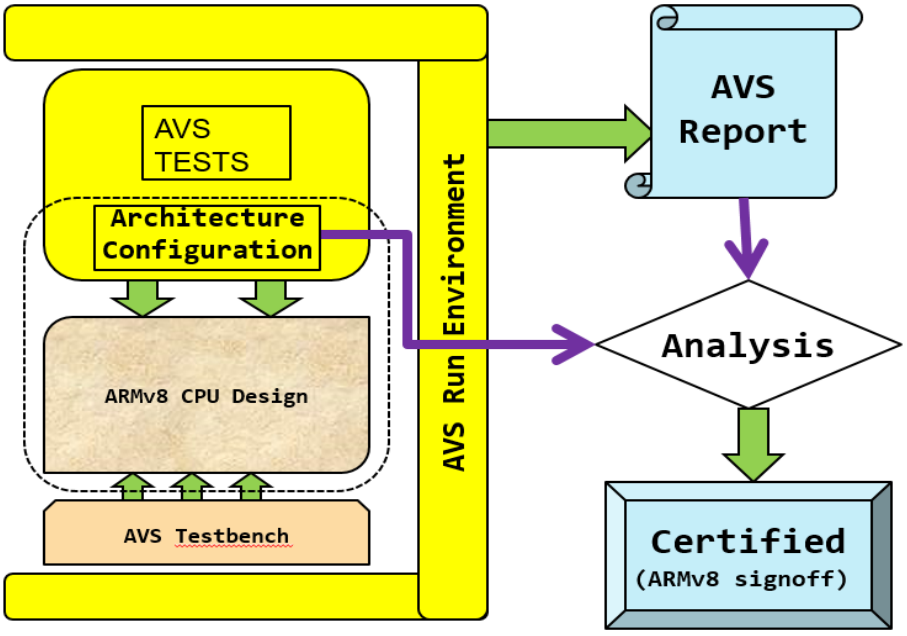


Figure 2.1: Arm compliance framework

- **Random stimulus generation**
  It is most effective way of verifying corner cases that are hard to anticipate using the deterministic simulation approach. With the help of templates describing generation tasks RIS tools generate random stimulus which are nothing but a sequence of random instructions [3].

- **Constrain Based simulation**
  These type of simulation is not truly random but cover certain areas of functionality in a systematic way keeping everything else random [5]. Addition of semi-directed instruction sequences that are deterministic in nature increases chances of covering desired corner cases with less investment in time [6].

2.2 RIS and MP-RIS Tool

The functional verification of the microprocessor design or the SOC (System On Chip) is one of the most critical phase of the entire design life cycle as it takes two third of the entire design cycle time. So we need to develop efficient tools in terms of time and completeness.Thus various methodologies proposed for function verification of the processor are as explained below [8].

An approach to test program generation known as Model Based Test-Generation, allows the complex knowledge of testing to be incorporated in the logic of model itself.Then this knowledge base is used in conjunction with the architectural model to generate test cases. Another test generation methodology assumes that the netlist of the processor is available and it requires the library of macros to be developed. These macros are written manually in such a way that they are able to excite all functions of the processor [8].

RIS generator tools are used to generate sequences of random instructions which act as stimuli for verification of processor design. The MP-RIS tool is the RIS tool that generates instruction sequences for multiprocessor systems. These instructions focuses on the data sharing and memory coherency in the multiprocessor environment.

The instructions can be focused uniquely to one processor or the multiprocessor environment where virtual to physical memory map or address and data space is shared. By defining such constraints we can generate different level of traffic specifying the interactions between processors and memory subsystem for multiprocessing elements. Thus using this approach the data coherency in a multiprocessing environment is stress tested [6].

RIS tools are highly configurable tools which provide options to generate test scenarios stressing some particular functionality of the processor. This can be done by providing weight to instructions as shown in figure 2.2. Thus the test sequence generated have some specific subset of the Instruction Set (ISA) that can be used to stress out some particular functional area of the processor [6].

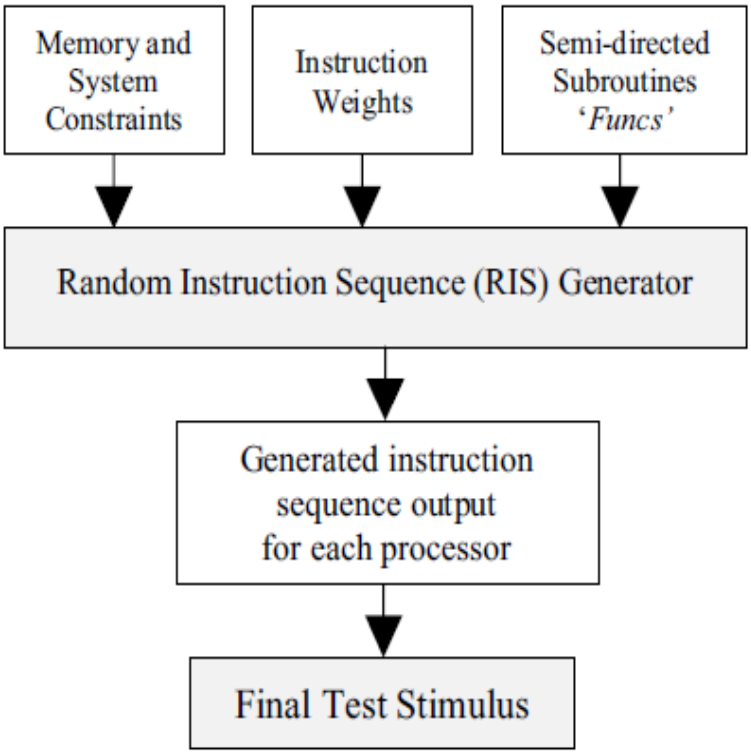


Figure 2.2: Working of RIS Generator [6]

RIS tools takes various configuration files as inputs to generate test sequences targeting one particular area of processor that needs to be stressed. They are described as below:

- **Memory and System Constraints**
  This gives information about the memory space that is available for generator, attributes of cache like how many levels of cache are implemented and their size, virtual memory organization, system memory map and the configuration of the knobs i.e. the values of the architecture specific parameters [6].

- **Instruction Weights**
  The user gives weights to instructions or group of instructions based on the intent of the test and the final instruction sequence that is generated has the distribution of instruction based on these weights [6].

- **Semi-directed Subroutines**
  These semi directed codes are targeted to test a specific processor functionality like linecross. This directed chunk of code is called funcs. They target some specific scenarios in the processor [6].

For multiprocessor systems environment stress testing can be done by a very common method of cache load store operations or writing to a shared memory between different processors. Thus the address generation must be such that the data is shared between multiple processors. Thus for memory coherency testing the MP-RIS generator have a memory model where each processor from a cluster have private memory region to which only that specific processor can write during the test and there is a shared memory region which can be accessed by more then one processing element [6].

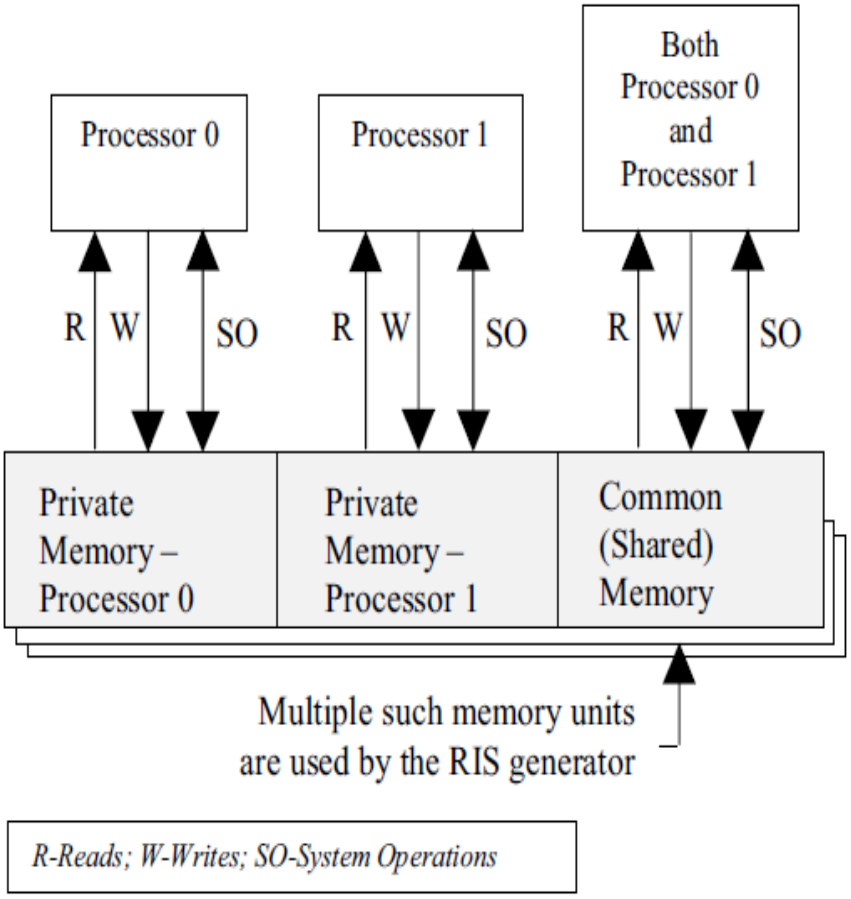


Figure 2.3: Memory organization for RIS Generator [6]

The figure 2.3 shows two processing elements each with private memory region and sharing one common memory region. Now different random instructions trigger processors to access these regions. The number of processing elements PEs accessing the shared memory are configurable for a RIS generator.

Now the generated instruction sequences are in Executable Linkable Format (ELF) which can be built using Arm tool chain to get executed. These instruction sequences are fed to either AEM model i.e. the software model of the processor or they are run on the processor RTL for verifying them.

Chapter 3

UNIFIED TARGET CONFIGURATION

3.1 Background

Both AVS and RIS tool require architecture configuration files for different implementations as an input to generate different test cases. Target configuration files captures this architecture specific configuration parameters and their values. Partners are approached multiple times by Partner Enablement team and tool engineers for generating these target configuration files. And then these files are hand coded referring to the engineering specifications. The input structure of the RIS Tool is as shown in figure 3.1

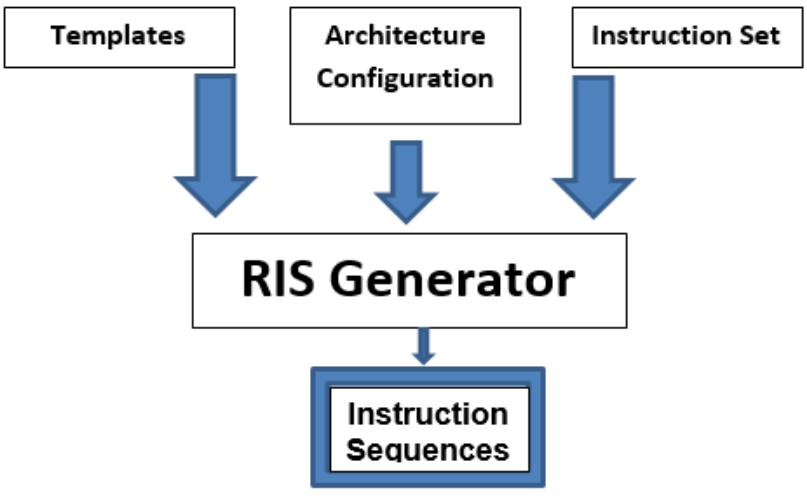


Figure 3.1: Architecture configuration as input to RIS tool

3.2 Problem Statement

- Partners need to be approached multiple times for same target configuration data but in different format and for different use cases.
- There is no automation for both AVS and RIS tool, thus configuration files are hand coded. This leads to various problems as listed below:
  - Human error
  - Tedious job
  - Requires more resources
  - Takes more time
- Future maintainability is a problem.
- Multiple files needs to be changed to reflect changes in architecture specifications.

3.3 Challenges

- A single automated application.
- Generates the configuration as required by RIS tool and AVS, without changing present output format.
- Syncing and Maintainability across several teams (DV engineers, Partner Enablement team, RIS tool engineers)
- No inter dependency across teams.AVS and RIS teams should be able to make progress without info specific to the others being filled in.
- Future maintainability must be possible.
- Realistic value based on the existing methodologies and Partners response.
- Would need to meet some key real-world requirements like handling incomplete information.
- Being extremely quick and easy to hack in new requirements.
- Would need to be updated regularly and kept in sync across all active projects.

3.4 Proposal

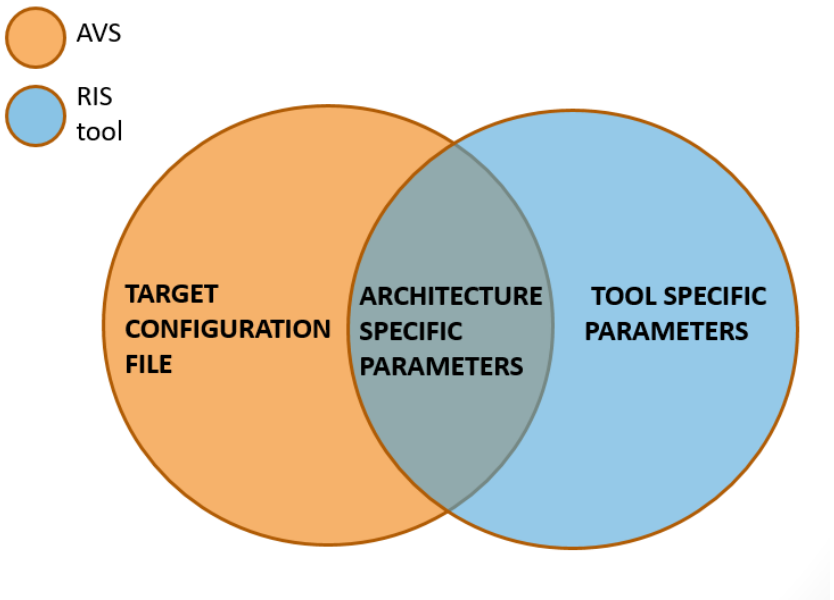


Figure 3.2: Unified Target Configuration

The architecture configuration yaml file that is fed to RIS tool consists of 2 parts 1) Tool specific parameters 2) Architecture specific parameters. After studying the target configuration file it is established that the architecture specific part of the architecture configuration yaml file that is fed to RIS tool can be derived from the target configuration file. This can be realized using a venn diagram as shown in figure 3.2

The parameters in the architecture specific of the yaml file parts can be obtained from target config file in two ways:

- Some parameters of the yaml file have a direct mapping to the parameters of target config file just with both the parameters having different names.
- Some parameters of the yaml file can be derived from the existing parameters in the target config file with the help of some equations.

3.5 Implementation

The target configuration file is parsed to a resource configuration user interface software tool which pushes the values of the parameters in a database. So to implement unified target configuration we need to auto-generate the yaml file containing architectural configuration in RIS tool acceptable format. Thus we need to follow certain steps as listed below:

- Parse the Target Configuration file to the resource configuration user interface software tool so that it gets the value of the parameters.
- Add equations in the tool for the derived parameters. This is one time configuration in the tool.
- Now query the database of the tool to get the value of the desired parameter.
- Now map the name of the parameter with the name that is accepted by RIS tool.
- Assign the value of the parameter fetched to the RIS tool parameter.
- Print the List of architecture configuration parameters in yaml format and in order acceptable by the tool.

3.5.1 Resource configuration user interface software tool

This software is used to collect the architectural configurations from the partner and the working of this tool is explained by figure 3.3. It is in development stage, but once it is developed then partners can directly provide architectural configurations through a GUI which would in turn flood the database with values to the specific architectural parameter. So now instead of Parsing values through GUI we parse target configuration file to the software which would later on generate this target configuration file.

Now we use this software to integrate equations of the derived parameters and to calculate their values and also query the values of directly mapped parameters.

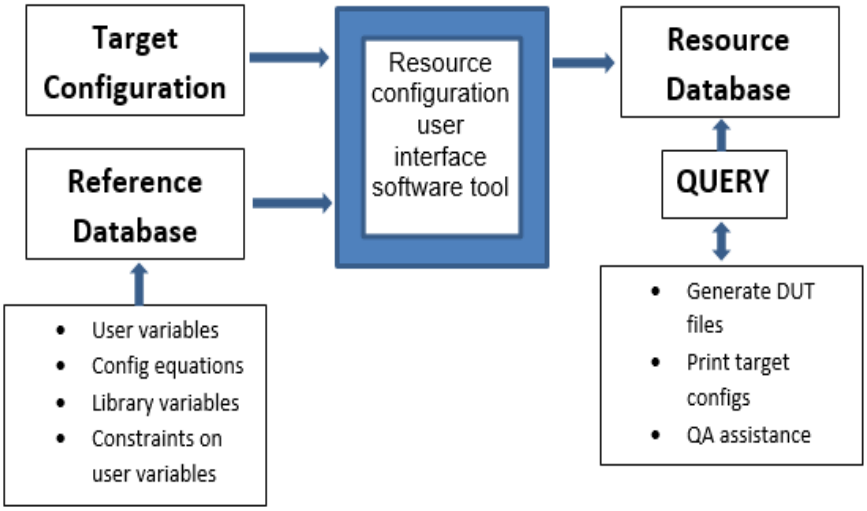


Figure 3.3: Resource configuration user interface software tool flow

3.5.2 Directly mapped parameters

Some of the architecture configuration parameters that are to be included in the RIS tool yaml file can be directly mapped to the parameters in target configuration file with just the names of both parameters being different. Some examples of such directly mapped parameters are shown in figure 3.4.

For these parameters we need to query the database of the tool for the values of target config parameter and to generate the yaml file we need to map that value fetched by querying with the corresponding RIS tool parameter. Example: We query the database of the tool to find value for cpu.has_el3 which comes out to be TRUE. Then we need to assign this value that is TRUE to cpu[0].exceptLevel.MonitorPrivLevel.present that is the name of RIS tool parameter. And print cpu[0].exceptLevel.MonitorPrivLevel.present = TRUE in tool acceptable format and order. This mappings file is in CSV (Comma Separated Value) format. The tool uses CSV parser to read this mappings file given as command line input.

RIS Tool Parameters	Target_Config_Parameters
cpu[0].excepLevel.MonitorPrivLevel.present	cpu.has_el3
cpu[0].excepLevel.MonitorPrivLevel.rw32Support	cpu.has_el3_32
cpu[0].excepLevel.MonitorPrivLevel.rw64Support	cpu.has_el3_64
cpu[0].excepLevel.HypervisorPrivLevel.present	cpu.has_el2

Figure 3.4: Directly mapped parameters example

3.5.3 Derived parameters

```
uint32_t Mask_12_3  = 0x1FF8;
uint32_t Mask_27_13 = 0xFFFE000;

uint32_t sys.l6_icache_sets     = ((cpu0.L6_ICACHE_CCSIDR_RESET) & (Mask_27_13)) >> 13 + 1;
sys.l6_icache_sets.attribute    = {(usr_var, 0), (raven_var, 1), (alias, "cpu.cache.cacheLevel.l6Cache.IcacheSet")};

uint32_t sys.l6_icache_ways     = ((cpu0.L6_ICACHE_CCSIDR_RESET) & (Mask_12_3)) >> 3 + 1 ;
sys.l6_icache_ways.attribute    = {(usr_var, 0), (raven_var, 1), (alias, "cpu.cache.cacheLevel.l6Cache.IcacheWays")};

uint32_t sys.l6_icache_linelen  = ((cpu0.L6_ICACHE_CCSIDR_RESET & 0x7) + 4);
sys.l6_icache_linelen.attribute = {(usr_var, 0), (raven_var, 1), (lib_name, RAVEN)};

uint32_t sys.l6_icache_size     = ((sys.l6_icache_sets) * (sys.l6_icache_ways) * (1 << sys.l6_icache_linelen)) ;
sys.l6_icache_size.attribute    = {(usr_var, 0), (raven_var, 1), (alias, "cpu.cache.cacheLevel.l6Cache.IcacheSize")};
```

Figure 3.5: Example Equations

Some of the architecture configuration parameters that are to be included in the RIS tool yaml file can be derived from parameters in target configuration file with the help of some equations as shown in figure 3.5. These equations needs to be integrated in the Resource configuration user interface software tool. The example of some equations is as above. In this example icache sets, ways ,line length and cache size is derived from the value of the CCSIDR Register (Current Cache Size ID Register).

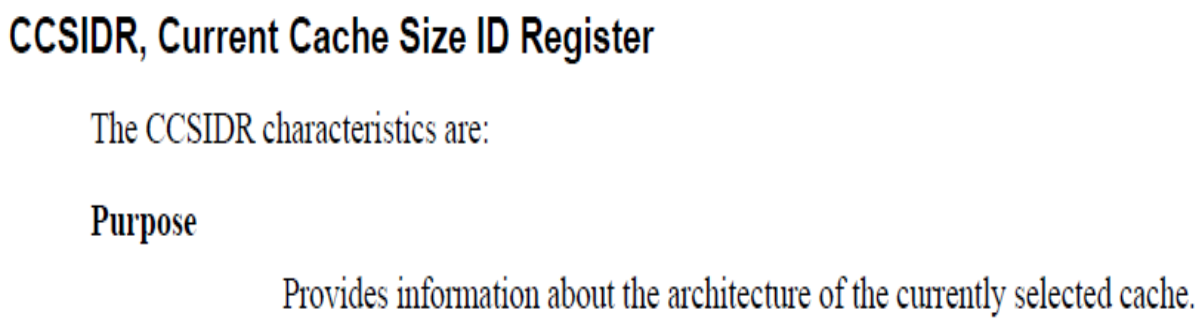


Figure 3.6: CCSIDR Register [7]

Different bit fields of the CCSIDR Register give us information about different features of cache. The detailed description of bit fields is as shown in figure 3.7. This description is available in Architectural Reference Manual [7]. And by studying it various equations can be derived for derived parameters as stated above.

Here the bit field [2:0] of the CCSIDR Register gives us information of the cache linesize. So if we have the information of the CCSIDR register and if we want to calculate linesize we need to follow reverse process. For example if the value of CCSIDR [2:0] is 1 then we need to write and equation that would add 4 to the value of bit field from bit 2 to 0 and then take an antilog with binary as base that is 2 thus the result obtained is nothing but the number of bytes in a cache line that is the linesize of a cache.

Similarly number of sets is given by bitfield [27:13] and associativity that is bitfields [12:3] is the number of ways cache lines can be arranged into. From all these parameters we can also find the cache size as it is the multiplication of cache linesize, cache sets and also cache ways.

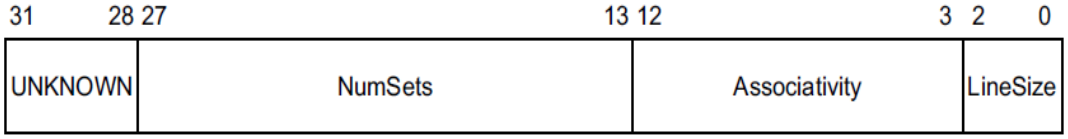
3.5.4 Structure of Unified Target Configuration Tool

This Unified Target Configuration generation tool is a python based tool for the auto generation of architecture configuration yaml file in RIS tool acceptable format. This tool takes the mappings file, target configuration file and the an optional name of output yaml file. Then it parses these three command line arguments provided while executing the python script using argparser module. Now these command line arguments are processed. The mappings file which describes the directly mapped parameter is given in CSV format so it is parsed using CSV parser module.

The target configuration file is used to initialize resource configuration user

Field descriptions

The CCSIDR bit assignments are:



UNKNOWN, bits [31:28]

Reserved, UNKNOWN.

NumSets, bits [27:13]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.

For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Figure 3.7: CCSIDR Register field description [7]

interface tool. While initialization the tool is built and the values of target configuration parameters is stored into the tool database and also the derived parameters are calculated with the help of equation integrated into the software. Now this database is being queried to get the values of the required parameters and finally an output yaml file is generated in RIS tool acceptable format. The block diagram of tool structure for the unified configuration generation tool is shown in figure 3.8. Some debug messages are also printed depending on the verbosity level that we have given to the resource configuration user interface tool.

An extra added feature is that we can include the equations for the derived parameters in a file and pass these equations to the tool. Thus this feature enables to directly calculate the derived parameters from the equations without the need for explicitly re-building the resource configuration user interface tool. Thus this tool

provides a functionality to directly add the equations to the reference database of the resource configuration user interface tool.

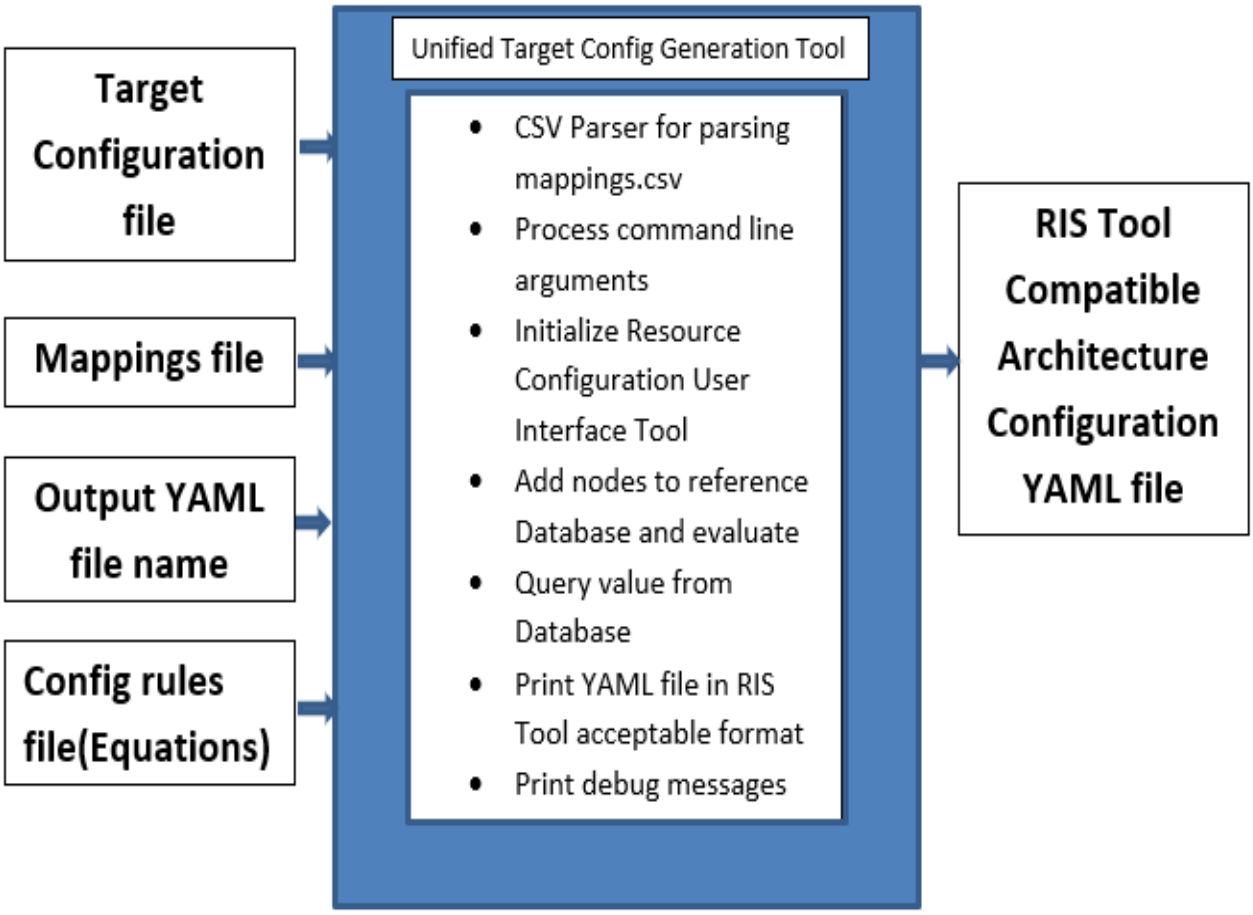


Figure 3.8: Flow of tool for RIS tool input architecture config YAML file generation

Thus with the help of this tool we get a single automated application which generates RIS tool acceptable input configuration file and that too without changing present format. But since we know that the RIS tool specific YAML file and the target configuration file cannot be mapped fully with each other thus this tool can auto generate the RIS tool specific YAML file only upto 90 to 95 percentage and the remaining RIS tool specific details such as its memory map parameter info is required to be added explicitly.

Chapter 4

MP RIS TOOL

4.1 Introduction

An SOC (System On Chip) can contain multiple clusters of processors and verification of such multi core environments is a big challenge, as the bugs are difficult to find and are often found late in the design cycle.There are different approaches to multicore verification and one of them is a Random Instruction Stream (RIS) generator tool. Here we would be talking about Arms own MP RIS tool developed by engineers of Architecture & Technology Group which uses a testing approach to target memory subsystem components critical to multicore functionality, such as Cache and Memory Hierarchy, cross-core coherency transactions, and the Load Store pipeline [10].

The verification of a processor having multiple core design is becoming more and more challenging due to the increasing complexity in the design of processor, weakly ordered memory, bus protocols, and presence of multiple levels i.e. multiple hierarchies of caches in memory subsystem. Arm architecture also defines some attributes and characteristics which are required to support devices and memory in system memory map which would further increase the complexity. Making the memory consistent and coherent is a key issue which needs to be addressed as different processors access same shared memory locations. As an example a unified or shared cache and update a single shared address space.The possibility of incoherence arises if there are multiple actors with access to caches and memory. Thus there are chances that they might access same address location. In modern systems, these actors are DMA engines, processor cores, and external devices which are able to read and/or write to caches and memory [11]. Thus we need to maintain a system that is coherent by observing same value for any particular address.

The MP RIS tool generates the output in the form of native Arm executables. The source file obtained from the generator is compiled using Arm assembly tool-chain. Then these test sequences generated are intended to run in a top-level

simulation based environment. The MP RIS generator provides the functionality to the user to control many aspects of the test which includes: Deciding upon the number, location and also the size of address spaces. It also allows the mix of instructions that will operate on the defined memory regions. Now these memory regions are also configurable so that the same regions can be shared between the Processing Elements (PEs). To allow such sharing we need to define address constraints to trigger various sharing scenarios and invoke snoops and cache-line migrations. The sharing scenarios are predefined in the form of assembly sequences and they are divided into various zones [9].

4.2 Features of MP RIS Tool

MP RIS tool focuses on memory sub-system operations and cross-PE coherency transactions in Multi-Processor/Cluster systems. It is a static RIS generator and known for achieving higher generation rate i.e. greater than 1000 Instructions Per Second (IPS).It offers full support of ARM v8-A AArch64 execution state. The AArch32 A32 (ARM) ISA is partially supported, and no support is available for AArch32 T32 (Thumb) ISA. Instruction recipes i.e. the test cases targeting to test some particular intent or some particular feature of the processor design are predefined and the logic is embedded into the tool. Thus we can functionally verify specific operations and micro-architectural features [10].

The features targeted by the MP RIS Tool are:

- Multi-processor memory coherency.
- Barriers.
- Cache and TLB maintenance operations.
- Message passing (Exclusive operations, Load Acquire or Store Release, Atomics).
- Load-store dependencies and hazards.
- Generate traffic to maximize use of load-store pipeline and evictions.

4.3 Test Topology

The threads executed by PEs are partitioned into equal number of zones. The transition between two zones is called global synchronization point as shown in figure 4.1. A zone or iteration specifies the number of 'independent' test sections

within the test case. Within each test section there is register initialization, pointer register setup, data tables and comparison tables [9]. In each such test section i.e. inside each zones there are scenarios. Scenarios are nothing but threads associated with memory dependencies. So inside each scenarios there is memory dependencies between different PEs.

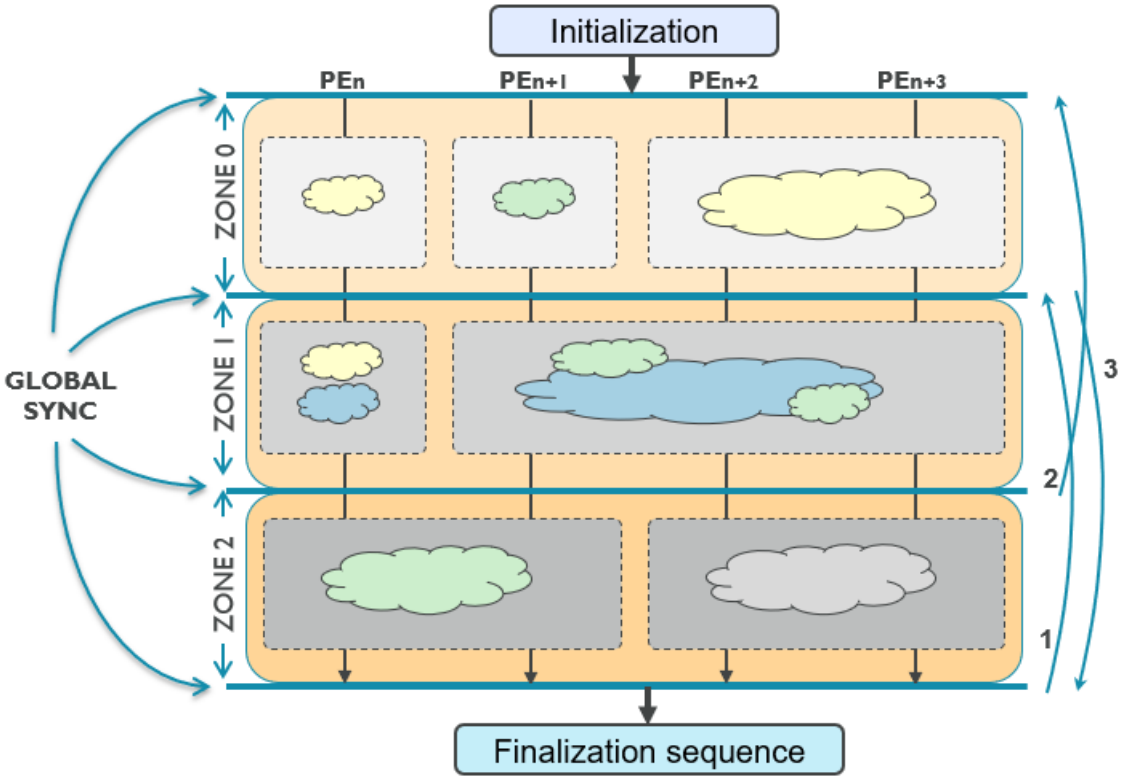


Figure 4.1: RIS tool test topology

. The MP RIS tool provides functionality to configure memory share percentage between PEs in different scenarios. All PEs are synchronized in time both at the beginning and also at the end of each zone. The initialization sequence is used to set up the test environment, and the finalization sequence simply does the clean up task.The intent of MP RIS Tool is mainly to generate stimulus that creates interesting bus traffic between PEs within a memory subsystem. The sequence of events for test generation is:

- Initialization sets up the test environment.
- Threads execute in each PE.
- All threads synchronize globally.

- Partition the test flow into Zones.
- Zone order defines the initial test sequence.
- Scenarios include one or more PEs. Scenarios define memory dependencies between PEs.
- Re-execution of Zones in a random order.
- Finalization sequence performs clean up.

There is an optional (configurable) re-execution of the zone, or even the entire testcase. This re-execution helps in stressing the timing relations between the cores and caches that are cold in the first pass, and warm in the second pass.

The generator itself is written in C++, accompanied by a host of libraries in Python, Arm Assembly, and XML utils. The ouput of the tool is a testcase in Arm assembly format. It also produces associated files required to assemble and link the testcase into an Executable and Linkable Format (ELF) file. Modern RTL simulators can consume these ELF files as a memory-resident image during simulation.

4.4 Design Configuration

The target designs configuration is the first input to the tool, in the form of files in a user-defined configuration directory (config-dir). They have the files as in figure 4.2 and figure 4.3:

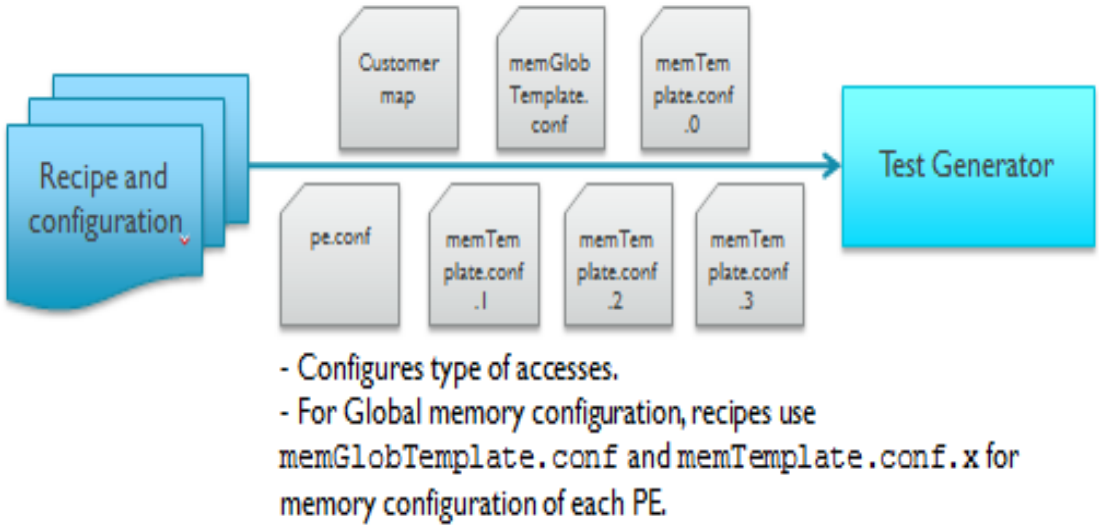


Figure 4.2: Design configuration files

- **Customer Map:**
  The target designs physical memory map, with regions dedicated for I/O, peripherals, ROM, Interrupt Controller etc. The available regions excluding the above regions are where the MP RIS Tool places its test data pages.

- **pe.conf:**
  Specifies the system configuration of the target design. It has inputs for number of clusters, number of PEs in each cluster, L1/L2/L3 cache parameters, feature enables, architecture etc.

- **Overrides:**
  Override files for recipe knobs.

4.5 RIS Tool Recipes

The tools inputs are configuration files, and the Armv8 ISA specified in XML format. The configuration files are organized as recipes that define the intent of the test. A recipe specifies the design space that the test targets, and provides maximum possible coverage in that space through knob randomization. Different generation runs of the recipe yield test cases that take in different combinations of the same knobs, that provide uniqueness in their own ways, and stress the same piece of logic

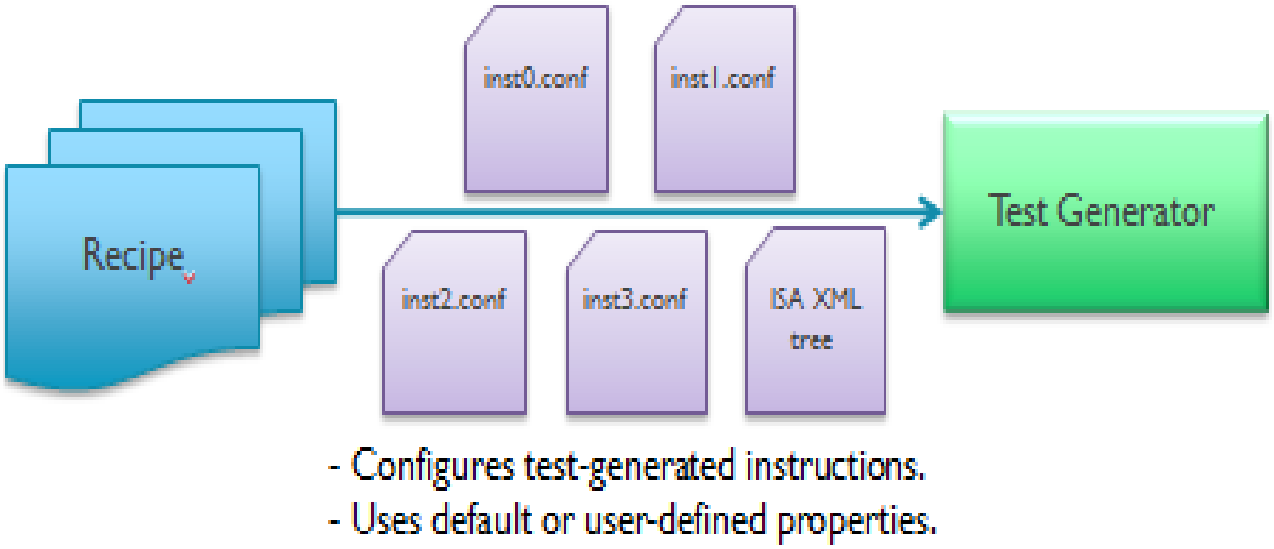


Figure 4.3: Recipe design configuration

from ever-so-slightly different angles. A recipe includes instruction configuration files and memory configurations for every core and global memory configuration file that can be used to tune the instruction stream. The memory configuration files specify bias for different Armv8 memory attributes, cache indexing, page crossing, cacheline crossing etc.

The tool also provides an override capability for these files, so users can specify an overriding value for specific or a set of knobs, to be fixed for the entire regression while other knobs are randomized. Users can generate tests using recipe categories in accordance with the maturity of their RTL design. The MP RIS Tool offers several regression script options for a convenient test-generation process.

Recipes are organized into three categories bring-up, functional, and microarch-stress. A block level view is as shown in figure 4.4

4.5.1 Bring-up Recipes

These recipes focus on the initial bring-up of load and store instructions, the memory subsystem and basic data-side MP features. These are the first set of MP RIS Tool recipes the user will focus on with the generated tests ranging between 5,000 and 50,000 instructions depending on design maturity and available resources.

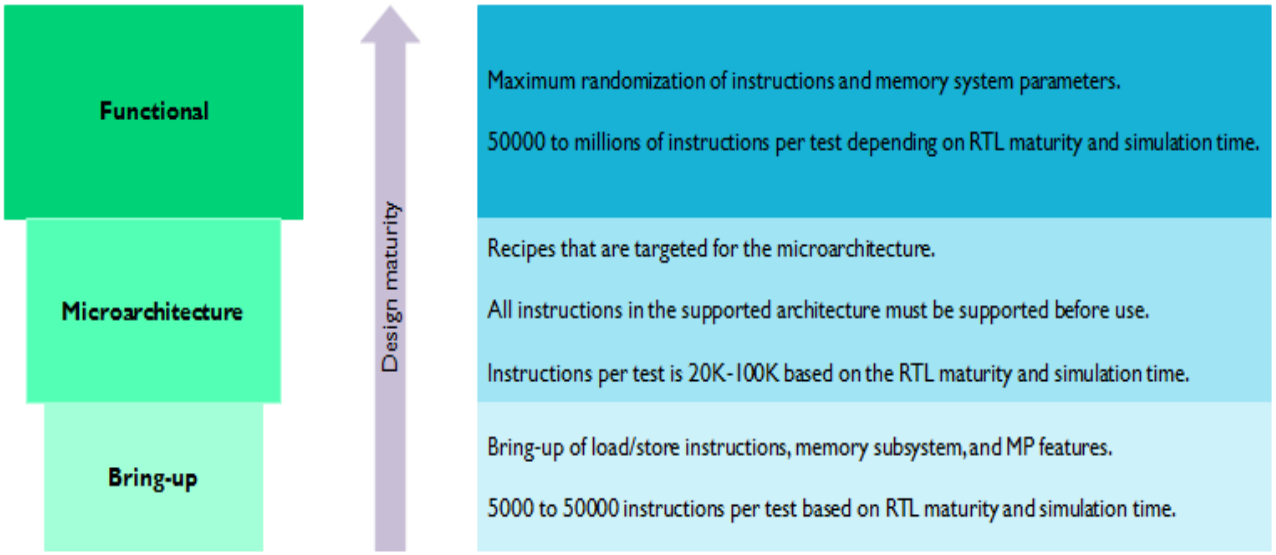


Figure 4.4: Recipe Types

4.5.2 Micro-architecture Recipes

These recipes are focused on distinct features in the architecture with the expectation that all available instructions and memory features are implemented and verified at unit level. Architectural features that are addressed include EL switching, exercising different features in the MMU, and interleaving various types of instructions so as to create hazards and address dependencies. These are the second set of recipes the user will focus on with the generated tests ranging between 20,000 and 100,000 instructions depending on design maturity and available resources.

4.5.3 Functional Recipes

Functional recipes are the final set of MP RIS Tool tests for the user to concentrate on since they are intended for mature designs. These recipes are expected to be used with tests ranging from around 50K to 1M plus instructions in length and focus on maximally randomizing the instructions and memory parameters.

4.6 Test Generation Process

The output from the generator is an assembly test file, and a scatter file that specifies addresses in the physical memory for ELF loading by the linker. The final

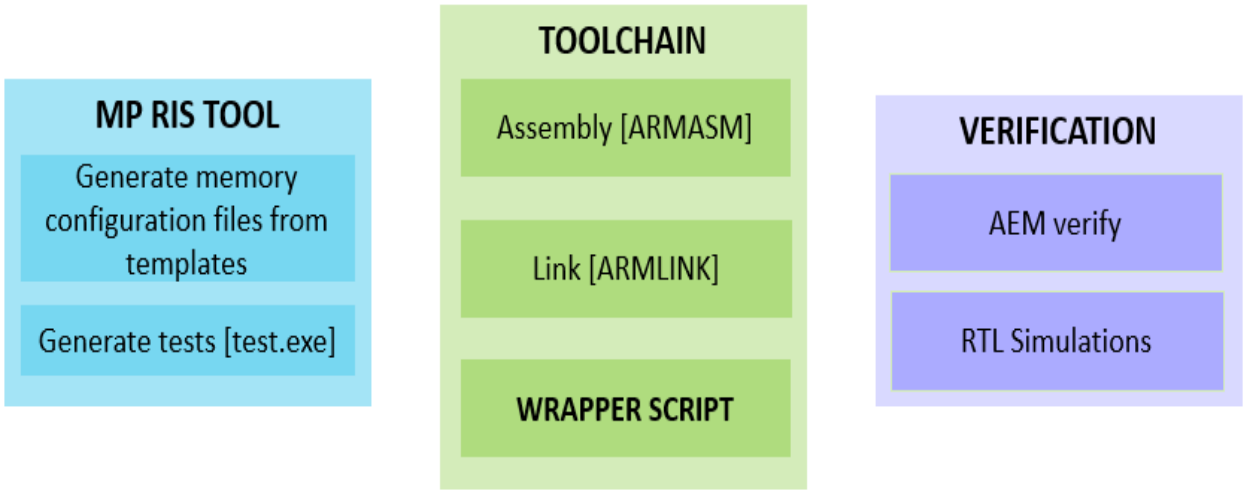


Figure 4.5: Test generation process

ELF is created after assembling and linking these files. The components involved are as shown in figure 4.5

The installation of the MP RIS Tool provides a tool binary (.exe) and a tool wrapper script. The binary generates the assembly and scatter files associated with a test. However, to generate an ELF file that can be run in simulation, an additional step is required to assemble and link the files output by the tool binary file. Thus, generating a test ELF file involves 2 steps:

1 Invoke MP RIS Tool binary to generate the tests assembly and scatter file.

2 Invoke the ARM assembler and linker to build an ELF test file.

Moreover, the resulting ELF test can be disassembled which becomes useful for debugging purposes. The wrapper script provided with the installation combines the steps above into a single operation. The detailed explanation of working of the wrapper script is provided in next section.

4.7 RIS Tool Wrapper

The MP RIS Tool wrapper is a shell script for the RIS/Verification tool. Its intent is to orchestrate the steps needed to configure, run and analyze a MP RIS

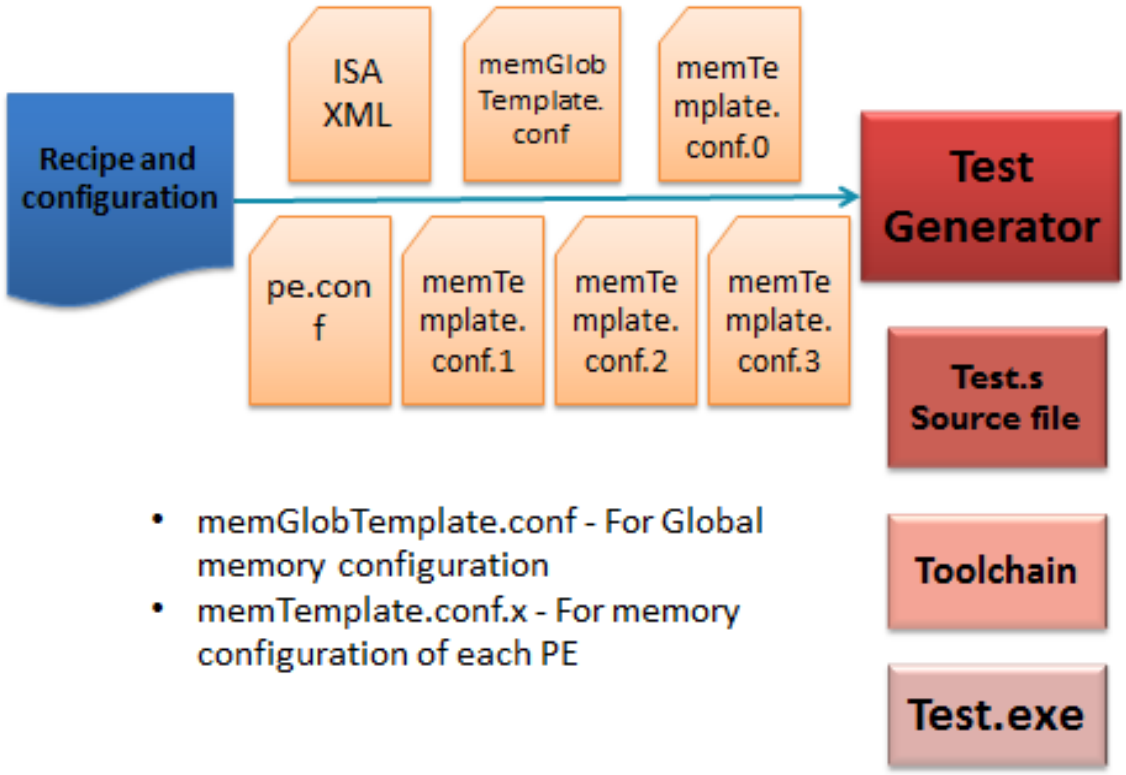


Figure 4.6: MP RIS tool Block Diagram

Tool regression from test generation to results validation.The wrapper allows a user to configure knobs that pertain to recipe values that influence test generation. A user can specify one set of knob values to one or more recipes or allow the MP RIS Tool wrapper to randomize knob values for them.The wrapper script provides a command-line interface that allows the user to control test generation by selecting recipes and parameters that define the intent, topology and size of the test.

The wrapper script manages the following:

- Environment setup
- Memory / Configuration setup
- Test Generation
- Assembly
- Linking

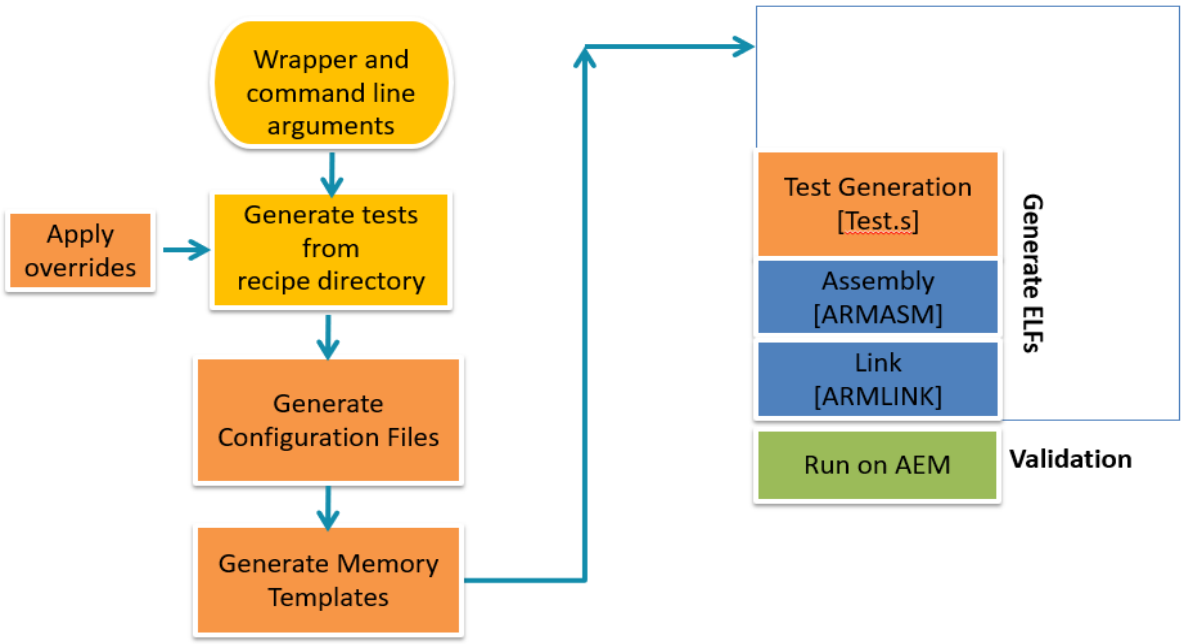


Figure 4.7: MP RIS tool Block Diagram

- AEM Verification
- Post test Validation - MP RIS Tool QA

The MP RIS Tool wrapper can override a design configuration. For that we need to specify a path to configuration directory that contains required files. The wrapper will use the values specified in these files in the place of the default values. The configuration override directories are required to have the following files:

- customer_map
- glob_override.conf
- memGlob_override.conf
- pe.conf

Thus this wrapper script is used to invoke all the procedures in a proper predefined sequence for generating the test case. Ablock level view of all the components involved is shown in figure 4.6. It reads the recipe directory for basic configuration of knobs. These knobs are overwritten if the same knobs are present in the override.conf file. Now the final list of knobs and their values is overwritten by the config files. Then by using these knobs and assigning some particular value to these

knobs the MP RIS Tool generates a test case containing sequence of instructions. These generated test cases are in the form of a source file i.e. .s file which needs to be assembled and linked further to generate an ELF file and this is done using Arm assembly and linking tools. Now this generated ELF file is used to verify the AEM model and also the RTL design of the processor as also explained by figure 4.7. Thus this MP RIS Tool proves to be efficient to cover the corner cases for a multicore environment. This tool also requires some Quality Assurance mechanism to keep a check on the quality of the generated test cases. The already available QA tool and the newly developed plugins for QA are discussed in upcomming chapters.

Chapter 5

PREGENERATION QA - RECIPE QA

5.1 Introduction

It is a tool that performs trace-based QA checks on MP tests from the MP RIS generator and measures test intent to help to tune generation. It also points to test issues that could potentially cause the Device Validation teams to pursue debug of a false failure. Thus as a result it increases quality of MP RIS tests used for validation and in turn decreases the probability of post-silicon bugs.

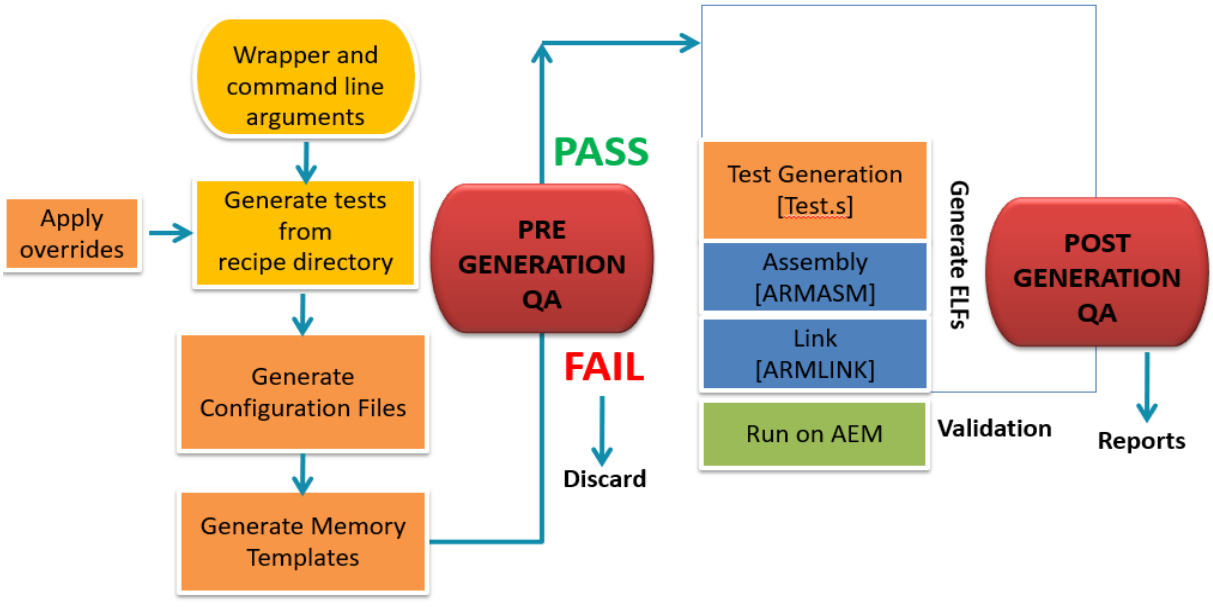


Figure 5.1: MP RIS Tool temp directory structure

The Pre-generation tool script is integrated with the wrapper script and is

called by the wrapper script after it generates the final set of knobs and corresponding values after applying all the overrides. Figure 5.1 shows where the pre-generation as well as the post generation tools are included in the execution cycle of the MP-RIS Tool. The pre-generation QA i.e. the recipe QA is divided into 3 phases as explained by figure 5.2:

- Checks for configs
- Checks for crosses
- Knob randomization report

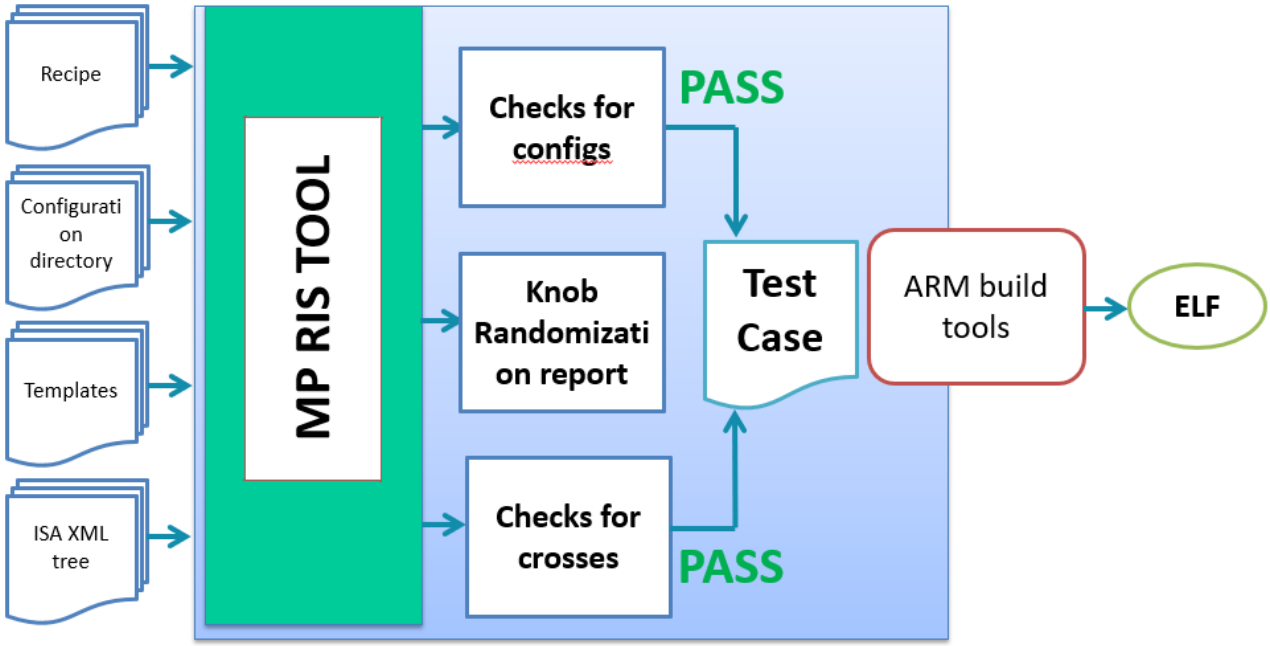


Figure 5.2: MP RIS Tool temp directory structure

Now the MP RIS tool takes various parameters from the users like the name of recipe, the config which is to be applied i.e. the path to config directory, test count etc. And then the MP RIS Tool applies all these overrides and generates a final list of knobs and their corresponding value to generate tests. Thus for each test it creates a temporary directory and this test can belong to any recipe thus the MP RIS Tool put this test directory inside a recipe directory to which it belongs.

Inside every test directory we have the basic recipe config directory which gives the basic set of knobs for that particular recipe. Now this knobs are overwritten by the _override.conf files to give a final list of knobs and values. If the config directory

is specified by the user then the knobs and values present in the config directory are also considered for overwriting the final list of knobs. The final list of knob and values taken by the MP RIS tool is not dumped anywhere and it just does the overrides on the fly during run time.

```
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA]$ cd anvil_20171121235418/
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA/anvil_20171121235418]$ ls
batch_1.lsf              el_switch      evict_ildst          logs            nstore_stb   stb_stress   vipt_loadstore
datapload_stcmoirr       el_switch_el2  evict_scale          noevict_cmoirr  pagecross    uao          vipt_neon
dos_strex_ronce_sameidx  evict_cmoirr   interleaved_spr_seq  nondet_sharing  results.log  vipt_cmoirr
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA/anvil_20171121235418]$ cd uao/
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA/anvil_20171121235418/uao]$ ls
7_uao_v8_2_a64_np4_nz4_ni500_sh100_re1_2424180454_le_21112017  elfs
9_uao_v8_2_a64_np4_nz4_ni500_sh100_re1_1134313636_le_21112017  knob_randomization
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA/anvil_20171121235418/uao]$ cd 7_uao_v8_2_a64_np4_nz4_
ni500_sh100_re1_2424180454_le_21112017/
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA/anvil_20171121235418/uao/7_uao_v8_2_a64_np4_nz4_ni500
_sh100_re1_2424180454_le_21112017]$ ls
Product.Compiler65.0201707V2068558fe000057fe4ecd8515.liccache  customer_map          memGlob_override.conf
anvil.disasm                                                   glob_override.conf    mem_override.conf.0
anvil.elf                                                      instr_override.conf   mem_override.conf.1
anvil.exe                                                      lib_anvil_pgedit.a    mem_override.conf.2
anvil.o                                                        lib_anvil_pgt.a       mem_override.conf.3
anvil_aem.tarmac                                               lib_bootswc.a         pe.conf
anvil_asm.log                                                  lib_bootswc_common.a  pg.dat
anvil_gen.log                                                  lib_handlers.a        pg.h
anvil_link.log                                                 lib_intrhand.a        pg.hs
cloud_bias.conf                                                lib_swc.a             pg.s
confTemplate                                                   lib_uhandler.a        pg.scat
config_checks                                                  load_file_ns          recipe-config
crosses_checks                                                 load_file_s
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA/anvil_20171121235418/uao/7_uao_v8_2_a64_np4_nz4_ni500
_sh100_re1_2424180454_le_21112017]$ cd recipe-config/
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA/anvil_20171121235418/uao/7_uao_v8_2_a64_np4_nz4_ni500
_sh100_re1_2424180454_le_21112017/recipe-config]$ ls
7_uao_v8_2_a64_np4_nz4_ni500_sh100_re1_2424180454_le_21112017.addr  instr1.conf  instr_conf_bias_0  mem.conf.1
glob.conf                                                          instr2.cl    instr_conf_bias_1  mem.conf.2
instr0.cl                                                          instr2.conf  instr_conf_bias_2  mem.conf.3
instr0.conf                                                        instr3.cl    instr_conf_bias_3  memGlob.conf
instr1.cl                                                          instr3.conf  mem.conf.0
```

Figure 5.3: MP RIS Tool temp directory structure

The recipeQA tool first of all takes as an input the path of the temp directory that is created by the MP RIS Tool and it parses through all the files inside a directory to store all the recipes, their corresponding tests, config files and override files in a data structure. The directory structure is as shown in figure 5.3 and 5.4. Thus the tool after reading the temp directory have a data structure of multilevel list having dictionary as its last level. Thus value for knobs for each test of each recipes are stored. Then another data structure stores the override knob and values. now a final list of knob and values is created by updating the values of the original conf files with the values in override files. If the knob is present in both the conf file and override file then the final list will have the knob with the value equal to that of the override file.

Thus the recipe QA Tool have various modules for different task:

- Process command line arguments:

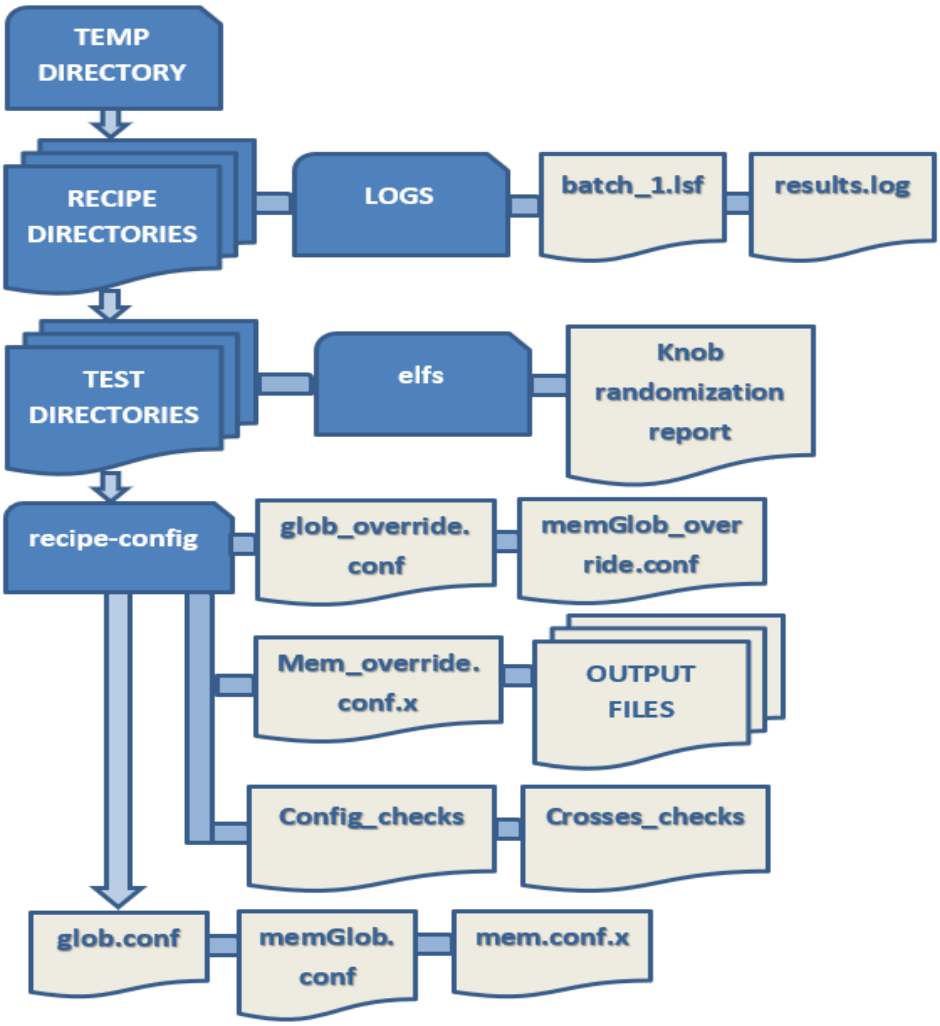


Figure 5.4: MP RIS tool temp directory structure block diagram

It processes the input temporary directory and the name of the config if checks for configs is on and also the levels for which we want to do the recipe QA. We can simultaneously do the QA for all three levels. The was this pre-generation recipeQA script can be executed is shown on figure 5.5.

- Fetch recipe directory:
  This function takes as an input the path to temporary directory generated by the MP RIS Tool and then stores the path to every recipe directory inside the temp directory and discared all other files and folders in temp directory.

- Fetch test directory:
  It goes to path of every recipe directory and then stores the path for test inside

every recipe directory.

- Fetch config files:
  This function takes the path to every test directory for every recipe and then stores path to the required config files that are useful like the mem_conf files, memGlob_conf and glob_conf files and this function ignores all the other files present in the test directory. This function also stores the override file paths, just we need to change the input path from test directory path to test_dit_path+recipe-conf as the override files are inside the recipe-conf directory.

- Write data-structure:
  The knobs and values inside the config files or the override files are populated inside a data structure using this function.

```
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA]$ python RecipeQA.py --help
Usage: recipe_QA [OPTION] -t/--test_dir <input temp_test_dir_path>
                OPTION
                    -h/--help        Print help message
                    -t/--test_dir    Temp directory path
                    --level1         checks for configs
                    --level2         checks for crosses
                    --level3         checks for knob randomization
                    -c/--config      Name of config to be analyzed (level1)
```

Figure 5.5: Recipe QA help

- Apply overrides:
  The data structure storing recipe config knobs and values are overwritten using the override data-structure to get the final list of knobs and their corresponding values.

- Make knob Randomization out file:
  This function makes a file to show the level of knob randomization achieved or every recipe. This file is placed inside every recipe directory of the temp directory i.e. parallel to the test directories.

- Parse config details input file:
  This function is used to read the config details input file and make list of valid configs.

- Perform checks for config:
  This function check the final data structure with the knobs and values inside the config directory to check if the final list of knobs is overwritten with the values of knobs in config directory or not. If the knob is not overwritten then

it flags a fail. This function also checks if the config specified as comand line input is from the list of valid configs or not. If it is not valid then the recipe QA Tool prints the list of valid configs.

- Make checks for config out file:
  This function prints the output file with all configs checks flagged as pass or fail and places the output config checks file inside every test directory since the checks for configs are performed for every tests.

- Process crosses details file:
  This function is used to parse the crosses details input file to make the list of valid crosses and also the list of required crosses.

- Perform checks for crosses:
  This function reads the values of knobs from the data structure and compares if any invalid combination of knobs are occurring in same test simultaneously. It also checks if the required crosses are not occurring and flags the fails and the pass.

- Generate checks for crosses out file:
  This function prints the output file with all checks for crosses flagged as pass or fail and places the output crosses checks file inside every test directory since the checks for invalid and required crosses are performed for every tests.

- Main function calling all functions. And after the output files are generated and placed in their respective locations a print message appears on the screen as shown in figure 5.6.

```
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA]$ python RecipeQA.py --test_
dir ./anvil_20171121235418/ --level1 --level2 --level3 -c prmfault_stg1
--------------------------------------------------------------------------------
Each recipe dir Contains: Knob Randomization report.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Each test dir Contains: Config checks report.
--------------------------------------------------------------------------------
```

Figure 5.6: Recipe QA execution

5.2 Recipe QA Execution

For the execution of the RecipeQA Tool we need to compulsorily provide a path to the temp directory and provide the levels for which we need to perform the

checks. If we are enabling checks for configs i.e. level1 then we also need to specify the name of config for which the checks for configs is to be done. If the config is invalid then it is flagged and the RecipeQA Tool prints the list of all valid configs as shown in figure 5.7.

After generating the knob randomization reports and placing them inside every recipe directory the RecipeQA Tool prints a status on command line saying: " Each recipe dir contains: Knob Randomization report." And then similarly it prints status as shown in figure 5.6 after generating the config checks and crosses checks report and placing them inside every test directory.

```
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/bakup/recipeQA]$ python RecipeQA.py --test_
dir ./anvil_20171121235418/ --level1 --level2 --level3
--------------------------------------------------------------------------------
Each recipe dir Contains: Knob Randomization report.
--------------------------------------------------------------------------------
Error:Can't proceed. Specify config to be analyzed.
The list of all valid configs is as below:
['alignment_check_disable', 'alignment_check_enable', 'mmuon_dcon_icon', 'pan', 'vhe_e2h1_tge0_el
0ns', 'vhe_e2h1_tge0_el2', 'vhe_e2h1_tge1', 'vipt_vhe', 'ras_error_region', 'ras_error_region_esb
', 'sync_bakers', 'sync_atomics', 'sync_acqrel', 'sync_exclusive', 'sync_exwfe', 'acc_fault', 'as
ize_fault', 'codemem_attr', 'evict', 'falsesharing_maxsh', 'linecross', 'minlines_addrdep100', 'n
ondetsharing_maxsh', 'pagecross', 'prmfault_stg1', 'prmfault_stg2', 'regdep_rar', 'regdep_raw', '
regdep_war', 'regdep_waw', 'sameview', 'semamem_attr', 'trans_fault', 'unalign', 'unalign_linecro
ss', 'vipt']
Usage: recipe_QA [OPTION] -t/--test_dir <input temp_test_dir_path>
                OPTION
                    -h/--help        Print help message
                    -t/--test_dir    Temp directory path
                    --level1         checks for configs
                    --level2         checks for crosses
                    --level3         checks for knob randomization
                    -c/--config      Name of config to be analyzed (level1)
```

Figure 5.7: Recipe QA list of configs

5.3 Knob Randomization Report

The MP RIS Tool does not dump a final list of knobs and values taken after applying all the override files. Thus with this dump of knob randomization report we get the final values of knobs taken by the MP RIS Tool. This report is generated for every recipe. Thus suppose we have 5 test corresponding to one recipe then the knob can have maximum 5 different values. Thus in knob randomization report the knob is printed once and that knob has 5 values separated by comma as shown in figure 5.8 for two test belonging to one recipe. Thus by analyzing this file we can

```
uao:
    glob.conf:
        NUM_INSTR_ZONE = ['1272 : 2000', '1637 : 2000']
        RAND_REG_REINIT_PERIOD = ['200', '200']
        SYNC_METHOD{ACQ_REL} = ['10', '10']
        SYNC_METHOD{ATOMIC_CASP} = ['10', '10']
        SYNC_METHOD{ATOMIC_CAS} = ['10', '10']
        SYNC_METHOD{ATOMIC_LDADD} = ['10', '10']
        SYNC_METHOD{ATOMIC_LDEOR} = ['10', '10']
        SYNC_METHOD{ATOMIC_LDSET} = ['10', '10']
        SYNC_METHOD{ATOMIC_LDSMAX} = ['10', '10']
        SYNC_METHOD{ATOMIC_LDSMIN} = ['0', '0']
        SYNC_METHOD{ATOMIC_LDUMAX} = ['10', '10']
        SYNC_METHOD{ATOMIC_LDUMIN} = ['0', '0']
        SYNC_METHOD{ATOMIC_SWP} = ['10', '10']
        SYNC_METHOD{BAKERS} = ['10', '10']
        SYNC_METHOD{LDST_EX_WFE} = ['10', '10']
        SYNC_METHOD{LDST_EX} = ['10', '10']
        SYNC_METHOD{LO_BAKERS} = ['10', '10']
        SYNC_METHOD{NON_RESET} = ['0', '0']
    memGlob.conf:
        ADDRESS_DEPENDENCY = ['50', '60']
        ATOMIC_ENABLE_DEV_NC = ['OFF', 'OFF']
        CODEREGIONDE = ['0', '0']
        CODEREGIONNC = ['0', '0']
        CODEREGIONWB = ['100', '100']
        CODEREGIONWT = ['0', '0']
        FALSE_SHARING = ['50', '20']
        GRAN_TG_16KB = ['100', '100']
        GRAN_TG_16KB_PGSZE_16K = ['100', '100']
        GRAN_TG_16KB_PGSZE_32M = ['100', '100']
        GRAN_TG_4KB = ['100', '100']
        GRAN_TG_4KB_PGSZE_16M = ['0', '0']
        GRAN_TG_4KB_PGSZE_1G = ['100', '100']
        GRAN_TG_4KB_PGSZE_1M = ['0', '0']
```

Figure 5.8: Output knob randomization report

see if the value the knob is taking is being randomized or not. Thus we can get an estimate of the level of randomization that is achieved.

The functions specific to generating knob randomization report in the Recipe QA Tool are:

- Write data-structure:
  The knobs and values inside the config files or the override files are populated inside a data structure using this function.

- Apply overrides:
  The data structure storing recipe config knobs and values are overwritten using the override data-structure to get the final list of knobs and their corresponding values.

- Make knob Randomization out file:
  This function makes a file to show the level of knob randomization achieved or every recipe. This file is placed inside every recipe directory of the temp directory i.e. parallel to the test directories.

5.4 Checks for crosses

```
#Invalid Crosses
PAGE_CROSSING,>,0 x LINE_CROSSING,=,OFF
PAGE_CROSSING,>,0 x ADDRESS_DEPENDENCY,>,0
PAGE_CROSSING,>,0 x CONTIGUOUS_BIT_SET,>,0
INDEX_DEPENDENCY,>,0 x CONTIGUOUS_BIT_SET,>,0
PAREUSEWITHINPAGE,>,0 x CONTIGUOUS_BIT_SET,>,0
PAGE_CROSSING,>,0 x L2EVICTEQU,=,ON
CONTIGUOUS_BIT_SET,>,0 x L2EVICTEQU,=,ON
WAYSCALE,>,0 x INDEX_DEPENDENCY,=,0
IDENTITYMAPPING,>,0 x VIPT,=,ON
VIPT,=,ON x IDENTITYMAPPING,>,0
SCENARIO_PAGE_STACK,>,0 x INDEX_DEPENDENCY,>,0
SCENARIO_PAGE_STACK,>,0 x PAGE_CROSSING,>,0
GRAN_TG_64KB,>,0 x VIPT,=,ON
STAGE2_MMUON,>,0 x VMID_ENABLE!,=,ON
UNALIGNED,=,0 x UNALIGNED_LIMIT,>,0
CNP,>,0 x VIEW_CONSTRAINT!,=,MIN
SECURE,=,TRUE x EL,=,EL3NS or EL2 or EL1NS or EL0NS
SECURE,=,FALSE x EL,=,EL3S or EL1S or EL0S
INDEX_DEPENDENCY,>,0 x ADDRESS_DEPENDENCY,>,0
#Required Crosses
L2EVICTEQU,=,ON & INDEX_DEPENDENCY,>,0
WAYSCALE,>,0 & INDEX_DEPENDENCY,>,0
L2HARDCODE,=,ON & L2EVICTEQU,=,ON
VMID_ENABLE,=,ON & VMID_COUNT,>,0
VIPT,=,ON & INDEX_DEPENDENCY,>,0
```

Figure 5.9: Crosses_details file snapshot

```
Recipe: pagecross
Test: 11_pagecross_v8_2_a64_np4_nz4_ni500_sh100_re1_687648052_le_21112017
++++++++++++++++++++++++++++++++++++++++
SCENARIO_PAGE_STACK,>,0 x PAGE_CROSSING,>,0 INVALID CROSS FAIL
SCENARIO_PAGE_STACK:10 x PAGE_CROSSING:100
**********
```

Figure 5.10: Output checks for crosses file

The checks for crosses is done to ensure that no such cross appears that would hamper the intent of the test. It also makes sure that in randomization scenario no invalid combination occurs simultaneously in a test case. Thus there is check for invalid crosses. Also there is a check for required crosses. In the check for required crosses if one knob has some particular value then there should be a required cross knob present in the same test with a fixed value. Thus this checks for crosses plays an important role by checking the combination of knob that the test case that is going to be generated from a list of knob and values would be a valid test case or not. If the test is invalid i.e. it contain an invalid cross or does not contain a require cross then the test case is discarded and not generated. Thus this saves the MP RIS Tool time and resources that would have been wasted in generating invalid test case that would be flagged after running this test case on either AEM or on the RTL simulation model.

The RecipeQA Tool take the crosses details file as shown in figure 5.9 as an input file to lay the rules of invalid crosses and required crosses and then with the help of these rules generate the checks for crosses output file as in figure 5.10.

The functions specific to checks for crosses in the Recipe QA Tool are:

- Write data-structure:
  The knobs and values inside the config files or the override files are populated inside a data structure using this function.

- Apply overrides:
  The data structure storing recipe config knobs and values are overwritten using the override data-structure to get the final list of knobs and their corresponding values.

- Process crosses details file:
  This function is used to parse the crosses details input file to make the list of valid crosses and also the list of required crosses.

- Perform checks for crosses:
  This function reads the values of knobs from the data structure and compares if any invalid combination of knobs are occurring in same test simultaneously.

It also checks if the required crosses are not occurring and flags the fails and the pass.

- Generate checks for crosses out file:
  This function prints the output file with all checks for crosses flagged as pass or fail and places the output crosses checks file inside every test directory since the checks for invalid and required crosses are performed for every tests.

5.5 Checks for configs

```
**************************************

mmuon_dcon_icon:

**************************************
==> mem_override.conf.0 <==
MMUON_DCON_ICON = 7

==> mem_override.conf.1 <==
MMUON_DCON_ICON = 7

==> mem_override.conf.2 <==
MMUON_DCON_ICON = 7

==> mem_override.conf.3 <==
MMUON_DCON_ICON = 7
**************************************

pan:

**************************************
==> memGlob_override.conf <==
PAN = 1

==> mem_override.conf.0 <==
E2H = 0
TGE = 0
EL = EL0NS

==> mem_override.conf.1 <==
E2H = 0
TGE = 0
EL = EL2

==> mem_override.conf.2 <==
E2H = 0
TGE = 0
EL = EL0NS

==> mem_override.conf.3 <==
E2H = 0
TGE = 0
EL = EL0NS
**************************************
```

Figure 5.11: Config_details file snapshot

The checks for configs are required to make it double sure that the configs are being overwritten properly in the final list of knob:values or not. Thus if the values of knobs is changed in the config directory or some new knob is added in the config directory file then the same would not be reflected in the config details file as the config details file would have the old details unless it is modified. Thus this new changes would be flagged. Thus for a test to pass the checks for configs we need to include the new knob or change the value of knob in the config details file as well. This prevents the user from accidentally modifying the values of the config file. This tool is also used to flag a fail when the user is running the MP RIS Tool with some other config. For example when the user is trying to generate MP RIS Tests with configs specific to device validation suites then it would flag a fail specifying the required cinfig settings for the MP RIS Tool.

The functions specific to checks for crosses in the Recipe QA Tool are:

- Write data-structure:
  The knobs and values inside the config files or the override files are populated inside a data structure using this function.

- Apply overrides:
  The data structure storing recipe config knobs and values are overwritten using the override data-structure to get the final list of knobs and their corresponding values.

- Parse config details input file:
  This function is used to read the config details input file and make list of valid configs.

- Perform checks for config:
  This function check the final data structure with the knobs and values inside the config directory to check if the final list of knobs is overwritten with the values of knobs in config directory or not. If the knob is not overwritten then it flags a fail. This function also checks if the config specified as comand line input is from the list of valid configs or not. If it is not valid then the recipe QA Tool prints the list of valid configs.

- Make checks for config out file:
  This function prints the output file with all configs checks flagged as pass or fail and places the output config checks file inside every test directory since the checks for configs are performed for every tests.

The RecipeQA Tool take the config details file as shown in figure 5.11 as an input file to lay the rules for the list of knobs that must be present and if present then to lay the rules on their values. And then with the help of these rules file i.e. the config details file the RecipeQA Tool generates the checks for configs output file simillar to the one shown in figure 5.12.

```
Recipe: pagecross
Test: 11_pagecross_v8_2_a64_np4_nz4_ni500_sh100_re1_687648052_le_21112017
++++++++++++++++++++++++++++++++++++++++
File: memGlob.conf
    CODEREGIONDE:0 PASSED OK
    CODEREGIONNC:0 PASSED OK
    CODEREGIONWB:100 PASSED OK
    CODEREGIONWT:0 PASSED OK
    MINPAGESETATTR:WB_WB PASSED OK
    MINPAGESETDEPENDENCY:50 PASSED OK
    SEMAREGIONDE:0 PASSED OK
    SEMAREGIONNC:0 PASSED OK
    SEMAREGIONWB:100 PASSED OK
    SEMAREGIONWT:0 PASSED OK
File: mem.conf.0
    ACCESS_FLAG_FAULT:0 PASSED OK
    ADDRESS_SIZE_FAULT:0 PASSED OK
    STAGE1KERNRO:5 FAIL
    STAGE1KERNRW:5 FAIL
    STAGE1USRRO:5 FAIL
    STAGE1USRRW:90 FAIL
    TRANSLATION_FAULT:0 PASSED OK
File: mem.conf.1
    ACCESS_FLAG_FAULT:0 PASSED OK
    ADDRESS_SIZE_FAULT:0 PASSED OK
    STAGE1KERNRO:5 FAIL
    STAGE1KERNRW:5 FAIL
    STAGE1USRRO:5 FAIL
    STAGE1USRRW:90 FAIL
    TRANSLATION_FAULT:0 PASSED OK
File: mem.conf.2
    ACCESS_FLAG_FAULT:0 PASSED OK
    ADDRESS_SIZE_FAULT:0 PASSED OK
    STAGE1KERNRO:5 FAIL
    STAGE1KERNRW:5 FAIL
    STAGE1USRRO:5 FAIL
    STAGE1USRRW:90 FAIL
    TRANSLATION_FAULT:0 PASSED OK
File: mem.conf.3
    ACCESS_FLAG_FAULT:0 PASSED OK
    ADDRESS_SIZE_FAULT:0 PASSED OK
    STAGE1KERNRO:5 FAIL
    STAGE1KERNRW:5 FAIL
    STAGE1USRRO:5 FAIL
    STAGE1USRRW:90 FAIL
    TRANSLATION_FAULT:0 PASSED OK
++++++++++++++++++++++++++++++++++++++++
Total Pass: 22
Total Fail: 16
RecipeQA Status: FAILED
```

Figure 5.12: Output checks for configs file

Chapter 6

POSTGENERATION QA - TEST LENGTH ANALYSIS

6.1 Introduction

Post-generation QA is a test length analysis tool that performs trace based QA checks on generated tests from the MP-RIS generator. It measures test length by reading tarmac trace to help debug generated test cases. It calculates the number of instructions anticipated and compares with actually generated instructions. And based on the results of comparison gives a fail if not equal as shown in figure 6.1. This tool makes it very easy to debug tarmac which was being done manually otherwise.

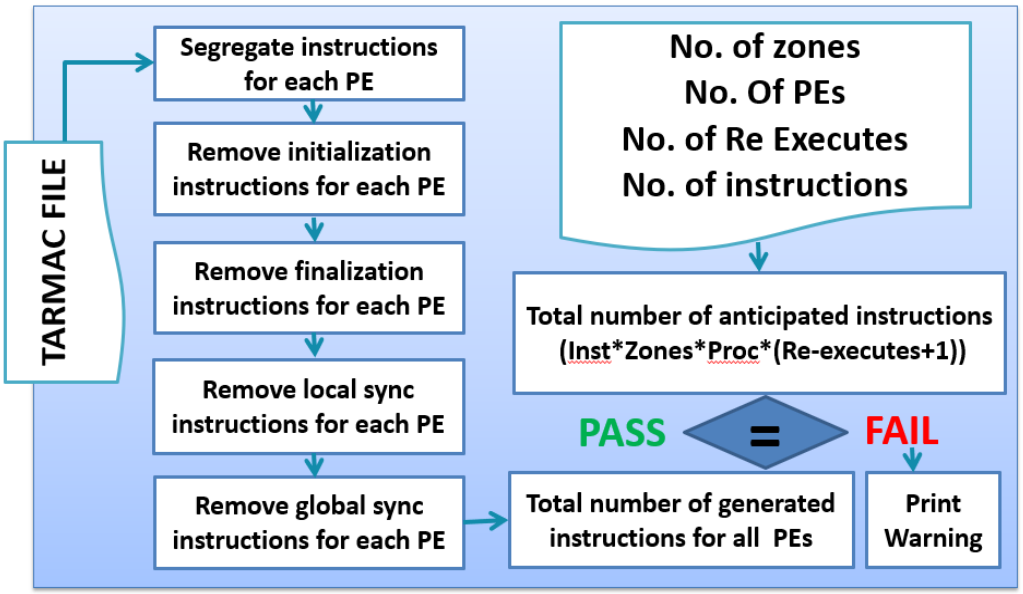


Figure 6.1: Post-generation QA Tool block diagram

```
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/anvil/anvil_20180320020358/idp]$ python ver
ify_test_length.py --help
Usage: verify_test_length.py [OPTION] -t/--test_dir <input test_dir_path>
               OPTION
                   -h/--help       Print help message
                   -t/--test_dir   Test directory path
```

Figure 6.2: Help option of tool to verify test length

6.2 Working

This tool takes as an input the path to the test directory as shown in figure 6.2. Then it would parse the files present in the test directory to read the aem_tarmac file and qa.logs file. The main intent of the development of this tool is to analyze the tarmac file to calculate the overhead instructions that are added. Thus based on the analysis the instruction count should be such that the useful number of instructions generated are more than the overhead instructions added. Thus this would increase the generation efficiency of the MP RIS tool.

```
verify_test_length.py
[khusun01@login4 /arm/projectscratch/pd/metl/khusun01/anvil/anvil_20180320020358/idp]$ python ver
ify_test_length.py --test_dir 11_idp_v8_2_a64_np2_nz1_ni10_sh100_re0_3597187278_le_20032018/
-------------------------------------------------------
Instructions expected in the test (Inst*Zones*Proc*(Re-executes+1)) are: 20
anvil_qa.log: Insts of Interest: 23
Total number of instructions generated by the test are: 1376
Number of instructions generated by the test (between ORR X11,X11,X11s and ORR X12,X12,X12s ) are
: 21
Number of instructions in the test (Removing init, finalization, local syncs and global syncs) ar
e: 261
-------------------------------------------------------
```

Figure 6.3: Output of tool to verify test length

The output of this tool gives as shown in figure 6.3 us the number of instructions that are used for initialization, finalization, the local synchronization between processing elements and the global synchronization between zones. Thus all these instructions are overhead added to generate some useful instructions. Thus for 50%

generation efficiency the number of instructions must be atleast double than the overhead instructions added. The AEM Tarmac shown in figure 6.4 is the output file obtained after running the test which is analyzed to calculate the overhead added. This tarmac file gives the information about the cpu cycles and the instructions that are executed in every cycle. Thus the post generation QA tool counts the overhead instructions by reading this file.

```
660 clk cpu1 IT (352) 0000000d80000f74 d61f0020 0 EL0t_n : BR       x1
661 clk cpu1 IT (353) 0000000d80001258 aa0b016b 0 EL0t_n : ORR      x11,x11,x11
661 clk cpu1 R X11 78566EC272156908
662 clk cpu1 IT (354) 0000000d8000125c 1b0bd860 0 EL0t_n : MSUB     w0,w3,w11,w22
662 clk cpu1 R X0 0000000010F0ACDD
663 clk cpu1 IT (355) 0000000d80001260 8a202049 0 EL0t_n : BIC      x9,x2,x0,LSL #8
663 clk cpu1 R X9 0000000D00000068
664 clk cpu1 IT (356) 0000000d80001264 1a000055 0 EL0t_n : ADC      w21,w2,w0
664 clk cpu1 R X21 0000000090F0BD46
665 clk cpu1 IT (357) 0000000d80001268 138b02d1 0 EL0t_n : EXTR     w17,w22,w11,#0
665 clk cpu1 R X17 0000000072156908
666 clk cpu1 IT (358) 0000000d8000126c f12867db 0 EL0t_n : SUBS     x27,x30,#0xa19
666 clk cpu1 R cpsr 20000000
666 clk cpu1 R X27 0000000D8000030B
667 clk cpu1 IT (359) 0000000d80001270 f2df9a6a 0 EL0t_n : MOVK     x10,#0xfcd3,LSL #32
667 clk cpu1 R X10 ACDDFCD32AD210F0
668 clk cpu1 IT (360) 0000000d80001274 dac016c5 0 EL0t_n : CLS      x5,x22
668 clk cpu1 R X5 0000000000000000
669 clk cpu1 IT (361) 0000000d80001278 138402b4 0 EL0t_n : EXTR     w20,w21,w4,#0
669 clk cpu1 R X20 000000005A421E15
670 clk cpu1 IT (362) 0000000d8000127c 9ac20d4a 0 EL0t_n : SDIV     x10,x10,x2
670 clk cpu1 R X10 FFFFFFFFF9D78E05
671 clk cpu1 IT (363) 0000000d80001280 5ac015a2 0 EL0t_n : CLS      w2,w13
671 clk cpu1 R X2 0000000000000001
672 clk cpu1 IT (364) 0000000d80001284 d2d65919 0 EL0t_n : MOV      x25,#0xb2c800000000
672 clk cpu1 R X25 0000B2C800000000
673 clk cpu1 IT (365) 0000000d80001288 aa0c018c 0 EL0t_n : ORR      x12,x12,x12
673 clk cpu1 R X12 B3761390AB4843C2
674 clk cpu1 IT (366) 0000000d8000128c aa0d01ad 0 EL0t_n : ORR      x13,x13,x13
674 clk cpu1 R X13 3C2B3761390AB484
675 clk cpu1 IT (367) 0000000d80001290 d53bd043 0 EL0t_n : MRS      x3,TPIDR_EL0
675 clk cpu1 R X3 0000000100000005
676 clk cpu1 IT (368) 0000000d80001294 d360fc63 0 EL0t_n : LSR      x3,x3,#32
```

Figure 6.4: AEM Tarmac file

6.3 Execution of the tool

The of RE_EXECUTES defines the number of times a test is re-executed: the PEs execute through all zones in the original test order (e.g. zone 0, followed by zone 1, and others.) and after the global synchronization following the last zone, the test will execute all zones in the test again.The ability to re-execute the zones in a test to be run multiple times allows the execution to be affected by a warm

cache and the consequent differences in timing for the same scenarios in different zones. Typical values of RE_EXECUTES for RTL simulation are between 1 and 3. Higher values can also be used to take advantage of timing variances at the cost of increasing simulation times.

The total number of instructions generated by an MP RIS Tool generated test is a function of the number of PEs (–pe), the number of zones (–nz), and the number of instructions in a zone (–ni). A lower limit on the total number of instructions generated is estimated as:

TOTAL_GENERATED_INSTRUCTIONS = (NUMBER_OF_ZONES x NUMBER_OF_PE x NUMBER_OF_INSTRUCTIONS)

The total number of instructions that are executed on the target system will in addition depend on the number of times a test is re-executed (–re):

TOTAL_EXECUTED_INSTRUCTIONS = (TOTAL_GENERATED_INSTRUCTIONSx NUMBER_OF_RE-EXECUTIONS)

	re-ex	0	re-ex	1
	zone	1	zone	1
	proc	1	proc	1
	initialization instructions	72	initialization instructions	72
	finalization instructions	32	finalization instructions	32
	sync instructions	0	sync instructions	61
	total overhead	104	total overhead	165
for 50% generation efficiency	no. of instruction	>208	no. of instruction	>165
	re-ex	0	re-ex	1
	zone	2	zone	2
	proc	1	proc	1
	initialization instructions	72	initialization instructions	72
	finalization instructions	32	finalization instructions	32
	sync instructions	61	sync instructions	61x3
	total overhead	165	total overhead	287
for 50% generation efficiency	no. of instruction	>165	no. of instruction	>144

Figure 6.5: Analysis of output of tool to verify test length

In addition to the generated instructions, there is overhead from instructions required for setting up tests and zones, address generation, and others. In addition, since every zone requires global synchronization between all PEs in the system, there is a cost associated with defining a larger number of zones since more overhead is added to the test for synchronization cycles. Thus, it is worth considering on using a lower number of zones with more instructions defined in each zone.

6.4 Reading MP RIS QA Logs

```
CPU: ananke
ARCH: v8_2_a64
Uninitialised Data Check:  enabled
Exc threshold is set to :       0
Page Attribute check(page_attr): enabled
vipt_check is set to :       0
test_efficiency check :    enabled
Number of CPUs: 2

++++++++++++++++++++++++++++++++++++++++
Page Attribute Check for Atomic Instructions :
         page_attr : PASS
CPU0 Initialization Time: 281
CPU0       Zone Init Time: 102
CPU0   Acquire Zone Time: 59
CPU0       Poll Zone Time: 28
CPU0            Test Time: 11
CPU0  Efficiency of Test: 2.2869%

CPU1 Initialization Time: 280
CPU1       Zone Init Time: 96
CPU1   Acquire Zone Time: 58
CPU1       Poll Zone Time: 98
CPU1            Test Time: 12
CPU1  Efficiency of Test: 2.20588%


+------------------------+
|QA Checks Summary       |
+----+---------+---------+
|ZONE|CHECK    |PASS/FAIL|
+----+---------+---------+
|-1  |page_attr|PASS     |
|-1  |uninit   |PASS     |
+----+---------+---------+

++++++++++++++++++++++++++++++++++++++++

Number of CPUs: 2
Total Insts: 1025
Insts of Interest: 23
QA Status: PASSED OK
```

Figure 6.6: QA Logs file

This tool also reads the tool QA logs file to read the instruction of interest given by the QA tool that is inbuilt with the MP RIS Tool. This file is as shown in figure 6.6. Then it compares the anticipated number of instructions and the instruction of interest given by the QA logs file to check if the QA Logs file are giving correct results.

Thus this post generation QA tool is efficiently used to calculate the overhead for a particular scenario of no. of zones, no. of PEs and no. of re-executes as shown in figure 6.5. So when executing the test case with same scenario the next time the no. of instructions can be pre-decided to give an efficiency of 50% or more. This would prevent the tool from generating the test cases with more no. of overhead instructions than the actual usesfull instructions.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

As we know that the verification effort is often more than the design effort with increasing design complexity of the processor design now a days.Thus constant efforts are being made to reduce the time to verify the design."Random Instruction Sequencer" (RIS) tools being the most commonly used solution across the processor design industry for verification and validation of processor design, plays an important role in verification cycle. Thus developing RIS tools would simplify the process of processor design verification which may help us to considerably reduce time to product.

The work presented here introduces the RIS Tool for single core verification and MP RIS Tool for the multi-core processing environments. It explains how both the tools are efficient in targeting the corner cases for uncovering the bugs which might not have been possible using the directed stimulus for verification of the designs. This project focuses on the development of Arm verification tools by developing plugins for RIS Tools that would in turn increase the efficiency of the tools either by increasing the accuracy or by reducing the time and efforts required for the configuration of the tools.

In this project the development of unified target configuration tool is discussed in detail which proves the tool helpful in subsequent reduction of time to configure the RIS Tool by automating the generation of the configuration file. Thus a result of unified target configuration tool, 85 to 90% of the configuration file is auto generated and only remaining 10 to 15% of the information require to be hand-coaded. This project also introduces us to the pre-generation as well as the post generation QA tools for MP RIS Tool which proves helpful in increasing the efficiency of the MP RIS Tool by discarding invalid test cases and also by increasing the usability of generated test case by calculating and remarkably reducing the percentage of added overhead. Here we have considered an example where 50% is the generation efficiency for each

test case.

7.2 Future Work

The Post-generation tool can be further enhanced to automatically calculate the no. of useful instructions for making the test generation efficiency to be 50% or some configurable value. Then a database can be maintained for every scenario of number of zones, number of PEs and number of re-executes. The pre-generation tool can be further tuned to read this database to automatically take the number of instructions as specified in the database. But since this would not allow the flexibility for user to choose the number of instructions to be generated. Another proposed solution can be to flag a warning if the number of instructions specified by the user is less for a particular scenario than that specified in the database.

References

[1] Shajid Thiruvathodi and Deepak Yeggina, A Random Instruction Sequence Generator for ARM Based Systems, 15th International Microprocessor Test and Verification Workshop (MTV) 2014, pp. 73-77, doi:10.1109/MTV.2014.20

[2] John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach, The Morgan Kaufmann Series in Computer Architecture and Design, 5th Edition, 2009, pp. 196-264.

[3] Nitin Sharma and Bryan Dickman, (2001) Verifying an ARM Core [online],Available: $https://www.eetimes.com/document.asp?doc_id = 1277271$, [Accessed: 2017, Nov 29]

[4] Vishnu A. Patankar, Alok Jain and Randal E. Bryant, "Formal Verification of an ARM processor", VLSI Design conference (1999), doi:10.1109/ICVD.1999.745161

[5] Andrei Tatarnikov, "An approach to instruction stream generation for functional verification of microprocessor designs", 2016 IEEE East-West Design & Test Symposium (EWDTS), vol. 00, no. , pp. 1-4, 2016, doi:10.1109/EWDTS.2016.7807721

[6] Deepak Venkatesan and Pradeep Nagarajan, "A case study of multiprocessor bugs found using RIS generators and memory usage techniques", 15th International Microprocessor Test and Verification Workshop (MTV) 2014, doi:10.1109/MTV.2014.28

[7] ARM infocenter, (2017) ARM DDI0487B ARMv8-A Reference Manual, Issue B.a [online], Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html, [Accessed: 2017, December 9]

[8] K Uday Bhaskar, M Prasanth, G Chandramouli and V Kamakoti, "A Universal Random Test Generator for Functional Verification of Microprocessors and System-on-Chip", 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID05), 2005, doi:1063-9667/05

[9] John Hudson and Gunaranjan Kurucheti, A Configurable Random Instruction Sequence (RIS) Tool for memory coherence in multiprocessor systems, 15th International Microprocessor Test and Verification Workshop (MTV) 2014, pp. 1550-4093, doi:10.1109/MTV.2014.26

[10] ARM ATEG RIS Tool user guide

[11] Daniel J. Sorin, Mark D. Hill and David A. Wood, A Primer on Memory Consistency and Cache Coherence, Morgan Claypool Publishers, 2011, DOI: 10.2200/S00346ED1V01Y201104CAC016.