ARM CPU Verification Methodologies

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

In

Electronics & Communication Engineering

(Embedded Systems)

By

Hardik Shah

(14MECE22)

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

May 2016

ARM CPU Verification Methodologies

Major Project Report

Submitted in partial fulfillment of the requirements
for the degree of

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

By

Hardik Shah

(14MECE22)

Under the guidance of

External Project Guide:
Mr. Rajesh C.M.
Senior Verification Engineer,
ARM Embedded Technologies Pvt. Ltd.,
Banglore.

Internal Project Guide:
Dr. Sachin Gajjar
Associate Professor in EC,
Institute of Technology,
Nirma University, Ahmedabad.

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2016

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.

- Hardik Shah

Disclaimer

"The content of this thesis does not represent the technology, opinions, beliefs, or positions of ARM Embedded Technologies Pvt. Ltd., its employees, vendors, customers, or associates."

Certificate

This is to certify that the Major Project entitled **"ARM CPU Verification Methodologies"** submitted by **Hardik B. Shah (14MECE22)**, towards the partial fulfillment of the requirements for the degree of Masterr 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. Sachin Gajjar
Internal Guide

Dr. N. P. Gajjar
Program Coordinator

Dr. D.K.Kothari
Section Head, EC

Dr. P. N. Tekwani
Head of EE Dept.

Dr. P. N. Tekwani
Director, IT

Certificate

This is to certify that the Major Project entitled **"ARM CPU Verification Methodologies"** submitted by **Hardik B. Shah (14MECE22)**, towards the partial fulfillment of the requirements for the degree of Masterr of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination.

Mr. RAJESH C. M.
Staff Verification Engineer, CPEG
ARM Embedded Technologies Pvt. Ltd.
Bangalore.

Acknowledgements

I would like to express my gratitude and sincere thanks to **Dr. P. N. Tekwani**, Head of Electrical Engineering Department, and **Dr. N. P. Gajjar**, PG Coordinator of M. Tech. Embedded Systems program for allowing me to undertake this thesis work and for his guidelines during the review process.

I take this opportunity to express my profound gratitude and deep regards to **Dr. Sachin Gajjar**, guide of my major project for his exemplary guidance, monitoring and constant encouragement throughout the course of this thesis. The blessing, help and guidance given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I would take this opportunity to express a deep sense of gratitude to Project Mentor **Mr. Rajesh C. M.**, Staff Verification Engineer, ARM Embedded Technology Pvt Ltd. for his cordial support, constant supervision as well as for providing valuable information regarding the project and guidance, which helped me in completing this task through various stages. I would also thank to **Mr. Desikan Srinivasan**, Principal Engineer, ARM Embedded Technology Pvt Ltd. and **Mr. Saransh Jain** for always helping, giving good suggestions and solving my doubts and guide me to complete my project in better way.

I am obliged to all the members of CPEG, ARM Embedded Technology Pvt. Ltd. for the valuable information provided by them in their respective fields.

Lastly, I thank almighty, my parents, and friends for their constant encouragement without which this assignment would not be possible.

- Hardik Shah

14MECE22

Abstract

Main aim of this project is to do support work for various ARM CPU verification methodologies. This project includes scripting work, C programming work and assembly test writing work to evaluate certain ARM architecture features. Team is only concern with Final Id that is made of sub id, test name and context name, so YAML parser is a perl script that is used to parse specific filed to make Final Id.

Like .exe file we can not open .ELF file so ELF parser is a C program to read all values from ELF Header Table, Section Header Table and Program Header Table and to print them. With the help of this ELF parser code merging of ELF is achieved. ELF parser code is used to merge 200 ELF at one time. This Final merged ELF is used in Self Modifying Code to run each ELF at run-time.

Self Modifying Code tests are written in assembly language with the help of A64 and A32 instruction set. These codes are used to alter the flow of execution at run time. Self Modifying test code loads a merged ELF at run-time and execute them. The runtime flow of ELF is used to check different aspect like virtual aliasing and cache invalidation.

There are three types of verification methodologies: AVS, DVS and RIS. In AVS methodology, number of self written assembly test are made to run on ARM architecture to verify its behaviour. Debugging of failed test, report analysis, various comparison report with different hardware, final pass rate analysis and skipp test analysis come under AVS verification methodology.

Contents

List of Figures

Abbreviation Notation and Nomenclature

ARM ARM ..... ARM Architecture Reference Manual
AVS ..... Architecture Validation Suits
AVK ..... Architecture Validation Kit
DVS ..... Device Verification Suits
DUT ..... Device Under Test
MPU ..... Memory Protection Unit
PMSA ..... Protected Memory System Architecture
YAML ..... Yet Another Mark-up Language
FIQ ..... Fast Interrupt Request
IRQ ..... Interrupt Request
ELF ..... Executable Linkable Format
RIS ..... Random Instruction Sequence
UTB ..... Unified Test Bench
APSR ..... Application Program Status Register
IPSR ..... Interrupt Program Status Register
EPSR ..... Exception Program Status Register
SCS ..... System Control Space
WFI ..... Wait For Interrupt
VAL ..... Validation Abstraction

Chapter 1

Introduction

1.1 Motivation

The CPU Engineering Group(CPEG) at ARM aims to verify CPU as a whole including its compliance. To verify different architecture features, different verification methodologies and tool are used. RAVEN is a RIS tool used for CPU verification and YAML file is part of the flow of RAVEN tool. Team is only concerned with the Final Id of input YAML file. So it is required to make a script that parse Final Id from the YAML file.
Historically, self-modifying code verification is weak in ARM with little support for this in RIS tools.This project aims to fill the gap using a new method based on the real use case of it in Android.In Android, java byte codes are converted into ARM instructions at run-time. This is mimicked here by generating no. of RIS tests using Raven tool with identical memory map and then a handwritten test load and execute these RIS tests sequentially. The flow is also used to verify virtual aliasing, cache invalidation. This is done for both AArch32 and AArch64 instruction sets.For that a C program to generate RIS tests and merge them into single test and number of handwritten tests to load and execute RIS test sequentially are required.

1.2 Objective

To verify different architecture features, different verification methodologies and tool are used.There is need of script that is used to parse input YAML files and make one output file that only contains Final Id.
To verify certain architecture features like virtual aliasing, cache invalidation and runtime behavior, it is required to write a C program to generate RIS tests and merge them into one single test and number of handwritten code to load and execute them.
Optimization and automation of regression process required to save cluster time, cluste size and manual work.

1.3 Problem Statement

ARM CPU Verification Methodology includes:

- Write a PERL script to parse Yet Another Mark-up Language(YAML)[4] files and create a merged YAML file as a part of flow for next generation ARM CPU verification.

- Write a C program to parse many Executable and Linkable Format(ELFs) and create a merged ELF.

- Write Self Modifying assembly tests directed towards verification of certain Architecture features like virtual aliasing and cache invalidation.

- Regression and Verification process.

- Optimization of flow of Regression.

- Automation of Regression process.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *Literature Survey* , gives overview ARMv8-A and ARMv7-M architecture, Execution states, System Register and ARMv7-M exception model.

Chapter 3 , *YAML Parser* , highlights the structure of a YAML file and YAML::TINY package that is used to parse YAML file.This chapter also describes how to read and write YAML file with an example.

Chapter 4, *ELF Parser* , highlights the well defined structure of an ELF file. This chapter also describes the different attributes of Header Table, Program Header Table and Section Header Table.

Chapter 5, *Self Modifying Code* , highlights the basic definition of Self Modifying Code and working of it. This chapter describes three different type of Self Modifying Code tests.

Chapter 6, *AVS Regression and Verification* , highlights the aim and steps of Regression Process and different types of reports.

Finally in **Chapter 7** , *Conclusion and Future Scope* , concluding remarks and scope of future work is presented.

Chapter 2

Literature Survey

The ARM architecture is basically RISC architecture. RISC stands for Reduced Instruction Set Computer. The ARM architecture includes following RISC architecture features:

- ARM architecture has a large uniform register file.
- ARM architecture supports load/store mechanism, in which data-processing related operations based only on register contents, not on memory content.
- ARM architecture supports simple addressing modes, with all load/store addresses related to register content and instructions only.

The ARM architecture defines interrelation of Processing Element with memory, caches, and a memory translation system. ARM architecture also states in which way multiple Processing Elements inter relate with each other and with the observers of the system. It supports implementation across a wide range from a performance points.
Basically, three main key attributes of the ARM architecture: Implementation Size, very low power consumption and performance.
There are two Execution state in ARMv8 architecture[1]:

- A 64 bit Execution state called as AArch64 Execution state.
- A 32 bit Execution state called as AArch32 Execution state.

AArch32 Execution state is backward compatible with previous version of architecture. Both Execution state AArch64 and AArch32 supports floating point and SIMD (Single Instruction Multiple Data) instruction.
AArch32 Execution state includes:

- SIMD base instruction sets operate on 32-bit general-purpose registers.
- Advanced SIMD instructions sets operate on SIMD and floating-point register file.
- Floating-point instructions sets operate on SIMD and floating-point register file.

AArch64 Execution state includes:

- Advanced SIMD instructions sets operate on SIMD and floating-point register file.
- Floating-point instructions sets operate on SIMD and floating-point register file.

The AArch32 is backward compatible with the ARMv7-A architecture profile, support some features that are there in the AArch64. So backward compatibility with previous ARMv7-A architecture enhances a profile that supports more feature in AArch64 Execution state.

2.1 ARM Architecture Profiles

ARM architecture has evolved significantly since its introduction, continues to develop it. There are eight major versions of the ARM architecture have been defined, noted by the version numbers 1 to 8.
Listed below Architecture and its Processor Family:

- ARMv1:ARM1
- ARMv2:ARM2, ARM3
- ARMv3:ARM6, ARM7

- ARMv4:Strong ARM, ARM7TDMI, ARM9TDMI
- ARMv5:ARM7EJ, ARM9E, ARM10E, XScale
- ARMv6:ARM11, ARM Cortex-M
- ARMv7:ARM Cortex-A, R, M

The generic names AArch64 and AArch32 describe the 64-bit and 32-bit Execution states. AArch64 Execution state is the 64-bit state, so that addresses are held in 64-bit wide registers, and base instructions set can address 64-bit registers for their processing. A64 instruction set is supported by AArch64 Execution state.
AArch32 Execution State is the 32-bit state, so that addresses are held in 32-bit wide registers, and base instruction sets can address 32-bit registers for their processing. AArch32 Execution state supports two types of the instruction set like: T32 and A32 instruction set.
ARM itself name defines three basic architecture profiles[1]:
A Application profile:

- Supports a MMU (Memory Management Unit) based VMSA (Virtual Memory System Architecture).
- Supports all the instruction set: A64, A32, and T32 instruction sets.

R Real-time profile:

- Supports a MPU (Memory Protection Unit) based PMSA (Protected Memory System Architecture).
- Supports the only A32 and T32 instruction sets.

M Microcontroller profile:

- M profile implements a programmers' model that specially designed for low-latency interrupt processing.

- Implements a variant of the R-profile.
- Supports a variant of the T32 instruction set.

ARMv8 architecture introduces many changes to the ARM architecture, while maintaining backward compatibility with previous versions of the architecture.

2.2 ARMv8-A Execution State[1]

There are two Execution states in ARMv8-A architecture:
AArch64: The 64-bit Execution state. This Execution state:

- Includes 31 64-bit registers for general-purpose use from which X30 register is used as the procedure link register to store return address.
- Includes a 64-bit stack pointer (SP), a 64-bit Program Counter (PC) and Exception Link Registers (ELRs).
- Includes 32 128-bit registers to support SIMD vector and scalar floating-point operation.
- Supports only one instruction set A64.
- Includes the ARMv8 Exception model, with four different Exception levels, EL0 - EL3 to support an execution privilege hierarchy.
- Includes 64-bit virtual addressing.
- Includes a number of Process state (PSTATE) elements that hold PE State.

AArch32: The 32-bit Execution state. This Execution state:

- Includes 13 32-bit register for general-purpose use, and a 32-bit Stack Pointer, Program Counter and a 32-bit Link Register (LR). Link Register is used as both Exception Link Register and Procedure Link Register.

- Includes a single Exception Link Register (ELR) to store exception returns address from Hyp mod.

- Includes 32 64-bit registers to support Advanced SIMD vector and scalar floating-point operation.

- Supports two instruction sets, T32 and A32.

- Includes the ARMv7-A Exception model and maps this onto the ARMv8 Exception model.

- Includes 32-bit virtual addressing.

- Includes a number of Process state (PSTATE) elements that hold PE State.

2.3 ARMv8-A Instruction Sets

In ARMv8-A architecture supports three different type of instruction set based on in which Execution state it is.

AArch64 Execution state supports only A64 instruction set. A64 is a fixed-length instruction set with 32-bit instruction set encodings.

AArch32 Execution state supports two types of the instruction sets:

A32: A32 instruction set is a fixed-length instruction set with 16-bit instruction encodings.

T32: It is also called as Thumb instruction set. T32 is not a fixed-length instruction set. It is a variable-length instruction set with both 32-bit and 16-bit instruction encoding.

These instruction sets are also known as Thumb and ARM instruction sets.

ARMv8 architecture extends each of these three instruction sets.

2.4 ARMv8-A System Registers

System registers are used to provide status and control information of architecture features.

Standard naming format is used to call the system register.

Register name is used to identify specific registers and bit filed name is used to control and status bits within a system register.

The ARMv8-A architecture includes two register files named as: A general-purpose register file and A SIMD and floating point register file.

In each of these register file, register widths depend only on in which Execution state it is.

In AArch64 Execution state:

A general-purpose register file includes only 64-bit width registers:

- These general purpose registers can be accessed as 64-bit registers or as 32-bit registers, using only the 32 bits.

A SIMD and floating-point register file includes 128-bit width registers:

- Only the quad word integer data types and floating point data types can access the SIMD and floating-point register file.
- Depending upon which type of A64 instruction is used, the effective vector length of register can be decodes as 128-bit or 64-bit.

In AArch32 Execution state:

A general-purpose register file includes 32-bit width registers:

- Two 32-bit width registers can be used support a double word.
- A general-purpose register file support vector formatting.

A SIMD and floating-point register file includes 64-bit width registers:

- AArch32 Execution state does not support floating-point and quad word integer data types.

2.4.1 System Register In AArch64

In the AArch64 Execution state, a system register can be viewed as:
There are 31 general purpose register in AArch64 state named as R0-R30.
Each register can be accessed as:

- A 64-bit general-purpose register named as X0 to X30.
- A 32-bit general-purpose register named as W0 to W30.

As shown in below figure one system register can be accessed as 32-bit width register named as Wn and 64-bit width register named as Xn.

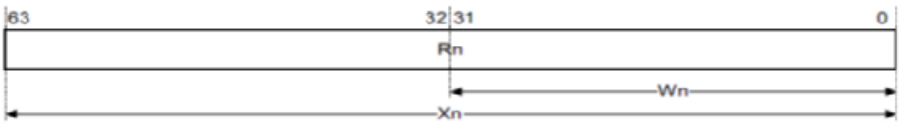

Figure 2.1: System Register In AArch64[1]

Stack Pointer: Stack Pointer register is of 64-bit width register in AArch64. One can access the least significant 32 bits of Stack Pointer register as register name WSP. By writing SP in operand of an instruction one can use the current stack pointer.
Program Counter: A 64-bit Program Counter is used to hold the address of the current instruction. PC value cannot update by software. The value of Program Counter can only be updated on an exception entry, branch or exceptions return.
There are 32 128-bit wide SIMD and floating-point registers named as V0 to V31. As shown in below figure each one register can be accessed as:

- A whole 128-bit register can be accessed as named Q0 to Q31.
- A 64-bit register can be accessed as named D0 to D31.
- A 32-bit register can be accessed as named S0 to S31.
- A 16-bit register can be accessed as named H0 to H31.
- An 8-bit register can be accessed as named B0 to B31.

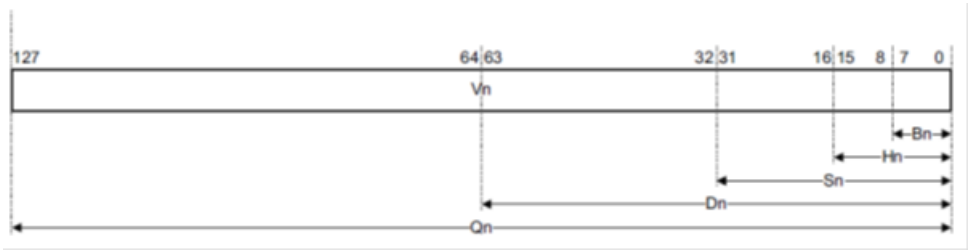

Figure 2.2: SIMD Register In AArch64[1]

2.4.2 ARMv8-A Processor State

PSTATE is known as Processor state is an abstraction of information of processor state. PSTATE provides information of processor execution state and in which state it is working.

The PSTATE provides information that is accessible at EL0:

The condition flags:

The Conditional flags can be set by flag-setting instructions. They are:

N: Negative condition flag.

Negative condition flag is set as the result of the instruction regarded as a two's complement.

- It sets to 1 if the result is negative.
- 0 if the result is positive or zero.

Z: Zero condition flag.

- It sets to 1 if the result of the instruction manipulation is zero.
- 0 for other values.

Set value of Zero condition flag is often indicates an equal result of a comparison values.

C: Carry condition flag.

- It is set to 1 if the instruction results in a carry overflow, like an unsigned integer overflow after the result of an addition operation.

- It sets to 0 otherwise.

V: Overflow condition flag.

- It sets to 1 if the instruction results in an overflow condition, like a signed overflow because of an addition operation.
- Its sets to 0 otherwise.

ARMv8 architecture also support conditional execution of instruction in which it test the N, Z, C and V condition flags to determine whether the instruction must be executed or not. So the execution of the particular instruction is conditional based on the previous instructions result.

The exception masking bits[1]:

D, A, I, and F bits are known as exception masking bits. Each bit there are two values: if it has 0 then exception not masked and if it has 1 then exception masked.

D bit is known as Debug exception mask bit. This bit is modified when EL0 is enabled. This bit is architecturally ignored at Exception level 0.

A SError interrupt mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

2.5 ARMv7 Architecture[5]

ARMv7 architecture is documented as a set of different profiles as ARMv8 architecture.

- **ARMv7-A** This architecture supports both ARM and Thumb instruction sets and virtual address support in the MMU.
- **ARMv7-R** This architecture also supports ARM and Thumb instruction set and only physical address support in MMU.

- **ARMv7-M** This micro-controller architecture supports only Thumb instruction set. In this profile deterministic operations and overall size are more important.

2.6 ARMv7-M Architecture Profile

ARM has introduced this ARMv7-M for micro-controller application, complementing its strength in real embedded and high performance markets.
Key features for ARMv7-M are[5]:

- Enable implementations with performance, low power and area constaints.
  Provide simple pipeline design
- Deterministic operation.
  Low cycle execution
  Cacheless operation
  Minimal interrupt latency with short number of pipeline
- C C++ target
- Design for embedded system
  Low pin-count
  Enable new entry level opportunities

2.7 ARMv7-M Instruction Sets

ARMv7-M profile only supports Thumb instructions. If we enable Floating Point extension then it adds Floating instruction to the Thumb instruction.
ARMv7-M supports two type of instruction extension. One is DSP extension and another one is Floating Point extension. DSP extension adds the ARM Digital Signal Processing instruction to the Thumb set. These instruction includes Singal Instruction Multiple Data (SIMD) instructions.

2.8 ARMv7-M Registers

The application level model of architecture provides details of the special purpose and general purpose registers. These registers are used to load and store values from memory and manipulating data.

2.8.1 ARM Core Registers

There are 13 general purpose 32-bit registers starts from R0 to R12 and additional 3 32-bit registers with their specific usage.

- **StacK Pointer (SP)** used to point to the active stack. SP is banked register of SP Main and SP Process. The current which stack register is used that depends on core mode and the value of CONTROL SPSEL bit.
- **LinK Register (LR)** used to store return address from a subroutine that entered using BL instruction. At reset value of LR is 0xFFFFFFFF. The value of LR is also updated on exception entry and return.
- **Program Counter (PC)** used to store the address of next instruction that is going to fetched. PC also refereed as R15.

2.8.2 Program Status Registers xPSR

The special purpose Program Status Register is a 32 bit register that combination of three sub registers called as APSR, IPSR and EPSR.

Application Program Status Register APSR

Flag setting instruction modify the flags bits of APSR and the processor uses there flags combination for conditional execution of instruction. GE bits are used in DSP extension implementation.

Interrupt Program Status Register IPSR

The value of IPSR is getting updated on exception entry and return. It holds the value of

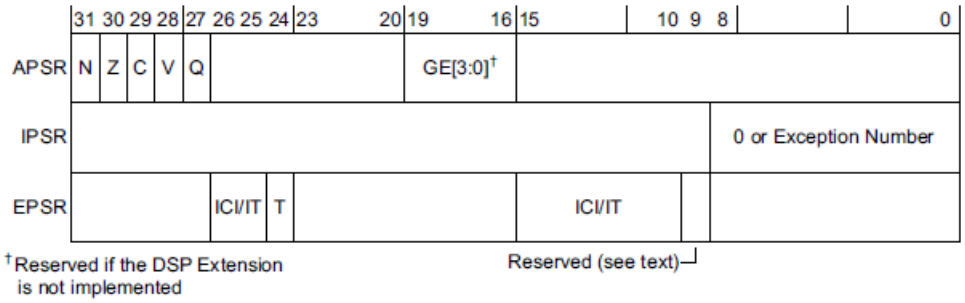

Figure 2.3: The xPSR Register[5]

exception number of currently executing exception.

Execution Program Status Register EPSR

The EPSR T bit, if it is set to 1 that indicates the processor executing Thumb instructions set. IT and ICI field of EPSR supports IT and interrupt continue load store instruction. Combination of above these three registers accessed by xPSR.

2.8.3 Mask Registers[5]

The special purpose Mask registers used for execution priority boosting.

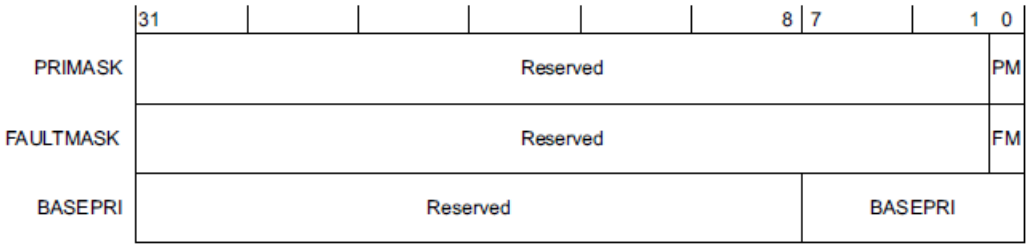

Figure 2.4: The Special Purpose Mask Registers[5]

PRIMASK

This is a 1 bit register used to set execution priority to 0.

BASEPRI

The base priority mask is an 8 bit register used to change preemption execution priority.

FAULTMASK

The fault mask is a 1 bit register used to raise execution priority to -1, priority of Hard-Fault.

2.8.4 CONTROL Register[5]

The special purpose CONTROL register is a 3 bit or 2 bit register.

nPRIV, bit[0]

Defines which type of execution privilege:

0 Privileged access in Thread mode.

1 Unprivileged access in Thread mode.

SPSEL, bit[1]

Defines which stack is used:

0 Use SP main as Stack Pointer in Thread mode.

1 Use SP process as Stack Pointer in Thread mode

FPCA, bit[2]

Defines whether the Floating Point extension is active or not:

0 Not active.

1 Active.

2.9 ARMv7-M Exception Model

The ARMv7-M architecture supports the following excepions:

Reset

There are two levels of reset in ARMv7-M. One is Power on reset and another one is Local reset. Which registers fields are forced to their reset value depends upon applying level of reset.

The Reset exception has a fixed priority of -3 and it is permanently enabled.

NMI

Non Maskable Interrupt is the highest priority than reset. It has a fixed priority of -2 and it is also permanently enabled.

HardFault

HardFaut has a fixed priority of -1. HardFault is the fault that exists for all cases of fault that can't be handled by other mechanism.

MemManage

It will handle memory protection faults generated by Memory Protection Unit or by memory constrains for both data and instruction transaction. It has a configurable priority.
If MemManage fault is not enabled then it escalates to HardFault.

BusFault

It handles memory related faults that generates during bus operation. BusFault has a configurable priorty. If it is not enable then it escalated to HardFault.

UsageFault

It handles non memory related faults happened due to execution of instruction. A number of reason that will generate UsageFault:

- Undefined Instruction.
- Invalid state on instruction execution.
- Error on exception return sequence.
- Access to disabled coprocessor.
- unaligned access.
- Division by Zero.

SVCall

SVC instruction will cause this exception. It has a configurable priority.

Interrupts

The ARMv7-M architecture supports 496 external interrupts and two system level interrupts. Each of them have configurable priority.

The two system level interrupts are:

- **PendSV** Used to handle software generated calls.
- **SysTick** Generated by SysTick timer that is a part of ARMv7-M processor.

Each exception has an associated exception number as:

Exception Number - **Exception Name**

1 - Reset

2 - NMI

3 - HardFault

4 - MemManage

5 - BusFault

6 - UsageFault

7-10 - Reserved

11 - SVCall

12 - DebugMonitor

13 - Reserved

14 - PendSV

15 - SysTick

16 - External interrupt 0

2.9.1 Execution Priority and Priority boosting

The execution priority is defined as the highest execution priority determined from below:

- The base level value of execution priority.
- Highest priority of all active exception.

- The impact of priority boosting registers.

textbfPriority Boosting

Software can use below registers to boost the execution priority:

PRIMASK

Raises the execution priority to 0.

FAULTMASK

Raises the execution priority to -1, priority to HardFault.

BASEPRI

Software can write value N to this register to set execution priority. When this register is cleared then it has no effect.

2.9.2 Stacking on Exception Entry

The architecture guarantees that stack pointer value should be aligned as 4 bytes.There are two types of frame that stacked during exception entry.

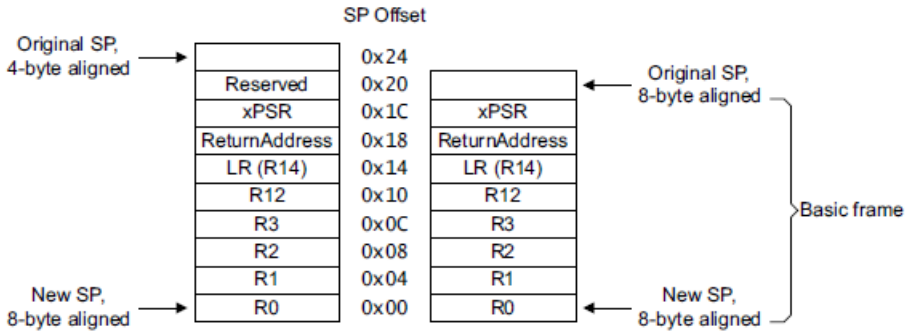

Figure 2.5: Stacking of Basic Frame[5]

Basic Frame is made of registers R0-R3, R12, Link Reguister, Return Address and xPSR. This basic frame stacked during the exception entry.

Extended Frame is stacked when Floating Point Extension is enabled. This frame stacked Basic Frame register with Floating Point register S0-S15 and FPSCR.

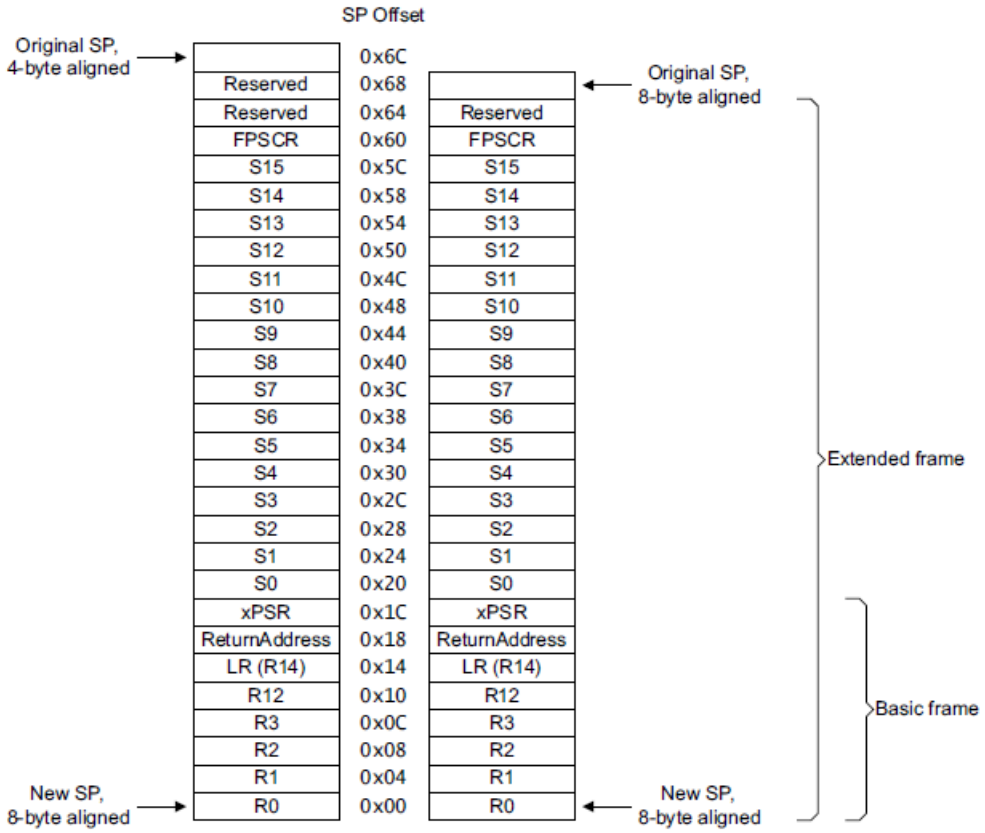

Figure 2.6: Stacking of Extended Frame[5]

2.9.3 Exception Return

An exception return occurs when the processor is in Handler mode and one of the instruction loads 0xFxxxxxxx value to the Program Counter[5]:

- POP/LDM instreuction that include loading the Program Counter.
- BX with register.
- LDR instruction with PC as a destination.

subsectionException on Exception Entry During the exception entry process other exception can occur due to fault on an operation involved in exception entry or an interrupt with higher priority come.

Late-arriving Exceptions[5] The architecture does not specify the point at which high

priority exception come during the exception entry. To support low interrupt latency, the architecture allows high priority exception to become active, without causing that entry sequence to repeat.
The exception entry process started by exception can be used by late-arriving exception to ensure low interrupt latency.

2.9.4 Exception on Exception Return

During exception return sequence other exception can affect behaviour, either because of a fault on the operation performance during exception return, or due to an asynchronous exception that is higher priority level that the exception return is returning to.
Tail-chaining Exception[5] It is the optimization of an exception entry and exception return sequence by removing stacking and unstacking process. newline A Tail-chaining mechanism use in the following case:

- To handle a derived exception.
- As an optimization to improve interrupt latency response.

Chapter 3

YAML Parser

YAML stands for Yet Another Mark-up Language. YAML is one kind of file structure used to represent data in hierarchical order. YAML parser is a perl script that used to parse specific data from YAML file. To parse the specific object or data from YAML file YAML::TINY package is used.

3.1 Introduction

YAML (rhymes with camel) is a data serialization and human-readable format[4]. YAML takes concepts from different programming and scripting languages such as Perl, Python and C. These languages are used to parse data from YAML file.

YAML file syntax was designed to be easily addressed to data types. It used list, associative array and scalar to describe data like high-level languages. It is in human readable format so humans are easily able to edit data structures. YAML file structure is used to describe data structures such as configuration files that are dumped during debugging process.

YAML file structure is well-suited hierarchical data representation so it easy to grep specific data using scripting languages likes Perl and Python. As shown in below figure YAML structure is quite easily readable and understandable. YAML file structure starts

with — syntax. Here in below figure customer (parent object) is combination of given and family object.

```
---
Receipt:     Oz-Ware Purchase Invoice
Date:        2012-08-06
Customer:
    Given:   Dorothy
    Family:  Gale

Items:
    - part_no:   A4786
      Descrip:   Water Bucket (Filled)
      Price:     1.47
      Quantity:  4

    - part_no:   E1628
      Descrip:   High Heeled "Ruby" Slippers
      Size:      8
      Price:     100.27
      Quantity:  1
```

Figure 3.1: YAML Example[4]

3.2 YAML::TINY

YAML::Tiny is a Perl class used for writing and reading YAML-style structure files. It is a little code written in Perl to reduce memory overhead and load time.This YAML::TINY module is primarily used to reading human-written YAML structure files (like simple config files) and to generate very simple human-readable files.

The one disadvantage of YAML::Tiny module is that it does not generate comments and preserve the order of object hashes. Only a subset of the Full YAML file is supported by YAML::Tiny package.[4]

3.3 YAML Example(Read)

Given below is the Perl script that is used to parse the data of a YAML file and print it. It uses YAML::TINY package by declaring with use clause.To declare YAML::TINY package one has to define it by use clause as shown below:

use Tiny;

Or in another method

use YAML::TINY;

As shown in figure items object is collection of array of different object like Price and Quantity.To parse these all object value one has to write for loop as below:

```
my $yaml=Tiny->read ('xyz.yaml');
my @part_no, @price, @quantity;
for (my $i=0; $i<=10; $i++)
{
   $part_no [$i] = $yaml-> [0] --> {items} -> [$i] -> {part_no};
   $price [$i = $yaml-> [0] --> {items} -> [$i] -> {price};
   $quantity [$i] = $yaml-> [0] --> {items} -> [$i] -> {quantity};
}
```

Figure 3.2: YAML Read Example

3.4 YAML Example(Write)

YAML::TINY package is also used to generate YAML file structure. One can write values to any object of YAML file using YAML::TINY package.

Given below code is used to generate new YAML file structure using YAML::Tiny package.

open my fh ,”write.yaml” or die ”could not open the file:!”;

Now if one wants to generate yaml file as shown in above YAML Example then one has to write below code:

```
my $yaml=Tiny->new ([{Receipt=>"Oz-Ware Purchase Invoice",
                      Date=>"2012-08-06";
                      Customer=> [{
                                   Given=>"Dorothy",
                                   Family=>"Gale",
                                  }]
                    } ] )
$yaml->write ('write.yaml');
```

Figure 3.3: YAML Write Example

Chapter 4

ELF Parser

ELF stands for Executable and Linkable Format. Like .exe file we cannot open .elf file. So ELF parser is a C program used to parse and print all values of ELF file. Format of ELF file is same universal and well defined.

4.1 Introduction

There are two types of ELF file one is Re-locatable file and another one is Loadable file. An Executable and Linkable Format object file consists of the following parts:

- **ELF File Header**, which is the beginning of the file.
- **Section Header Table**, is required for re-locatable files, and optional for loadable files.
- **Program Header Table**, is optional for re-locatable file but required for loadable files. In below two figures fixed structure of both re-locatable and loadable file is shown.

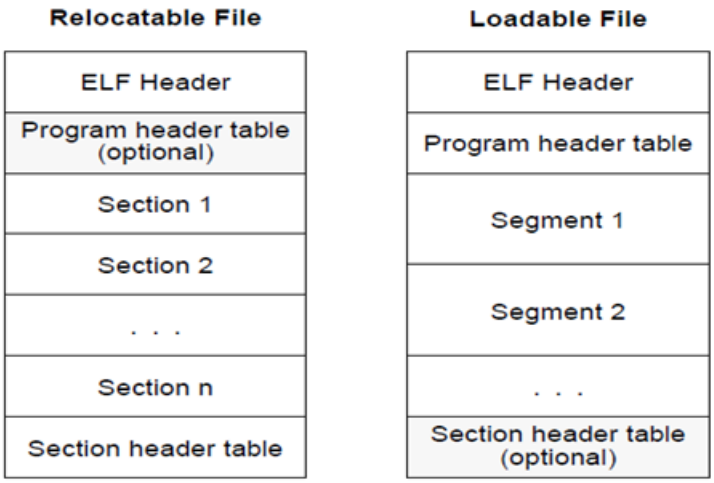

Figure 4.1: Structure Of An ELF File[2]

4.2 ELF File Header

The ELF file header is always located at the beginning of the file. It is of fixes size and used to locate the Section Header Table and Program Header Table relatively from the beginning of the file. All the size of all parts of ELF file is in bytes.
The structure of ELF File Header is shown below. It is defined as a structure in C.

typedef struct

{

unsigned char eident[16];

Elf64_Half e_type;

Elf64_Half e_machine

Elf64_Addr e_entry;

Elf64_Off e_phoff;

Elf64_Off e_shoff;

Elf64_Half e_ehsize;

Elf64_Half e_phentsize;

Elf64_Half e_phnum;

Elf64_Half e_shentsize;

Elf64_Half e_shnum;
} Elf64_Ehdr[2];

e_ident provides information about ELF file as object file and data representation of it.
e_type identifies the which type of file ELF is. If it has 1 value then it is Re-locatable file and if it has value 2 then it is an Executable file. e_machine provides information about architecture of target.
e_entry provides the address of entry point of program in bytes.
e_phoff provides the offset address of the first Program Header Table in bytes.
e_shoff provides the offset address of the first Section Header Table in bytes.
e_ehsize contains the size of ELF Header in bytes.
e_phentsize and e_phnum provides the information related to program Header Table. e_phentsize contains the size of Program Header Table in bytes and it is fixed for all program header tables and e_phnum contains the total number of entries of Program Header Table.
e_shentsize and e_shnum provides the information about the Section Header Table. Both provides same info as above discussed but for Section Header Table.

4.3 Section Header Table

Size of the each section header table is fixed. Each section has its own Section Header Table which conveys all information regarding that section.
The structure of Section Header Table is shown below. It is defined as a structure in C.
typedef struct {
Elf64_Word sh_name;
Elf64_Word sh_type;

Elf64_Xword sh_flags;

Elf64_Addr sh_addr;

Elf64_Off sh_offset;

Elf64_Xword sh_size;

Elf64_Xword sh_entsize;

} Elf64_Shdr[2];

There is one string table which contains the all the strings used for symbol and section names. It is an array of strings. sh_name provides the offset value of the section name, in bytes, relative from the beginning of the string table. Using sh_name one can get the name of the section stored in string table.

sh_type identifies the which type of section it is. sh_flags conveys the information regarding the different attributes of section.

sh_addrs provides the virtual address of the particular section in memory. This filed is shown zero if section is not allocated any memory. sh_offset provides the offset of the section beginning in the file in bytes whereas sh_size contains the size of section in bytes. sh_entsize provides size of each section entries in bytes.

4.4 Program Header Table

The Program Header Table provides information for program segment that stored in memory. Each program segment has its own Program Header Table and size of each is fixed.

The structure of Program Header Table is shown below:

typedef struct

{

Elf64_Word p_type;

Elf64_Word p_flags;

Elf64_Off p_offset;

Elf64_Addr p_vaddr;

Elf64_Addr p_paddr;

Elf64_Xword p_filesz;

Elf64_Xword p_memsz;

} Elf64_Phdr[2];

p_type defines the which type of segment is.p_flags describes different attributes of a segment.Top 8 bits among 16 bits are reserved for specific use and rest of 8 bits are used for environment use.

p_offset gives value of offset of segment from the beginning of the file in bytes.

p_vaddr and p_paddr provide virtual address and physical address respectively in bytes.

p_filesz provides the size of segment in memory and p_memsz provides the size of memory image segment.

ELF Parser is a C program that used to parse each entry of Program Header Table and Section Header Table. ELF Parser is also used in program to merge ELF. In merge ELF program ELF Parser is called to extract value and later this values are used while merging.

Chapter 5

Self Modifying Code

Self Modifying Code alters its own instruction while it is executing. It changes the flow of execution at run time.Self Modifying Code is used to verify virtual aliasing and cache invalidation.

5.1 Introduction

Self Modifying codes are written in assembly language. A64 and A32 instruction sets are used to write Self Modifying Code.As a part of my project i have written three types of Self Modifying tests.

- Self Modifying Code Basic
- Self Modifying Code Repeat
- Self Modifying Code Virtual Alias

Above all these three test are written for both AArch64 and AArch32 instruction set. These tests load raven ELFs run-time and execute them. Each ELF is of 4k bytes.

5.2 Self Modifying Code Basic

This is the simplest test among all. This test first loads whole 4k bytes of ELF from virtual address 0x80000000 to virtual address 0x00000000. LDR and STR instructions are used to load whole ELF to 0x00000000 virtual address.
After loading whole 4k bytes of ELF to 0x00000000,it starts executing ELF from 0x00000000 address at run-time.
This test is written to run 200 ELFs. First ELF is start at 0x80000000 and next ELF starts after 4k bytes at 0x80001000 and so on.So the second ELF is stored from 0x80001000 to 0x00000000.

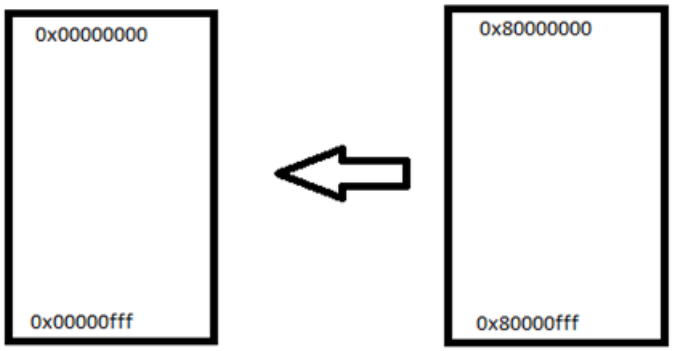

Figure 5.1: Working Of Self Modifying Code Basic

5.3 Self Modifying Code Repeat

The functionality of this test is same as basic one, first it loads a whole 4k byte of ELF from virtual address 0x80000000 to 0x00000000.Then it jumps to 0x00000000 to run ELF. This test is written to run each ELF four times before going to next ELF.

5.4 Self Modifying Code Virtual Alias

For this test there are 8 different virtual alias addresses like: 0x90000000, 0x90001000, 0x90002000, 0x90003000, 0x90004000, 0x90005000, 0x90006000 and 0x90007000 are predefined and mapped to 0x00000000 virtual address.
In this test, first it loads 4k ELF bytes from virtual address 0x80000000 to 0x00000000.Then it jumps to first virtual alias. As this virtual alias mapped to 0x00000000 , it starts executing ELF from 0x00000000. So in this code each ELF jumps to different predefined virtual alias, so it runs eight times.

So basically Self Modifying Tests are used to check run time behaviour of architecture. It is also used to verify cache invalidation and virtual aliasing.

Chapter 6

AVS Regression and Verification

There are three types of verification methodology used: AVS, DVS and RISC. AVS stand for Architecture Verification Suite. In AVS Regression procedure number of handwritten test cases are applied to architecture to verify its functionality.

6.1 AVS Regression

Main step of this AVS regression to apply number of different test cases to architecture and to see whether these all test cases are getting passed on architecture or not. These hand written test cases are provided by different team. we have to apply these all test cases to main architecture and to see final pass rate result.

These all test cases are written in assembly language using A64 and A32 instruction set. All test cases are belong to different types of suites like DEBUG, INT. Test cases are written related to their parent suites and included related instruction. So all the test cases are grouped into different suites.

AVS Regression procedure also include result analysis, comparison report analysis, report generation, uploading on data base, debugging and checking of failed test cases. There are some test cases that are getting abandoned and failed as a result then these test needed to be verify by dumping into tarmac file.

There are two types of report to be generate after the completion of whole regression. one is .rep report which used to calculate pass rate of each suite and another type of report is .csv report which used to do comparison and analysis of test cases. one can easily find total number of test was getting failed and from which suites. After this debugging process final result should be uploaded on val spider. val spider is one type of data base to store result of regression. One can easily find the result of previous regression on val spider.

6.2 Regression Process

Whole Regression Process includes:

- Set up of World for Regression
- Build RTL process.
- Compilation process.
- Applying Regression commands.
- Result analysis.
- Skip analysis
- Generating pass rate graphs and reports.
- Uploading of results.

As per above steps we can find that regression process is very long and tidy. Each steps requires sets of commands. So automation of Regression process is required. I wrote a perl script that take care of all above steps. So automation of Regression process saves your time and effort.

6.3 ELF Based Regression

ELF based regression is one the optimization approach of regression process.It will save cluster size as well as cluster time.

In regression process we have to run around thousands number of test on cluster to get result. We have to submit test cases in terms of jobs on cluster. Cluster will run parallel jobs as per the availability of parallel slots.

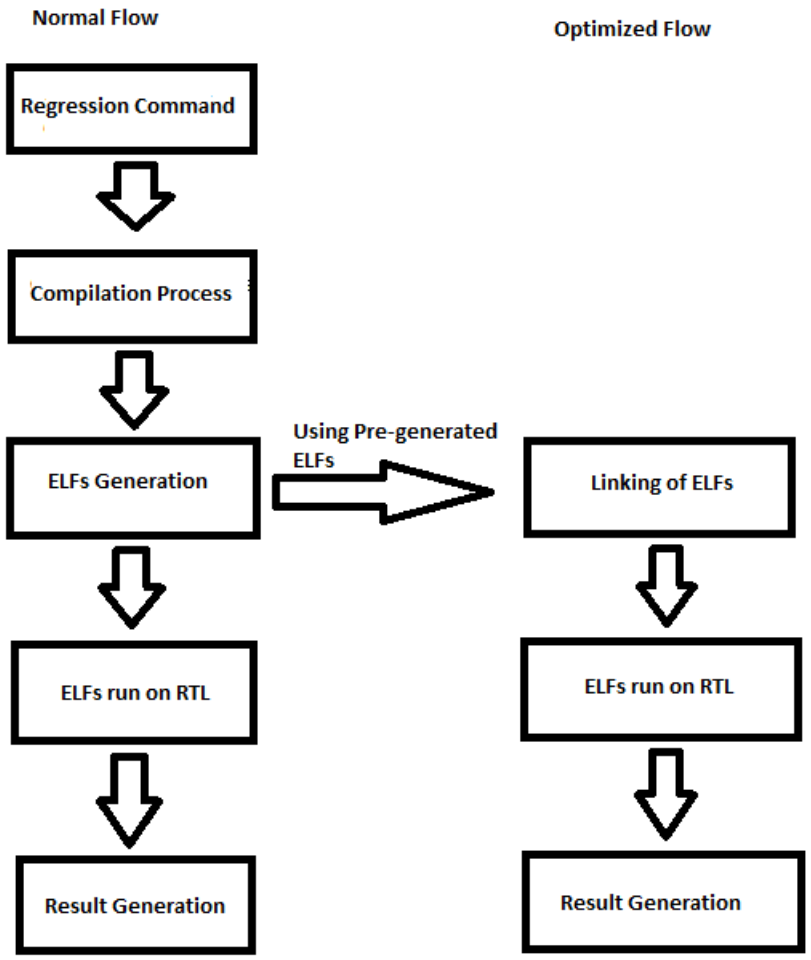

Figure 6.1: ELF based Regression

As per the normal flow shown in figure, each test has to pass through below stage:

- 1st stage is running regression command on cluster. For different hardware

configuration regression command will be different.

- Second stage is compilation stage. In this stage each of test is compiled by RTL and ELF is generated for each and every test cases. So we have all ELFs at the end of the compilation process.
- In third stage each ELF run on RTL.
- In final stage we will get results.

So ELFs generated during each weekly regression.So i implemented way to use already generated ELFs as shown in optimized flow.This whole process dumped into log file and stored on cluster. Each ELF required some cluster time to finish whole procedure.

As we use pre-generated ELF for our process so 1st two stage is not needed. Dumped log file starts with third stage only. The log size of new dumped log is lesser than the previous one. So in this manner it will save cluster size.

As we used pre-generated ELFs for our new regression, cluster will save time as compilation process is not there. So overall it will save huge cluster time. It will save 30% cluster time.

So optimized flow of regression process saves cluster time and cluster size.

6.4 Verification and Debugging

Verification is a process of evaluating the failed test cases in order to get higher percentage of pass rate. Once the test case get failed, it has to be verify by dumping the failed test case into tarmac file. Some of the failed test need to be run interactively with other hardware build and with different switches.

There are three types of tarmac dumped after the interactively run of failed test cases.

- RTL tarmac dumps the behaviour of tes case on RTL.

- MODEL tarmac dumps the behaviour of test case on MODEL.
- ISSCMP tarmacs shows the difference between RTL and MODEL tarmac.

Sometimes the failed issue is related to MODEL or RTL or printing tarmac issue.After the detection of failed reason we have to file JIRA explaining the reason of failed. Each failed cases has related JIRA that will assign to design team to fix problem.
The main of this AVS regression and verification is to get 100% pass rate which means all test are getting passed.

I made a AVS regression perl script that will apply all the test cases to architecture, generate both .rep report and .csv report and upload final result on val spider.

Chapter 7

Conclusion and Future Scope

7.1 Conclusion

YAML Parser script is used to parse primary id, sub id, test name and context name as required. It is used to create final id that is made of above all values. So Final Id is creted using YAML Parser script.

ELF Parser Program used to read and print all values of Header Table, Section Header Table and Program Header Table of an ELF. It is also used to merge number ELF to create one merged ELF.

Self Modifying Code test are written to check run time behaviour of ELF, Virtual aliasing and cache invalidation.

Regression and verification are the two most part of verifying process of architecture.Regression processor is very time consuming and tidy. A lot manual work one has to do complete whole regression process.

So automation script of regression will save time and manual work. ELF based regression process saves cluster time and cluster size.

7.2 Future Scope

Till now we have reached to overall 97% pass rate. So still there are 3% tests are failing due to some reasons. Debugging of these failed cases and raising JIRAs for them will be next task in verification and regression process.

ELF based regression process saves upto 30% cluster time. Next target is to achieve 50% saving of cluster time by modifying optimized flow.

Some manual work is required in ELF based regression. So There is requirement of a script that will do all steps related to ELF based regression.

References

[1] ARM Info-center, “ARM Architecture Reference Manual for ARMv8 ”,Version May 2013, Available: $http://arminfo.emea.arm.com/$ [Accessed: 2015, June 2]

[2] “ELF-64 Object File Format Manual” Version 1.5,May 1998

[3] ARM Info-center,“v8 VAL User Guide ” Version January 2015, Available: $http://arminfo.emea.arm.com/$ [Accessed: 2015, July 16]

[4] ”YAML::TINY package guide”, Available: $http://www.cpan.org/yaml/$ [Accessed: 2015, July 16]

[5] ARM Info-center, “ARM Architecture Reference Manual for ARMv7-M ”,Version June 2015, Available: $http://arminfo.emea.arm.com/$ [Accessed: 2016, January 16]