AVK RVCT TO LLVM TOOL CHAIN TRANSITION

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering (Embedded Systems)

By

JAYKUMAR P. PATEL
(12MECE36)

Electronics & Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad-382 481

May 2014

AVK RVCT TO LLVM TOOL CHAIN TRANSITION

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

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

By

JAYKUMAR P. PATEL
(12MECE36)

Under the guidance of

External Project Guide:

Mr. Praveen kumar Ramamoorthy
Staff Design Engineer,ATEG
ARM Embedded Technologies Pvt. Ltd.,
Bangalore.

Internal Project Guide:

Dr. Nagendra P. Gajjar
Professor, EC Department,
Institute of Technology,
Nirma University, Ahmedabad.

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2014

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.

- JAYKUMAR P. PATEL

Certificate

This is to certify that the Major Project entitled **"AVK RVCT To LLVM Tool Chain Transition"** submitted by **Jaykumar P. Patel (12MECE36)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Embedded System, Electronics and Communication Engineering (Embedded Systems) of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my 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 my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahmedabad

Dr. N.P. Gajjar
Guide

Dr. N.P. Gajjar
Program Coordinator

Dr. D.K.Kothari
Section Head, EC

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

Dr. K. Kotecha
Director, IT

Certificate

This is to certify that the Project entitled ""**AVK RVCT To LLVM Tool Chain Transition**"" submitted by **Jaykumar P. Patel (12MECE36)**, towards the submission of the Project for requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Mr. Praveen kumar Ramamoorthy
Staff Design Engineer,
Processor Division (Architecture Validation),
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. Nagendra P. Gajjar**, guide and Coordinator of M.Tech Embedded Systems program 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 Company Mentor **Mr. Praveen Kumar Ramamoorthy**, Staff Design 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 am obliged to staff members of Architecture Validation Department, ARM Embedded Technology Pvt. Ltd. for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment.

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

- Jaykumar P. Patel

12MECE36

Abstract

RVCT(Real View Compiler Tool) is a based on arm proprietary compiler technology using armcc for C/C++ compilation. v8AVK(ARMv8 Architecture Validation Kit) is a set of suites containing test-cases which is used to qualify the compliance of any core implementation as defined in the architecture specification. V8AVS(ARMv8 Architecture Validation Suite) product had been compliant and was using RVCT based tool chain for architecture validation. LLVM(Low Level Virtual Machine) is new compiler technology which ARM is adopting for its better configurable features and ease of optimization extensions and wide spread adaptability for being open source based. Under the Architecture validation group at ARM, this project aims at the task to convert the code base of ARMv8 AVK that is only RVCT tool chain compliant into one that supports both LLVM and RVCT compliant. The converted code is to be qualified on the simulator to ensure that the LLVM tool chain had generated the appropriate code and to provide feedback on the tool chain and get associated bugs in tool chain fixed if the code conversion behavior is inconsistent. This project list down the various aspects of this porting work to ensure how this transition path was made smoother to ensure thousands of tests were made compliant across RVCT and LLVM. There are different parts of the tool chain like the linker, compiler and assembler. This project primarily focuses on the compiler code base such that the LLVM support is enabled for AVK.

Abbreviation Notation and Nomenclature

AVK .... Architecture Validation Kit
RVCT .... Real View Compiler Tool
LLVM .... Low level Virtual Machine
ARM ARM .... ARM Architecture Reference Manual
ARMCC .... ARM RVCT Compiler
ARMCLANG .... ARM LLVM Compiler
V8VAL .... ARMv8 Validation Abstraction Layer
ARMASM .... ARM Assembler
ARMLINK .... ARM Linker
ARMAR .... ARM Librarian
FROMELF .... ARM Image Convert Utility
AAPCS .... ARM Application Procedure Call Standard
IR .... Intermediate Representation
RISC .... Reduce instruction Set Computer
BSD .... Berkeley Software Distribution
GPL .... General Public License
GCC .... GNU Compiler Collection
DSB .... Data Synchronous Barrier
DMB .... Data Memory Barrier
ISB .... Instruction Synchronous Barrier
CLZ .... Count Leading Zero
SEV .... Send EVent
SVC .... SuperVisor Call
IRQ .... Interrupt ReQuest
FIQ .... Fast Interrupt reQuest
GIC .... Generic Interrupt Controller

Contents

Declaration iii

Certificate iv

Certificate v

Acknowledgements vi

Abstract vii

Abbreviation Notation and Nomenclature viii

List of Tables xi

List of Figures xii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem definition . . . 2
- 1.3 Thesis Organization . . . 2

2 Brief Review Of Previous Work 4
- 2.1 ARMv8 AVK Overview . . . 4
- 2.2 RVCT Tool Chain . . . 5
- 2.3 LLVM Overview . . . 7
- 2.4 LLVM Tool Chain . . . 10
- 2.5 RVCT vs LLVM . . . 11
- 2.6 Summary . . . 12

3 Implementation Methodology 14
- 3.1 Block Level Design Of The Proposed Approach . . . 14
- 3.2 Proposed Approach . . . 14
- 3.3 Analysis Phase . . . 16
  - 3.3.1 Identification of ARMCC Dependent Features . . . 16
  - 3.3.2 ARMCC Dependent Features List . . . 16

3.3.3 Brief Description Of ARMCC Dependent Features[1][7] 16
3.4 Migration Phase 24
3.4.1 Structural Changes in Code Base 24
3.4.2 LLVM Equivalents List 29
3.4.3 LLVM Equivalents Of ARMCC Command Line Options . . . 29
3.5 Debug Phase 30
3.6 Product Phase 31
3.7 Summary 31

4 Tests Results, Analysis and Profiling 32
4.1 Issues Faced 34
4.2 Debug Progress 35
4.3 Profiling for Performance Improvement 36
4.3.1 Compiler Optimization 36
4.3.2 Profiling On Code size and Simulation Time 38
4.4 Summary 38

5 Conclusion and Future scope of work 40
5.1 Conclusion 40
5.2 Future scope of work 40

List of Tables

I Class-1 Features Equivalents 24
II Assembly Instructions of Class-3 Features 28
III LLVM Equivalents list 29
IV ARMCC Command line options[5][6] 30

List of Figures

2.1 LLVM Compilation Flow[11] 7

3.1 Block Level Design Of The Proposed Approach 15
3.2 ARMCC Dependent List 17
3.3 ASM Conversion 25
3.4 DMB Conversion 27

4.1 Snapshot of First Regression resultsfor group-1 testcases 33
4.2 Snapshot of First Regression results for group-2 testcases 33
4.3 Snapshot of Final Regression results for group-1 testcases 35
4.4 Snapshot of Final Regression results for group-2 testcases 35
4.5 Pass Rate per Week during Debug phase for group-1 36
4.6 Pass Rate per Week during Debug phase for group-2 37

Chapter 1

Introduction

1.1 Motivation

The Processor Division Architecture Validation Department (PDAV) at ARM aims to validate the Architecture and to check for the adherence of the core implementation towards the ARM Architecture. By using Architecture Validation Kit (AVK), AV team validate the features identified by the architecture specifications, clarify any missing / ambiguous statements in ARM(Architecture Reference Manual) manual and ensure that nothing is missed in the manual.The Architecture Validation Kit (AVK) is a product of ARM to ensure compliance of Architectural features. AVK is a set of test suites to check different parts of the ARM architecture. These suites qualify if a specific implementation complies with behavior as specified in ARM ARM. This Kit is used by partners who implement their own ARM complaint processors. Every partner of ARM intending to release ARM compliant chip, needs to ensure that their implementation complies with features specified in ARM Architecture Reference Manual. As the ARM Architecture Reference Manual is just a manual and does not check for compliance and the AVK is released along with this. AVK is a set of test suites to check different parts of the ARM architecture. These suites qualify if a specific implementation complies with behavior as specified in ARM Architecture

Reference Manual.

There is a growing need from partners to support more than one tool chain and their preference to support an open source tool. In order to ensure that the partners can use multiple tool chains, ARM had investigated multiple tool chains and found LLVM to be meritorious due to its flexibility in extension as against other open-source toolchains.

In short, the objective of this project is to release ARMv8 AVK which compatible to both LLVM (Low Level Virtual Machine) -ARM Compiler 6 Tool chain and RVCT (Real View Compiler Tool) -ARM compiler 5 which was only ARMCC compatible. This gives the partners flexibility to adapt to either or both of these tool chains on ARMv8 AVK.

1.2 Problem definition

The objective of this project is to convert the code base of ARMv8 AVK that is only ARMCC compliant into one that supports both LLVM and ARMCC. The converted code is to be qualified on the simulator to ensure that the LLVM tool chain had generated the appropriate code and to provide feedback on the tool chain and get associated bugs in tool chain fixed if the code conversion behaviour is inconsistent.

1.3 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *Brief Review Of Previous Work*, gives overview of ARMv8 AVK, RVCT Tool chain,LLVM Tool chain and Advantages of LLVM over RVCT. applications.

Chapter 3, *Implementation Methodology*, highlights the approach adopted for and the implementation methodology adopted forthe tool chain transition from ARMCC to LLVM .It also throws some light on the ARMCC dependent features with a brief description of each, and changes that were made in the ARMv8 codebase to make it ARMCC independent.

Chapter 4, *Tests Results, Analysis and Profiling*, enumerated the generic results and issues/bugs observed in the analysis, profiling for code size and simulation time improvement and confidentiality issues bar me from providing the whole results and data analysis done.

Finally, in **chapter 5** concluding remarks and scope for future work is presented.

Chapter 2

Brief Review Of Previous Work

This chapter discusses about the ARMv8 Architecture Validation Kit (AVK),RVCT tool chain (ARM Compiler 5) and the open source toolchain LLVM. ARMv8 AVK utilizes RVCT to ensure a complete environment to build and execute the validation suites. RVCT enable us to write and build applications for the ARM family of processors. We can use RVCT to build software programs in C, C++, or ARM assembly language. LLVM tool chain enables the same things. The LLVM Project is a collection of modular and reusable compiler and tool chain technologies. LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects. The clang technology is the one its sub projects apart from llvm-gcc and others. Clang is one the front-end compiler for the LLVM compilation process and has many distinct advantages over the GCC. ARM has used the clang technology of LLVM for its new compiler armclang of ARM compiler 6 also called LLVM tool chain.

2.1 ARMv8 AVK Overview

ARMv8 Architecture Validation Kit (AVK) is a product of ARM to ensure compliance of ARMv8 Architectural features. AVK is a set of test suites to check different

parts of the ARMv8 architecture. The ARMv8 AVK includes validation suites for the ARMv8 architecture as defined by the ARM Architecture Reference Manual. ARMv8 AVK provides a complete environment to build and execute the validation suites. The ARMv8 AVK enables ARMv8 architecture licensees to verify that their implementations are compliant with the architecture as defined by ARM.

The Components of Architecture Validation Kit are:

- **V8VAL**: ARMv8 Validation Abstraction Layer is the software layer on which the AV tests run. V8VAL implemented as a set of standard Application programming Interface (API) and build as a library. This library can be linked to any test case to get the final ELF image used to run on AEM (simulator). VAL provides an abstraction which enables the tests to be independent of target and platform implementation choices. It also provides support for several common features such as boot entry, handler installation, context switching, event generation, and memory management.

- **ARMv8 Test Suites (ARCH64 suites and ARCH32 suites)**: The tests are written to validate an ARM processor in a testbench environment. The tests are written in such way that it transmits test status and pass or fail messages on the terminal.The test suites comprises of AArch64 tests and AArch32 tests. The ARCH64 tests check the compliance with ARMv8 architecture in full 64 bit mode and ARCH32 tests check the compliance with ARMv8 architecture in full 32 bit mode.

2.2 RVCT Tool Chain

RVCT consists of a suite of tools, together with supporting documentation and examples. These tools enable us to write and build applications for the ARM family of processors.

We can use RVCT to build software programs in C, C++, or ARM assembly language.

The components of RVCT consists are[6]:

- **Armcc**:The ARM and Thumb compiler. This compiles your C and C++ code. It supports inline and embedded assembler syntax
- **Armasm**: The ARM and Thumb assembler. This assembles ARM and Thumb assembly language sources.
- **Armlink**: The linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program.
- **Armar**: The librarian. This enables sets of ELF format object files to be collected together and maintained in archives or libraries. You can pass such a library or archive to the linker in place of several ELF files. You can also use the archive for distribution to a third party for further application development.
- **Fromelf**: The image conversion utility. This can also generate textual information about the input image, such as disassembly and its code and data size.
- **C++ libraries**:The ARM C++ libraries provide Helper functions when compiling C++.
- **C libraries**: An implementation of the library features as defined in the C and C++ standards, Extensions specific to the compiler.

ARM Compiler is the term used for the compilation tools produced by ARM. After release of ARM Compiler 4.1, the term was renamed to Real View Compiler Tool(RVCT). The ARM Compiler 5 is the latest version as of this project date. The future release will be ARM COMPILER 6 which be based on LLVM and will be

backward compatible also. So ARM Compiler 6 is the next generation C/C++ compilation toolchain from ARM, based on Clang and the LLVM Compiler framework. Version 6.00 of the toolchain provides architectural support for v8 of the ARM Architecture and alpha support for v7-A. It can be used in conjunction with ARM DS-5 Development Studio to build and debug ARMv8 executable code.

2.3 LLVM Overview

The LLVM Project is a collection of modular and reusable compiler and tool chain technologies. LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "UIUC" BSD-Style license.[2]

LLVM Compilation process:

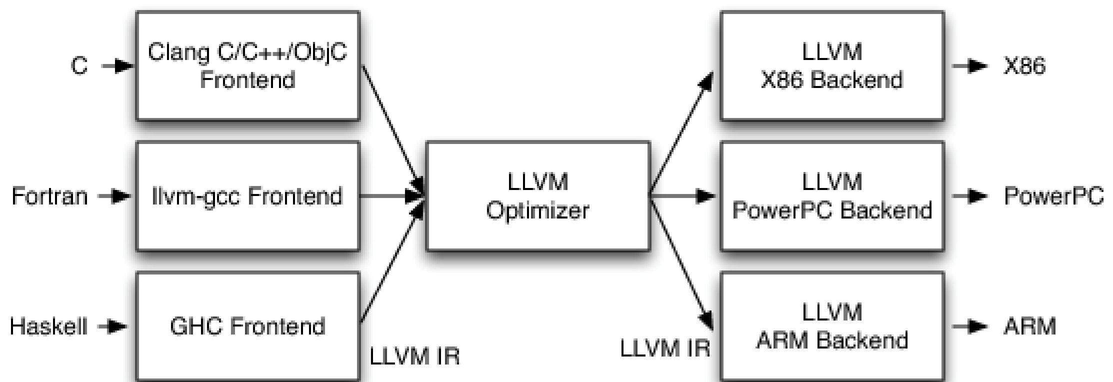

Figure 2.1: LLVM Compilation Flow[11]

The Low Level Virtual Machine (LLVM) is a compiler infrastructure which focuses on compile-time, link-time, and runtime optimization of programs written in multiple

languages.[12] Unlike most other compilers, LLVM compiles source code not directly into binary code, but into intermediate code with its own virtual instruction set. This low level intermediate code looks like a simple 3-address RISC instructions, but provides language independent type system.This intermediate code is in the Static Single Assignment (SSA) form to facilitate optimization. Based on this intermediate code, the LLVM system includes language independent and machine-independent optimization.

LLVM IR instructions are in three address form, which means that they take some number of inputs and produce a result in a different register. (This is in contrast to a two-address instruction set, like X86, which destructively updates an input register, or one-address machines that take one explicit operand and operate on an accumulator or the top of the stack on a stack machine.)

In an LLVM-based compiler, a front end is responsible for parsing, validating and diagnosing errors in the input code, then translating the parsed code into LLVM IR . This IR is optionally fed through a series of analysis and optimization passes which improve the code, then is sent into a code generator to produce native machine code.

LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects. The clang technology is the one its sub projects. The Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles (e.g. about 3x faster than GCC when compiling Objective-C code in a debug configuration), extremely useful error and warning messages and to provide a platform for building great source level tools.[9] The Clang Static Analyzer is a tool that automatically finds bugs in your code, and is a great example of the sort of tool that can be built using the Clang front-end as a library to parse C/C++ code. In short, clang(BSD license) is one the front-end compiler for the LLVM compilation

process apart from llvm-gcc (GPL license) and others. There are many advantages of clang over the GCC.

Pro's of clang vs GCC[8]:

- Design of clang intended to be easily understandable by anyone who is familiar with the languages involved and who has a basic understanding of how a compiler works. GCC has a very old codebase which is difficult for new developers.
- Clang is designed as an API, it allows to be reused in the IDEs (etc) and integrate into other tools. GCC is built as a monolithic static compiler, which makes it extremely difficult to use as an API and integrate into other tools.
- By using GCC, we can't link multiple targets into one binary, while Clang does not have this problem.
- Clang is much faster and uses far less memory than GCC.
- Clang aims to provide extremely clear and concise diagnostics (error and warning messages), and includes support for expressive diagnostics. GCC's warnings are sometimes acceptable, but are often confusing and it does not support expressive diagnostics. Clang also preserves typedefs in diagnostics consistently, showing macro expansions and many other features.
- Clang uses a BSD license, which allows it to be embedded in software and the proprietary commercialization of software containing clang source code. GCC is licensed under the GPL license which allows it to be embedded in software but prevent the proprietary commercialization of open source code.
- Clang inherits a number of features from its use of LLVM as a backend, including support for a bytecode representation for intermediate code, pluggable optimizers, link-time optimization support, Just-In-Time compilation, ability to link in multiple code generators, etc.

Con's of clang vs GCC[8]:

- GCC supports languages that clang do not aim to, such as, FORTRAN, Java, Ada, etc.
- GCC supports more targets than clang.

ARM utilized the clang technology of LLVM for its new release compiler of ARM Compiler 6 tool chain called ARMCLANG.

2.4 LLVM Tool Chain

ARM Compiler 6 enables you to build applications for the ARM family of processors from C, C++, or assembly language source.

The components of LLVM tool chain are[5]:

- **Armclang**: The compiler is based on LLVM and Clang technology. LLVM is a set of open-source components that allow the implementation of optimizing compiler frameworks. Clang is a compiler front end for LLVM, providing support for the C and C++ programming languages.
- **Armasm**: The assembler. This assembles A32, A64, and T32 assembly language sources.
- **Armlink**: The linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program.
- **Armar**: The librarian. This enables sets of ELF object files to be collected together and maintained in archives or libraries. You can pass such a library or archive to the linker in place of several ELF files. You can also use the archive for distribution to a third party for further application development.

- **Fromelf**: The image conversion utility. This can also generate textual information about the input image, such as disassembly and its code and data size.

- **ARM C++ libraries**: The ARM C++ libraries provide helper functions when compiling C++.

- **ARM C libraries**: The ARM C libraries provide:An implementation of the library features as defined in the C and C++ standards and Common nonstandard extensions to many C libraries.

2.5 RVCT vs LLVM

The compiler in ARM Compiler 6 toolchain is armclang, based on Clang, a C/C++ front end for the LLVM code-generation framework. LLVM is designed as a set of reusable libraries with well defined interfaces. In comparison, armcc, the compiler in ARM Compiler 5 is composed of modules with less well defined interfaces and separation, which makes the parts less reusable across a larger code generation problem space. armclang strictly adheres to the three phase LLVM design with a front end parser and syntax checker, a mid end optimizer and code generators that produce native machine code in the backend. The three phases have clear separation in terms of their intended function and this aspect of LLVM makes it reusable and flexible.

LLVM IR, or Intermediate representation, is the glue that connects the three phases. LLVM IR is the only interface to the optimizer and is designed as a first-class language with well defined semantics. LLVM IR was designed from ground up, with supporting common compiler optimizations in mind. The optimizer itself is designed as a set of passes that apply transformations to the input IR to produce IR that feeds into the code generator, which then can produce efficient machine code. The library based design of the optimizer allows the toolchain designers to select passes and optimizations that are most relevant for the given application domain and pro-

duce efficient code with minimum effort.

LLVM framework also makes it easier to add new targets, e.g. by using target description (.td) files to concisely describe a large part of the target architecture in a domain-specific language. Partners also have the option of adding their own backends for custom DSPs or GPUs as plugins into the toolchain. The code generator itself is based on several built-in passes for common problems like instruction selection, register allocation, scheduling etc, so adding new code generators from scratch is relatively easy. The expressiveness of target description syntax is being continuously improved, so that it becomes easier to add targets in the future. The modular design and robust LLVM IR format yields itself well to specialized code generation challenges such as security related extensions sometimes found on embedded micro-controllers .

ARM Compiler 6 comes with optimized libraries and armlink, an industrial strength linker that has been developed over the years as part of the ARM Compiler toolchain and this nicely complements the benefits accrued from LLVM as detailed above. For example, we expect to introduce link time optimization in a future version of the product that would bring together the benefits of LLVM technology for optimization (leveraging the support for bitcode format in LLVM) and the time tested robustness of armlink. When introduced, this would enable optimization across library boundaries which was not possible using older versions of ARM Compiler. By applying ARMs best-in-class embedded libraries, ARM Compiler 6 generates highly optimized library functions tuned specifically for the ARM architecture, improving both performance and power consumption.

2.6 Summary

The chapter discussed about the ARMv8 Architecture Validation Kit (AVK) .The ARMv8 AVK enables ARMv8 architecture licensees to verify that their implemen-

tations are compliant with the architecture as defined by ARM. As of now, ARMv8 AVK utilized RVCT to ensure a complete environment to build and execute the validation suites.RVCT enable us to write and build applications for the ARM family of processors. We can use RVCT to build software programs in C, C++, or ARM assembly language. ARM Compiler is the term used for the compilation tools produced by ARM. This term replaced to RVCT from release 4.1 onwards.The ARM Compiler 5 is the latest version till date. The future release will be ARM COMPILER 6 which be based on LLVM and will be backward compatible also. Clang is one the front-end compiler for the LLVM compilation process and has many distinct advantages over the GCC. ARM utilized the clang technology of LLVM for its new release compiler of ARM Compiler 6 tool chain called ARMCLANG.

Chapter 3

Implementation Methodology

The chapter discusses the approach adopted for and the implementation methodology adopted for the tool chain transition from ARMCC to LLVM. It also throws some light on the ARMCC dependent features with a brief description of each, and changes that were made in the ARMv8 codebase to make it ARMCC independent and how we replaced the ARMCC dependent features in the code-base with the ARMCC independent ones which would well supported by both of the compilers.

3.1 Block Level Design Of The Proposed Approach

The following block diagram depicts the approach that we have proposed and adopted for the tool chain transition to LLVM for ARMv8 AVK.

3.2 Proposed Approach

- **Analysis Phase**: Identification of armcc dependent features. This phase describes how we identified the armcc dependent features in the v8VAL. At the end of this phase, we knew that what were the armcc dependent features and where they were getting used.(Step-1)

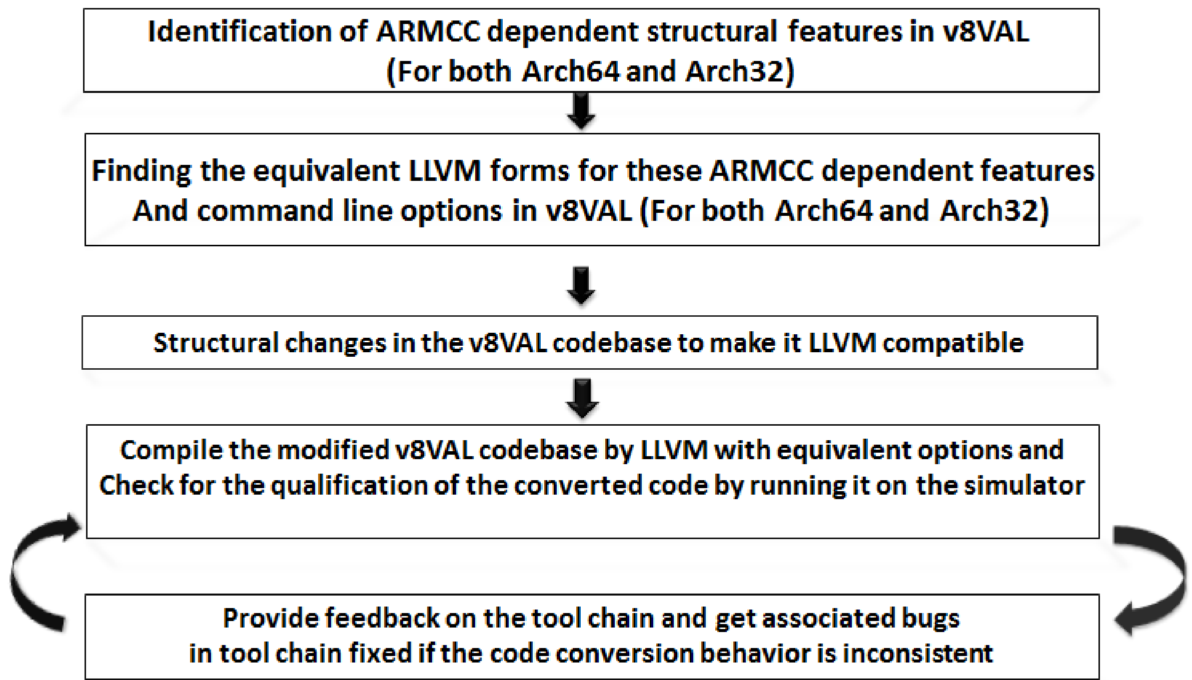

Figure 3.1: Block Level Design Of The Proposed Approach

- **Migration Phase**: Structural changes in v8VAL and ARCH64. This phase describes the conversion of each armcc dependent feature into independent ones and the structural changes made to v8VAL code base.(Step-2 and 3)

- **Debugging Phase**: Check for the qualification of the converted code, fix any inconsistencies observed. After making v8VAL LLVM compatible, the task was to qualify the converted code base on to the simulator. For the qualification, we would compile the modified v8VAL codebase and the tests cases by LLVM with equivalent options and would be running onto the AEM. And we would ensure that the LLVM tool chain had generated the appropriate code and provide feedback on the tool chain and get associated bugs in tool chain fixed if the code conversion behaviour is inconsistent.(Step-4 and 5)

- **Product Phase**:Once everything is observed stable, the next task is to prepare v8VAL and ARCH64 as product such that it could be released to partner with

LLVM toolchain to validate their ARM compliance implementation.

3.3 Analysis Phase

3.3.1 Identification of ARMCC Dependent Features

There are certain features in the v8VAL codebase that are compatible only with ARMCC compiler. They begin with double underscore. To identify these features, a tool called compatibility checker is used. The ARM Compiler Source Compatibility Checker examines C or C++ source code and highlights language extensions that were supported by previous versions of ARM Compilers but are no longer supported by ARM Compiler 6. Compatibility checker when run on the c code it throws warning when incompatibilities are identified.

For example: **Warning: armcc extension ' __asm'**
For implementing this, Make-files for v8VAL libraries were edited so that while running a v8 test, it would also run the compatibility checker for all the files. The result was stored in a log file. A Perl script was written that could extract those warnings for armcc extensions along with the file names which contained those features from that log file. This gave us all the armcc dependent features, all the concerned file names along with their path and the lines in which that feature has been used in the code.

3.3.2 ARMCC Dependent Features List

The following ARMCC dependent features were listed and we categorized them into following four classes:

3.3.3 Brief Description Of ARMCC Dependent Features[1][7]

a. **__weak**

Class1	Class 2	Class 3	Class-4
• __weak • __align • __packed	• __dsb • __dmb • __isb • __clz • __sev • __wfe • __strex • __ldrex • __rev	• __asm	• __TARGET_ARCH_AARCH64 • __TARGET_ARCH_AARCH32 • __BIG_ENDIAN

Figure 3.2: ARMCC Dependent List

The __weak keyword instructs the compiler to export symbols weakly. The __weak keyword can be applied to function and variable declarations, and function definitions. For declarations, this storage class specifies an extern object declaration that, even if not present, does not cause the linker to fault an unresolved reference. **For example**: __weak void f(void); ... f(); // call f weakly

If the reference to a missing weak function is made from code that compiles to a branch or branch link instruction, then either: The reference is resolved as branching to the next instruction. This effectively makes the branch a NOP. The branch is replaced by a NOP instruction.

b. **__align**

The __align keyword instructs the compiler to align a variable on an n-byte boundary. __align is a storage class modifier. It does not affect the type of the function.__align(n) is useful when the normal alignment of the variable being declared is less than n.Eight-byte alignment can give a significant performance advantage with VFP instructions.

c. **__packed**

The __packed qualifier sets the alignment of any valid type to 1. This means that there is no padding inserted to align the packed object and objects of packed type are read or written using unaligned accesses. The __packed qualifier applies to all members of a structure or union when it is declared using__packed. There is no padding between members, or at the end of the structure. All substructures of a packed structure must be declared using __packed. Integral sub fields of an unpacked structure can be packed individually.

The __packed qualifier is useful to map a structure to an external data structure, or for accessing unaligned data, but it is generally not useful to save data size because of the relatively high cost of unaligned access. Only packing fields in a structure that requires packing can reduce the number of unaligned accesses.

d. **__svc**

The __svc keyword declares a SuperVisor Call (SVC) function taking up to four integer-like arguments and returning up to four results in a value_in_regs structure.The __svc causes an exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SVC vector. Supervisor calls are normally used to request privileged operations or access to system resources from an operating system.

e. **__asm**

This keyword passes information from the compiler to the ARM assembler armasm. The precise action of this keyword depends on its usage.

- **Embedded Assembler**:The __asm keyword can declare or define an embedded assembly function. Functions declared with __asm or asm can have arguments, and return a type. They are called from C and C++ in the same way as normal C and C++ functions.

  For example:

```
__asmreturn-type function-name(parameter-list)
    {
    // ARM/Thumb assembler code
    instruction;comment is optional
    ...
    instruction
}
```

- **Inline Assembler**:The __asm keyword can incorporate inline assembly into a function.

  For example:

```
int qadd(inti, int j)
    {
    int res;
    __asm
    {
        ADD res, i, j
    }
    return res;
}
```

- **Assembler Labels**:The __asm keyword can specify an assembler label for a C symbol. Assembler labels specify the assembler name to use for a C symbol. For example, you might have assembler code and C code that uses the same symbol name, such as counter. Therefore, you can export a different name to be used by the assembler

  For example:

```
int count __asm__("count_v1"); // export count_v1, not count
```

- **Named Register Variables**: The __asm keyword can declare a named register variable. The compiler enables you to access registers of an ARM architecture-based processor or coprocessor using named register variables
  For example:
  register int foo __asm("r0");

f. **__dmb**

It generates a DMB (data memory barrier) instruction or equivalent CP15 instruction. The DMB instruction ensures that all explicit data memory transfers before the DMB are completed before any subsequent explicit data memory transactions after the DMB starts. This ensures correct ordering between two memory accesses. DMB ensures the observed ordering of memory accesses. Memory accesses of the specified type issued before the DMB are guaranteed to be observed (in the specified scope) before memory accesses issued after the DMB. For example, DMB should be used between storing data, and updating a flag variable that makes that data available to another core.

g. **__dsb**

The DSB instruction ensures all explicit data transfers before the DSB are complete before any instruction after the DSB is executed.It generates a DSB (data synchronization barrier) instruction or equivalent CP15 instruction. DSB ensures the completion of memory accesses. A DSB behaves as the equivalent DMB and has additional properties. After a DSB instruction completes, all memory accesses of the specified type issued before the DSB are guaranteed to have completed.

h. **__isb**

It generates an ISB (instruction synchronization barrier) instruction or equivalent CP15 instruction. This instruction flushes the processor pipeline fetch

buffers, so that following instructions are fetched from cache or memory. An ISB is needed after some system maintenance operations. An ISB is also needed before transferring control to code that has been loaded or modified in memory, for example by an overlay mechanism or just-in-time code generator. The ISB instruction flushes the pipeline in Cortex-M processors and ensures effects of all context altering operations prior to the ISB are recognized by subsequent operations. It should be used after the CONTROL register is updated.

i. **__sev**

It generates a SEV (send event) instruction. This causes an event to be signaled to all processors in a multiprocessor system. It is a NOP on a uniprocessor system. If SEV is implemented, WFE must also be implemented.

j. **__wfe**

It generates a WFE (wait for event) hint instruction, or nothing. The WFE instruction allows (but does not require) the processor to enter a low-power state If the Event Register is not set, WFE suspends execution until one of the following events occurs:

- an IRQ interrupt, unless masked by the CPSR I-bit
- an FIQ interrupt, unless masked by the CPSR F-bit
- an Imprecise Data abort, unless masked by the CPSR A-bit
- a Debug Entry request, if Debug is enabled
- an Event signaled by another processor using the SEV instruction

k. **__rev**

This intrinsic inserts a REV instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to convert a 32-bit big-endian data value into a little-endian data value, or a 32-bit little-endian data value into a big-endian data value from within your C or C++ code.

l. **__clz**

This intrinsic inserts a CLZ instruction or an equivalent code sequence into the instruction streamgenerated by the compiler. It enables you to count the number of leading zeros of a data valuein your C or C++ code.

For example:

unsigned char __clz(unsigned intval)

The __clz intrinsic returns the number of leading zeros in val.

m. **__ldrex- Load Register Exclusive**

This intrinsic inserts an instruction of the form LDREX[size] into the instruction stream generatedby the compiler. It enables you to load data from memory in your C or C++ code using an LDREX instruction. size in LDREX[size] is B for byte stores or H for halfword stores. If no size is specified,word stores are performed. Load-exclusive performs a load from memory and causes the physical address of the access to be tagged as exclusive-access for the requesting processor. This causes any other physical address that has been tagged by the requesting processor to no longer be tagged as exclusive-access.

For Example:

unsigned int __ldrex(volatile void *ptr)

Where:

- ptr points to the address of the data to be loaded from memory. To specify the type of the data to be loaded, cast the parameter to an appropriate pointer type.
- The __ldrex intrinsic returns the data loaded from the memory address pointed to by ptr

n. **__strex - Store-Exclusive Instruction**

Store-exclusive performs a conditional store to memory. The store only takes place if the physical address is tagged as exclusive-access for the requesting processor. This intrinsic inserts an instruction of the form STREX[size] into the instruction stream generated by the compiler. It enables you to use an STREX instruction in your C or C++ code to store data to memory.

int __strex(unsigned int val, volatile void *ptr)

Where:

- val is the value to be written to memory.
- ptr points to the address of the data to be written to in memory. To specify the size of the data to be written, cast the parameter to an appropriate integral type.

The __strex intrinsic returns:

- 0 if the STREX instruction succeeds.
- 1 if the STREX instruction is locked out.

o. **__BIGENDIAN**

The predefined macros of the ARM compiler for c and c++ where the value field is empty, the symbol is only defined. It considers to be true, If compiling for a big-endian target.

p. **__TARGET_ARCH_AARCH32**

The predefined macros of the ARM compiler for c and c++ where the value field is empty, the symbol is only defined. It considers to be true, When the target is in AArch32 state.

q. **__TARGET_ARCH_AARCH64**

The predefined macros of the ARM compiler for c and c++ where the value field is empty, the symbol is only defined. It considers to be true, When the target is in AArch64 state.

ARMCC Dependent Feature	LLVM Compatible
__weak	__attribute__((weak))
__packed	__attribute__((packed))
__align	__attribute__((aligned))

Table I: Class-1 Features Equivalents

3.4 Migration Phase

3.4.1 Structural Changes in Code Base

The third step of the proposed approach adopted for the tool chain transition to LLVM for ARMv8 AVK is the Conversion of ARMCC dependent features to ARMCC independent and base on derived features the structural changes in the v8VAL codebase would be made to make it LLVM compatible.
The conversion made as per the class:

- **Class-1 Features**:

  The equivalents for the class-1 were provided by the ACLE manual[3].This document specifies the ARM C Language Extensions to enable C/C++ programmers to exploit the ARM architecture with minimal restrictions on source code portability. The LLVM equivalents are listed in the table I.

- **Class-2 Features**

  For __asm feature which was most frequently used, different approaches were adopted for its conversion according to its usage pattern.The __asm keyword has been extensively used in the v8VAL libraries. So the above modification was done using some perl-scripts.

  When __asm used as Embedded assembler:

  Lets say xyz function declared with the __asm in the val_utils.c as follows: __asm void xyz (unit32_t)

  {

```
        MOV R1,R0
        MOV LR, R1
    }
```

Then this function was removed from the C code and the declaration written into in the header file Val_utils_func.h as void xyz (unit32_t); And the definition of the function was written into the assembly file Val_utils_asmfunc.s. The whole process depicted in the following figure:

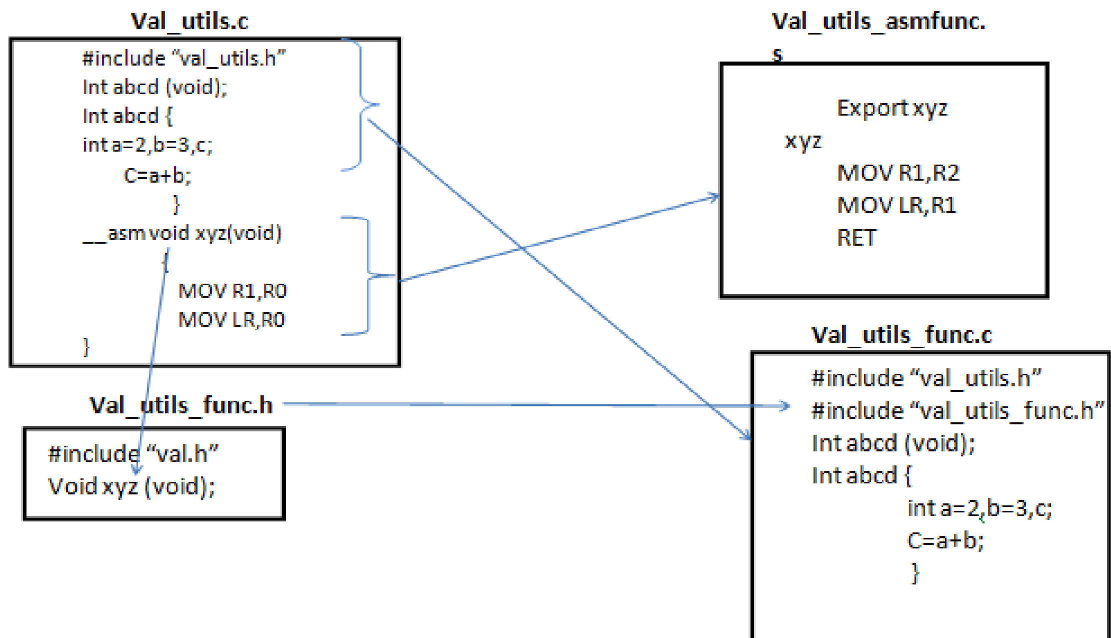

Figure 3.3: ASM Conversion

When used as Inline assembler:

```
__asm {
        MRS cpsr_val,cpsr;
    }
```

This was removed from the C code and declared as some function(say void read_cpsr(void))in the header file. And the definition was written into the

assembly code.

When used as Named register variables:

```
register unsigned int hdfar_reg __asm("cp15:4:c6:c0:0");
used in
unsigned val_read_hdfar_reg(void)
    {
        register unsigned inthdfar_reg __asm("cp15:4:c6:c0:0");
        return hdfar_reg;
    }
```

This was modified as

```
unsigned val_read_hdfar_reg(void)
    {
        unsigned int hdfar_reg;
        hdfar_reg = read_hdfar_reg();
        return hdfar_reg;
    }
```

The following function was declared in the header file:

unsigned int read_hdfar_reg(void);

Equivalent assembly code written for this:

```
        EXPORT read_hdfar_reg
read_hdfar_reg
        MRC p15,4,r0,c6,c0,0
        BX LR
```

- **Class-3 Features**

  These features were removed from the C code and their equivalent assembly in-

structions were written into assembly files. They can be invoked by a function call from other C/assembly files. And those functions (symbols in assembly) were made visible to other files by mentioning export before their names in the assembly code .Also those functions were declared in a separate header file which was then included in the C file. The whole process is depicted in the figure DMB Conversion3.4 and the assembly instruction of each feature are listed in the tableII.

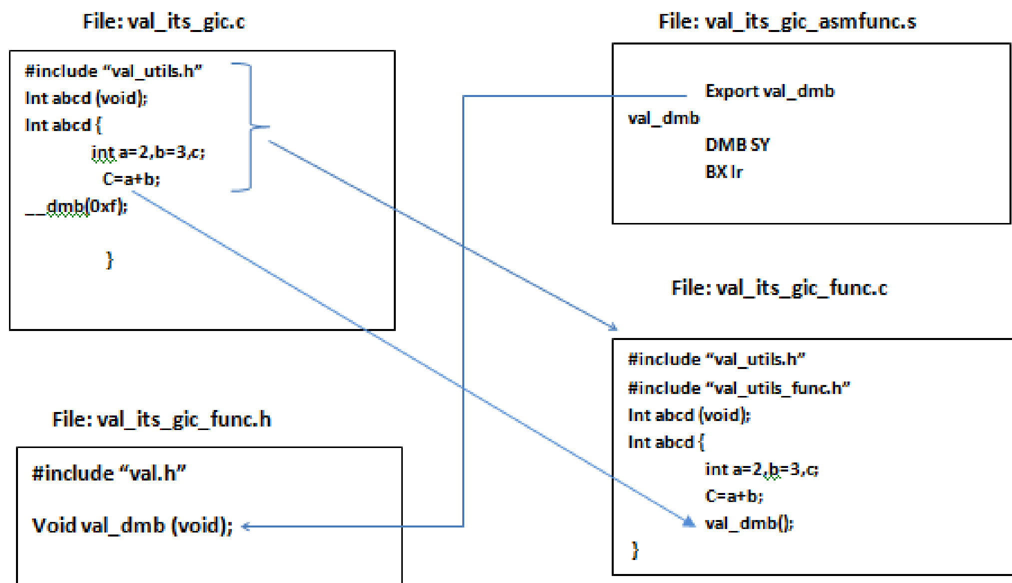

Figure 3.4: DMB Conversion

The __dmb functionality can be implemented by invoking val_dmb function which executes DMB instruction.

- **Class-4 Features**

  The class four was the predefined macros of the ARM compiler for c and c++ where the value field is empty, the symbol is only defined.

  Under the__ARMV8CC_VERSION (always defined for ARM Compiler),We have

ARMCC Dependent Feature	**Assembly Instruction**
__dmb	DMB
__dsb	DSB
__isb	ISB
__rev	REV
__wfe	WFE
__sev	SEV
__svc	SVC
__clz	CLZ
__ldrex	LDREX/LDXR
__strex	STERX/STXR

Table II: Assembly Instructions of Class-3 Features

defined other macros for theses predefined macros of ARM compiler in the one of the file of v8VAL as follows:

```
#ifdef __ARMV8CC_VERSION
#ifdef __TARGET_ARCH_AARCH64
#define COMPILER_AARCH64
#endif
#ifdef __TARGET_ARCH_AARCH32
#define COMPILER_AARCH32
#endif
#ifdef __BIG_ENDIAN
#define COMPILER_BIGENDIAN
#endif
#endif
```

Replaced __BIG_ENDIAN by COMPILER_BIGENDIAN , __TARGET_ARCH_AARCH32 by COMPILER_AARCH32 and __TARGET_ARCH_AARCH64 by COMPILER_AARCH64 in the files where it contained.

ARMCC Dependent Feature	LLVM Compatible
__weak	__attribute__((weak))
__packed	__attribute__((packed))
__align	__attribute__((aligned))
__dmb	DMB
__dsb	DSB
__isb	ISB
__rev	REV
__wfe	WFE
__sev	SEV
__svc	SVC
__clz	CLZ
__ldrex	LDREX/LDXR
__strex	STERX/STXR
__TARGET_ARCH_AARCH64	COMPILER_AARCH64
__TARGET_ARCH_AARCH32	COMPILER_AARCH32
__BIG_ENDIAN	COMPILER_BIGENDIAN

Table III: LLVM Equivalents list

3.4.2 LLVM Equivalents List

All the occurrences of ARMCC dependent features were replaced with the LLVM equivalents in the v8VAL by corresponding name as shown in the table namesIII. This would automated by the use of perl scripts. By removing all armcc dependent features from the v8VAL code base and replacing it with the LLVM equivalents(Table-III) that would make the v8VAL codebase LLVM compatible. This was the third step of proposed Approach.

3.4.3 LLVM Equivalents Of ARMCC Command Line Options

Perl scripts were developed to collect the ARMCC command line options used to compile the v8VAL files. The llvm equivalents of each command line options are described in the tableIV. This LLVM options were passed to the armclang compiler for the compilation of the v8val codebase.

ARMCC Options	**LLVM Options**
–fpu=none	-mfpu=none
–cpu=8-A.64	-target aarch64-arm-none-eabi
–cpu=8-A.32	-target armv8-arm-none-eabi
-c	-c
-o	-o
-D	-D
-I	-I
–bigend	-
–no_unaligned_access	-munaligned-access
–split_sections	-ffunction-sections
–thumb	-mthumb

Table IV: ARMCC Command line options[5][6]

ARM v8AVK has been supported only by armcc till now. The proposed approach aims to make the AVK compatible to armclang which is based on an open source LLVM compiler and also make it backward compatible with the armcc compiler. By replacing the armcc dependent features from the codebase with the armcc independent ones, it will be well supported by both of the compilers.

3.5 Debug Phase

In this Phase, the qualification of the converted code was checked, fixed inconsistencies observed. After making v8VAL code base LLVM compatible, the task was to qualify the converted code base on to the simulator. For the qualification, the modified v8VAL codebase and the tests cases compiled by LLVM with equivalent options and would be running onto the AEM. Meanwhile ensured that the LLVM tool chain had generated the appropriate code and provide feedback on the tool chain and get associated bugs in tool chain fixed if the code conversion behaviour is inconsistent. For qualification, the regression for CORE, DEBUG, GIC, TIMER and ENDIAN and MEM Tests Cases with the converted v8VAL code base were ran and were debugged the failed tests cases. This is the longest phase of the proposed approach because it required architecture knowledge, validation environment to be known and also require

some debugging skill to be developed. During debug phase, the found bugs related to testcases and toolchain should be captured and reported to the concern team.

3.6 Product Phase

Once it is observed that v8VAL and ARCH64 are stable, the next task is to merge the v8VAL with the mainline v8VAL through svn repository, help AVS team member to resolve the tests bug found in the ARCH64 suite and necessary scripts modification for the automation of the Validation flow. There were around 5000 testcases and three toolchain associated bugs were found. As everything was well in timed and placed, the AVK(v8VAL+ARCH64) is released to partner on the concern date.

3.7 Summary

The chapter discussed the approach adopted for and the implementation methodology adopted for the tool chain transition from ARMCC to LLVM .It also throws some light on the ARMCC dependent features with a brief description of each, and changes that were made in the ARMv8 codebase to make it ARMCC independent.By replacing the ARMCC dependent features in the code-base with the ARMCC independent ones, it would well supported by both of the compilers.

Chapter 4

Tests Results, Analysis and Profiling

After making v8VAL LLVM compatible, the task was to qualify the converted code base on to the simulator. The simulator was the V8 Architecture Envelope Model executed using load sharing mechanism. For the qualification, the modified v8VAL codebase and the testcases were compiled by LLVM with equivalent options and ran onto the AEM. The testcases belonged to CORE, DEBUG, GIC, TIMER, ENDIAN and MEM verification and all together called ARCH64 suite. The CORE suite test architecturally-invariant behavior of the Integer instruction set, Floating point instruction set, SIMD instruction etc. The DEBUG suite verifies the debug architecture defined in V8 reference manual. GIC is generic interrupt controller architecture for the ARM processor and GIC suite is written to validated the interrupt controller architecture. Timer suite verifies the interrupt timer registers and ENDIAN suite verifies the bit representation format - little and big endian. And the MEM suite verifies the basic memory system features page table walk, virtualization etc. Due to confidentiality concerns, it was restricted to put more details about the tests suites and the result achieved.

```
*********************************************
TOTAL TESTS         :       15931
TOTAL PASSED        :       10907    ( 68.4%)
TOTAL FAILED        :        1890    ( 11.8%)
TOTAL ABANDONED     :        2757    ( 17.3%)
TOTAL NOT RUN       :         0
TOTAL COMPILED OK   :         0
TOTAL COMPILE FAIL  :         377    ( 2.3%)
TOTAL SIM TIME      :         0.21 hours
*********************************************
```

Figure 4.1: Snapshot of First Regression resultsfor group-1 testcases

```
*********************************************
TOTAL TESTS         :        4829
TOTAL PASSED        :        4209    (87.16%)
TOTAL FAILED        :         222    (4.59%)
TOTAL ABANDONED     :         398    (8.24%)
TOTAL NOT RUN       :           0
TOTAL COMPILED OK   :           0
TOTAL COMPILE FAIL  :           0
TOTAL SIM TIME      :        0.11 hours
*********************************************
```

Figure 4.2: Snapshot of First Regression results for group-2 testcases

The whole ARCH64 Suite was divided into two group : 1. CORE, DEBUG and MEM suites 2. GIC, TIMER and ENDIAN. The reason behind division was GIC and TIMER contains totally C-based testcases and big-endian support for C- compiler was not available earlier by LLVM Team. So it had been decided to focus first on Group-1 testcases for debugging as they all are assembly based and easy to debug. And the failures debugging for group-2 testcase were started after bigendian support was availabled by LLVM Team for C-compiler. The figure 4.1 shows the snap-shot of the regression run with group-1 Tests Cases. There was 68.4% pass rate for LLVM and 84 % pass rate for ARMCC on the first regression for modified code base. Regression for ARMCC was ran because, as aim was to make c- code base compatible for both compilers. The expected pass rate is to be 95 to 100% and our job was to analyse reason behind failed, abandoned and compile-failed tests and solving them. The results for group-2 is mentioned in the figure 4.2 having 87% pass rate on LLVM and

92% pass rate on ARMCC. While debugging failed and abandoned testcases, bugs mentioned in following section were observed.

4.1 Issues Faced

There were many issues encountered during debug phase but only major three issues/bugs faced and detailed below:

- **The LLVM tool chain was generating the floating point instruction for non-float variables and function.**
  The compiler was not supposed to generate the floating point instruction for non-float variables. The Solution would be provided from the compiler team by providing the option to compile such it does not generate FP instruction automatically. This change in compiler would solved 100 test cases.

- **LLVM Compiler was not generating the atomic instruction for loading and storing of 64 bit Trick-box registers.**
  The trick-box registers designed to collect information from the processor for architecture validation. These registers are nothing but memory location. And it is mandatory to load and store these register in atomic fashion. The compiler supposed to generate the atomic instruction for loading and storing of 64 bit Trick-box registers. For the solution, Functions were wrote to load and store the Trick-box registers in atomic fashion. By implementing these function, there were 700 tests passed.

- **Tests cases violating the AAPCS rule.**
  The ARM Compiler enables us to call C and assembly language code from C++, and to call C++ code from C and assembly language. But to do that AAPCS rule must be followed. The ARM Procedure Call Standard (APCS) is a set of rules which regulate and facilitate calls between separately compiled or assembled program fragments. AAPCS tells which registers to be used for

argument passing and return when we are calling any c-function from assembly. Some of the tests cases were violating this rule and resulted into fails. The testcases were modified to follow the AAPCS and they resulted into pass(3000 tests).

After resolving above issue, the pass rate for group-1 and group-2 was almost 100 % on LLVM and ARMCC both. Figure 4.3 and 4.4 shows the status of the final regression results.

```
*********************************************
TOTAL TESTS        :       15931
TOTAL PASSED       :       15931     (100%)
TOTAL FAILED       :           0     ( 0%)
TOTAL ABANDONED    :           0     ( 0%)
TOTAL NOT RUN      :           0
TOTAL COMPILED OK  :           0
TOTAL COMPILE FAIL :           0
TOTAL SIM TIME     :        0.19 hours
*********************************************
```

Figure 4.3: Snapshot of Final Regression results for group-1 testcases

```
*********************************************
TOTAL TESTS        :        4829
TOTAL PASSED       :        4829     (100%)
TOTAL FAILED       :           0     ( 0%)
TOTAL ABANDONED    :           0     ( 0%)
TOTAL NOT RUN      :           0
TOTAL COMPILED OK  :           0
TOTAL COMPILE FAIL :           0
TOTAL SIM TIME     :         0.9 hours
*********************************************
```

Figure 4.4: Snapshot of Final Regression results for group-2 testcases

4.2 Debug Progress

The following chart shows the progress -pass rate per week while debug phase. The data was collected by solving the errors/issues faced in the tests at biweekly. It was

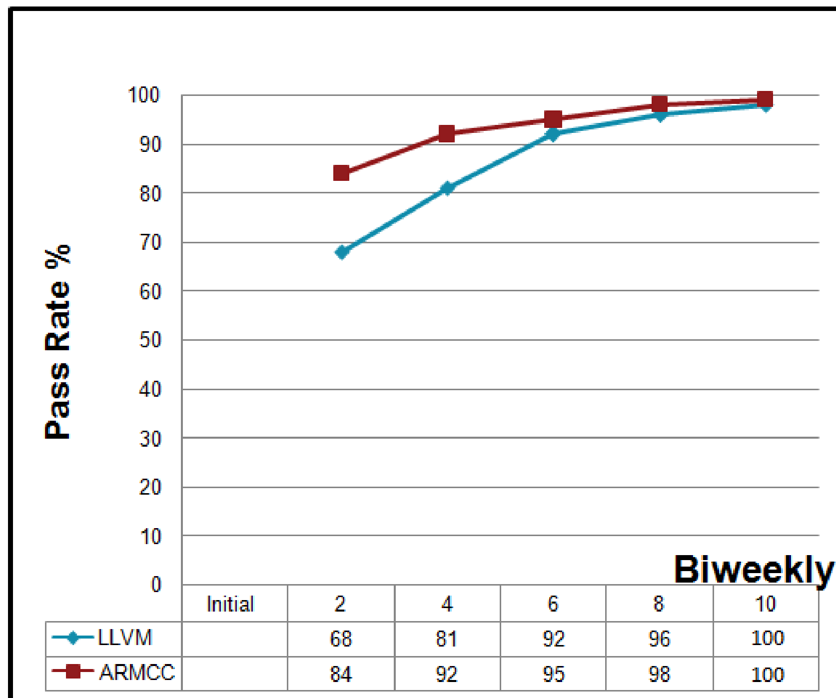

Figure 4.5: Pass Rate per Week during Debug phase for group-1

approximately 10 week and 5 week required to achieve almost 100 % pass rate group-1 and group-2 respectively on LLVM as shown in figure 4.5 and 4.6.

4.3 Profiling for Performance Improvement

The code generation by LLVM compiler was more than the C-compiler from RVCT. And it was due to different code optimisation level used by LLVM compiler and RVCT compiler. Because of that test simulation might have more number of clock cycles and more code size especially in boot-code. As result the task was carried to perform profiling on boot-code with respect to code size and simulation time.

4.3.1 Compiler Optimization

The precise optimizations performed by the compiler depend both on the level of optimization chosen, and whether you are optimizing for performance or code size.

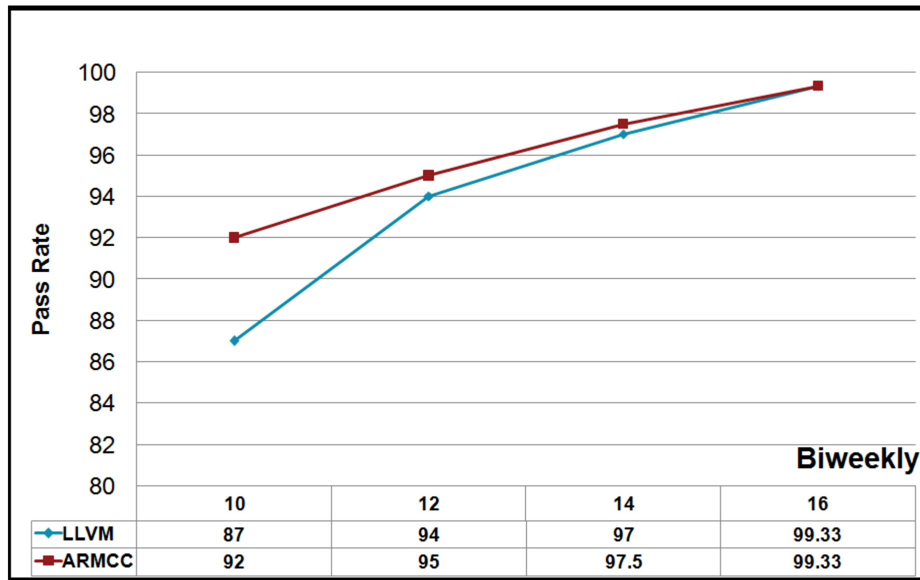

Figure 4.6: Pass Rate per Week during Debug phase for group-2

The armclang and armcc compilers supports the following optimization levels:

- -O0

  Minimum optimization. The compiler performs simple optimizations that do not impair the debug view. When debugging is enabled, this option gives the best possible debug view. This is the default optimization level for armclang.

- -O1

  Restricted optimization. When debugging is enabled, this option gives a generally satisfactory debug view with good code density.

- -O2

  High optimization. This is the default optimization level for armcc. When debugging is enabled, this option might give a less satisfactory debug view.

- -O3

  Maximum optimization. This is the most aggressive form of optimization available. Specifying this option enables multi file compilation by default where multiple files are specified on the command line.When debugging is enabled,

this option typically gives a poor debug view.

4.3.2 Profiling On Code size and Simulation Time

For the ease debug phase, -O0 optimisation was chosen for the armclang because it enabled satisfactory debug view but at the expense more code size. As armclang generated more code due to -O0 optimization is chosen, a Profile activity for code size and simulation time has been carried out to improve the performance of the testcases. Due to confidentiality concerns, it was restricted to put whole details about the profiled data and hence only results were mentioned:

- Simulation time Reduction

  Profiled the data related to Bootup code of 10k tests cases for both ARMCC and LLVM Compiler. The profiling of data contained the number of instructions related to memory(Load and store), memory barrier, cache maintenance and branch instruction. By analyse the collected data, the conclusion was made that there is need to modify bootup so that they can have less number of memory related instruction.

- Code Size Reduction.

  Analysed the Print functions effect on the test code size. Print functions are used print part status, test status, test failure and other important message. The print related functions were taking 20 to 30 % instruction of the tests code. Need to modify print functions so that it can take lesser code size without breaking their functionally.

4.4 Summary

Qualified the converted code of v8VAL by running the regression for CORE, DEBUG, GIC, TIMER, ENDIAN and MEM Tests Cases with 100% pass rate on LLVM and ARMCC both and reported the bugs faced to validation team and the LLVM toolchain

team and helped them to resolve the bugs. Analysed the effect of print function on total instruction count of the tests to improve the code size and the effect memory related instruction in the bootup code to improve the simulation time.

Chapter 5

Conclusion and Future scope of work

5.1 Conclusion

The code base of ARMv8 AVK which only ARMCC compliant was successfully converted into one that supports both LLVM and RVCT (ARMCC) for CORE, DEBUG, GIC, TIMER, ENDIAN and MEM suites(Approx. 18000 testcases). And concurrently ensured that the LLVM tool chain generates the appropriate code and provided feedback on the tool chain based on bug faced. From now, ARM partners have flexibility to adapt to either or both of these tool chains on ARMv8 AVK.

5.2 Future scope of work

- C- code base is now compatible to LLVM Tool Chain. The future task is to make AVK assembly code base to LLVM compatible.
- Qualify the migrated C-code base with LLVM -O3 optimisation level to improve code size and simulation time.

- Modify bootup code such that it can have less number of memory related instruction.
- Modify print functions such that it can take lesser code size without breaking their functionally.

References

[1] ARM,"ARMCompilerReferenceManualandARMAssemblerReference Manual"Website, July2013, http://arminfo.emea.arm.com

[2] LLVM,"TheLLVMCompilerInfrastructure"Website, July2013, http://llvm.org/

[3] ARM,"ARMCLanguageExtensions(ACLE)Manual"Website, Aug2013, http://arminfo.emea.arm.com

[4] RandalL.schwartz, briandfoyandTomphoenix,"LearningPerl", O'Reilly, 6thedition, 2011

[5] ARM,"ARMCompiler6ReferenceManual"Website, Oct2013, http://arminfo.emea.arm.com

[6] ARM,"ARMCompiler5ReferenceManual"Website, Sep2013, http://arminfo.emea.arm.com

[7] ARM,"ARMv8ArchitectureReferenceManual"Website, Sep2013, http://arminfo.emea.arm.com

[8] LLVM,"ClangvsGCC(GNUCompilerCollection)"Website, Oct2013, http://clang.llvm.org/comparison.html

[9] Clang,"Clang − Features, Goals, InternalDesignandImplementation" Website, Oct2013, http://clang.llvm.org/features.html

[10] Clang,"ExpressiveDiagnostics"Website, Oct2013, http://clang.llvm.org/diagnostics.html

[11] ChrisLattner,"TheArchitectureofOpenSourceApplications" Website, Oct2013, http://www.aosabook.org/en/llvm.html

[12] Jae − JinKim, Seok − YoungLee, Soo − MookMoonandSuhyunKim, "ComparisonofLLVMandGCContheARMPlatform", IEEEConference, 2010