UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DESIGN ENHANCEMENTS AND A VALIDATION FRAMEWORK FOR ARM EMULATOR

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Himabindu Thota

June 2013

The Thesis of Himabindu Thota
is approved:

Professor Jose Renau, Chair

Professor Matthew Guthaus

Professor Anujan Varma

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

DESIGN ENHANCEMENTS AND A VALIDATION FRAMEWORK FOR ARM EMULATOR

by

Himabindu Thota

Processor emulators allow micro-architecture researchers to evaluate research ideas quickly and at no extra cost. Micro-architecture Santa Cruz (MASC) Laboratory is developing an ARM processor emulator to execute ARM binaries. To ensure correctness of the emulator, it is expected that at the end of every instruction execution the modified registers, their contents, condition codes and any modified values in memory match that of the program execution on a computer with ARM processor. To this end, this work has three parts - first, to design and implement a validation infrastructure for the emulator ISA and the second, to improve the emulator functionality. Interestingly, these two goals go hand-in-hand. Having a validation infrastructure exposes incorrectly implemented or not yet implemented instructions while certain instructions prompt design enhancements in the validation infrastructure. Furthermore, we developed a methodology to generate Random Instruction Tests (RIT) to facilitate regression testing of the ARM emulator.

Acknowledgments

Most importantly, I would like to thank my advisor Professor Jose Renau for his invaluable guidance and academic support. I am grateful for the enriching experience working at the MASC lab.

I would also like to thank my committee members Professor Matthew Guthaus and Professor Anujan Varma for their time reviewing my thesis and providing valuable feedback.

I must give special thanks to Ehsan K. Ardestani for engaging in long and productive design discussions, and all my friends and colleagues at the MASC lab.

More personally I would like to thank my husband Sai for his immense patience, support and his constant encouragement to do my best, and my son Hrishikesh and daughter Anagha for always cheering me.

Furthermore, I would like to thank my parents, in-laws, brother, sister and all my friends who have positively influenced my life.

Chapter 1

Introduction

1.1 Motivation

Ever-increasing complexity of microprocessor design with many functional units and complex Instruction Set Architectures (ISA) has prompted the development of high quality software processor emulators to validate design ideas and performance optimizations before the actual microprocessor is fabricated. Even on an in-production microprocessor, evaluating design changes in real hardware is infeasible, as it not only is expensive but is also time-consuming.

Designers and researchers often develop and use microprocessor emulators to evaluate research ideas [7]. Emulators provide a software platform, where functionality can be added and modified with ease and, design choices and optimizations can be evaluated within a reasonable amout of time at no extra cost.

The goal of this work is to develop a high quality ARM processer emulator for use in the Micro-Architecture Santa Cruz (MASC) Research lab. At the beginning of this work, the MASC ARM emulator was able to decode and execute most of the ARM and Thumb-2 instructions. However, the emulator was not ready to execute a complete ARM binary as some of the functions, such as program loading and handling some of the system calls, were not available in the emulator. In addition, not all the implemented instructions were producing the correct results and some instructions were not implemented.

Consequently, as part of this work, first we designed and developed a validation framework to identify the missing emulator functions, incorrectly working and not implemented instructions. Second, by developing the missing emulator functions, fixing incorrectly working

instructions and implementing not implemented instructions, we are able to execute complete ARM binaries through the emulator. Furthermore, we developed a methodology to generate test assembly programs with random instruction sequences to facilitate regression testing of the emulator.

The overall structure of the thesis is as follows. Chapter 2 details the design and implementation of the validation framework. Chapter 3 describes the implemented emulator functionalities. Chapter 4 presents the approach for generating regression tests using random instruction sequences and finally, Chapter 5 summarizes our conclusions, results and opportunities for future work.

1.2 ARM Architecture

The following sections provide a summary of the ARM architecture [3].

1.2.1 ARM, A RISC Architecture

ARM incorporates typical features of a Reduced Instruction Set Computer (RISC) architecture such as a large uniform register file, simple addressing modes and a load/store architectural approach where data processing instructions operate only on register contents, but not directly on memory contents. Small implementation size, high performance and very low power consumption are key attributes of this architecture.

ARM instruction set is a set of 32-bit instructions providing data-processing and control functions. Thumb instruction set was introduced as a set of 16-bit instructions with a subset of the functionality to provide improved code size. For example, a particular compute functionality, which may require two 32-bit ARM instructions may be acheived by using three 16-bit Thumb instructions, leading to a space savings of 16-bits. However, this comes at a slight reduction in performance as the processor now has to execute one additional instruction in Thumb mode.

ARM supports a switch between the ARM and Thumb states, which allows for assembling performance critical segments using the ARM instruction set. The ARM and Thumb instructions can interwork freely, that is, different procedures can be assembled to different instruction sets.

The Thumb-2 instruction set was developed to extend the original 16-bit Thumb instruction set with many 32-bit instructions. Much of the functionality available is identical in ARM and Thumb-2 instruction sets. 16-bit Thumb instructions can be interleaved with 32-bit Thumb instructions within the same procedure resulting in an optimal combination of code size and performance.

1.2.2 Memory Organization

ARM instructions are word-aligned and Thumb instructions are halfword-aligned. For data, ARMv7 architecture supports unaligned data access, which is significantly different from the earlier versions of the ARM architecture. Table 1.1 shows the facilities provided by the ARM architecture memory model.

Address Space	A single flat address space of 2^{32} 8-bit bytes
Addressability	Byte addressable
Facilities	Faulting unaligned memory access. Restricting access by applications to specified areas of memory. Translating virtual addresses provided by executing instructions into physical addresses. Altering the interpretation of word and halfword data between big-endian and little-endian. Optionally preventing out-of-order access to memory. Controlling caches. Synchronizing access to shared memory by multiple processors.

Table 1.1: ARM Architecture - Memory.

1.2.3 Core Registers

ARM registers are 32-bits in size. To hold a doubleword, two consecutive registers are used and, to hold a quadword, four consecutive registers are used. Refer to Table 1.2 for a list of ARM registers.

1.2.4 Data Types

ARM architecture supports many data types for data held in memory and registers and, for data used in Load/Store operations. See Table 1.3 for a list of data types supported.

Registers	Function
R0 - R12	General Purpose Registers
R13	Stack Pointer (SP)
R14	Link Register (LR)
R15	Program Counter (PC)
APSR	Application Program Status Register with N, C, Z, V condition code flags
ISETSTATE	A set of J, T bits to determine instruction set used by processor J T 0 0 ARM 0 1 Thumb
ITSTATE	8-bits wide with top 3 bits indicate If-Then execution condition for Thumb IT instruction.
ENDIANSTATE	0 Little-Endian 1 Big-Endian

Table 1.2: ARM Architecture - Registers.

Data Location	Supported DataTypes
In Memory	Byte 8 bits Halfword 16 bits Word 32 bits Doubleword 64 bits
In Registers	32-bit pointers Unsigned or Signed 32-bit integers Unsigned 16-bit or 8-bit integers, held in zero-extended form Signed 16-bit or 8-bit integers, held in zero-extended form Two 16-bit integers packed into a register Four 8-bit integers packed into a register Unsigned or signed 64-bit integers held in two registers
For Load & Store Operations	Byte 8 bits Halfword 16 bits Word 32 bits Doubleword 64 bits Two or more words

Table 1.3: ARM Architecture - Data Types.

1.2.5 Addressing modes

Address for a Load/Store operation is formed from two parts - a Base Register and an Offset. See Table 1.4 for supported addressing modes in ARM.

Base Register	For Loads	R0 - R12, PC
	For Stores	R0 - R12
Offset Type	Immediate	
	Register	
	Scaled Register	
Addressing Modes	Offset	MemAddr = Base Register +/- Offset
	Pre-indexed	MemAddr = Base Register +/- Offset
		Base Register = MemAddr
	Post-indexed	MemAddr = Base Register
		Base Register = MemAddr +/- Offset

Table 1.4: ARM Architecture - Addressing modes.

1.2.6 Instruction Set

ARM and Thumb-2 architectures support a rich set of instruction types [3]. This section provides a summary of the instruction types with a few examples.

- Branch Instructions, See Table 1.11.
- Data-processing Instructions, See Table 1.5.
- Status Register Access Instructions. See Table 1.8.
- Load/Store Instructions, See Table 1.6.
- Load/Store Multiple Instructions. See Table 1.7
- Advanced Single Command Multiple Data (SIMD) and Vector Floating Point (VFP) Load/Store Instructions. See Table 1.9
- Advanced SIMD and VFP Data-processing Instructions. See Table 1.10

Table 1.5: ARM Architecture - Data Processing Instructions.

Instruction Class	Examples
Register	AND{S}<Rd>,<Rn>,<Rm>{<shift>} Instruction performs a bitwise AND of a register value Rn with an optionally-shifted register value Rm and writes the result to the destination register Rd. ADD{S}<Rd>,<Rn>,<Rm>{<shift>} Instruction performs a bitwise AND of a register value Rn with an optionally-shifted register value Rm and writes the result to the destination register Rd.
Register Shifted Register	EOR{S} <Rd>,<Rn>,<Rm> <type><Rs> Instruction performs a bitwise Exclusive OR of a register value Rn and a register shifted register value Rn, Rm and writes the result to the destination register Rd. CMN<Rd>,<Rm> <type><Rs> Compare Negative Instruction adds a register value Rn and a register-shifted register value Rm, Rs. It updates the condition flags based on the result, and discards the result.
	SUB{S}<Rd>,<Rn>,#<const> Instruction subtracts an immediate value
Immediate	Continued on next page

Table 1.5 – continued from previous page

Instruction Class	Examples
Immediate	#const from a register value Rn writes the result to the destination register Rd. ADR <Rd>,<label> Instruction adds an immediate value to the PC to form a PC-relative address, and writes the result to the destination register Rd.
Multiply & Multiply Accumulate	MUL{S}<Rd>,<Rn>,<Rm> Multiplies two register values Rn, Rm. The least significant 32-bits of the result are written to the destination register. SMULL <RdLo>,<RdHi>,<Rn>,<Rm> Signed Multiply Long multiplies two 32-bit signed register values Rn, Rm to produce a 64 bit result saved in RdLo, RdHi.
Saturating Addition & Subtraction	QADD<Rd>,<Rn>,<Rm> Instruction adds two register values Rm, Rn, saturates the result to the 32-bit signed integer range and writes the result to the destination register Rd. If saturation occurs, it sets the Q flag in the APSR.
	SMLABB<Rd>,<Rn>,<Rm> Signed Multiply Accumulate performs a
Halfword Multiply & Multiply Accumulate	Continued on next page

Table 1.5 – continued from previous page

Instruction Class	Examples
	signed multiply-accumulate operation on two 16-bit signed quantities taken from bottom 16-bits of Rn, Rm. The 32-bit product is added to a 32-bit accumulate and the result is written to the destination register Rd.

1.2.7 Conditional Execution

Most ARM instructions and, most Thumb instructions can be conditionally executed based on the values of the APSR condition code flags N, Z, C, V.

Bits [31:28] of the ARM instruction contain the condition and if this condition satisfies APSR condition code flags, the instruction is executed, otherwise the instruction acts as a NOP. See Table 1.12 for a list of condition code flags.

Thumb instructions can be made conditional by a preceeding IT instruction which encodes the condition. This 16-bit IT instruction provides an If-Then-Else functionality and makes upto four following instructions conditional. For example, an ITTTE EQ instruction imposes the EQ condition on the first three following instructions and NE condition on the next instruction. The instructions that are made conditional by an IT instruction are called its IT block.

1.3 The MASC ARM Emulator

Micro-architecture Santa Cruz (MASC) Laboratory is developing an emulator research infrastructure to execute ARM, SPARC and x86 binaries. The emulator can run on both x86 and ARM processors with Linux operating system.

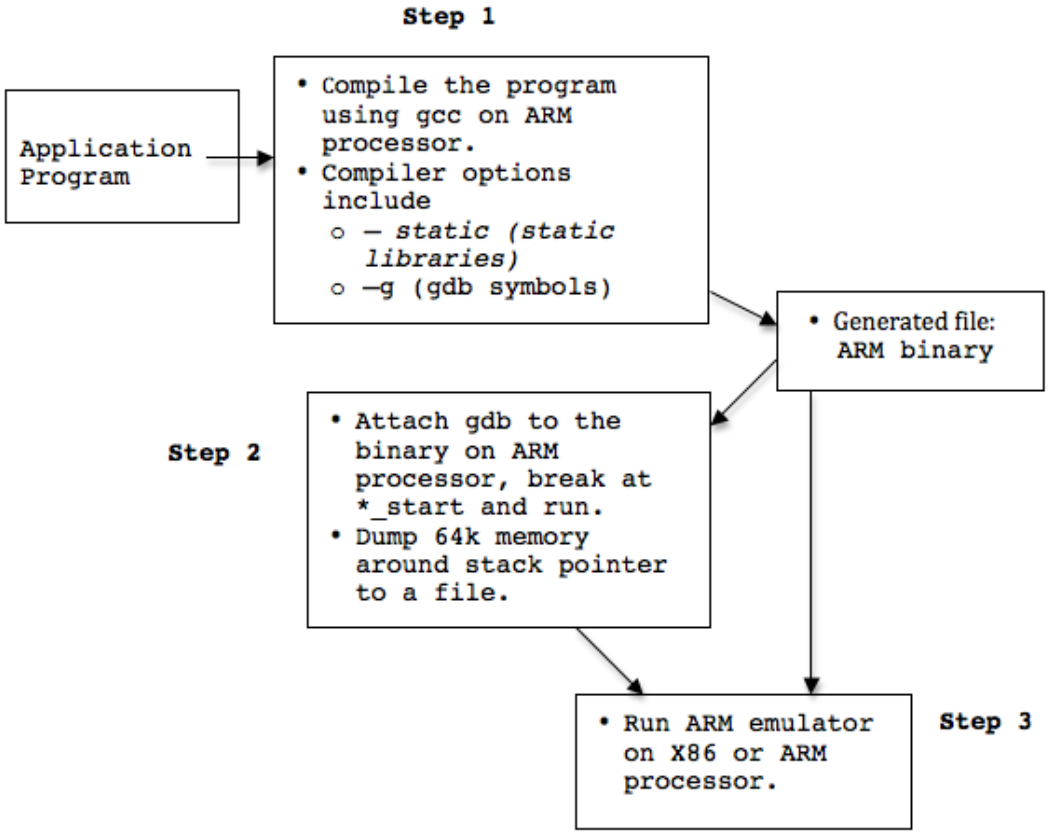

Figure 1.1: Steps To Run The ARM Emulator.

Fig. 1.1 shows the three steps to run the emulator. The first step is to compile the application program with static libraries and gdb symbols on a computer with ARM processor. The second step is to attach gdb to the binary to dump stack memory around the stack pointer to a file and, the third step is to run the emulator on a computer with x86 or ARM based processor.

The emulator requires certain input parameters to run. Mandatory parameters include stack specific information and ARM binary. Optional parameters, which are implemented as part of the validation framework, include an execution trace file and a system call trace file. See Appendix A, *Running the ARM Emulator*.

1.3.1 The Emulator Components

As shown in Fig. 1.2, the execution loop of the emulator fetches an instruction, then cracks the instruction into multiple emulator instructions and executes the cracked instructions. Validate component was developed as part of validation framework. Every instruction passes through Fetch, Crack and Execute stages mandatorily and Validate component optionally. In other words, user has an option to run the emulator with or without validation turned on.

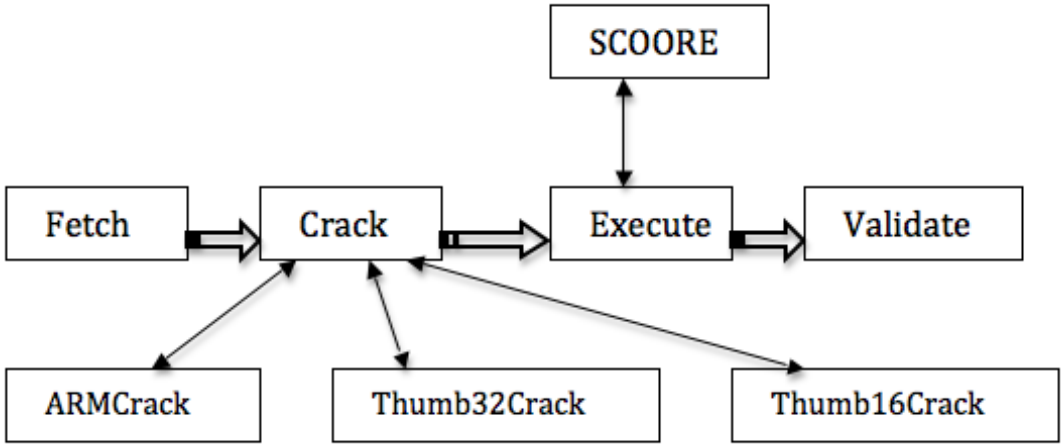

Figure 1.2: The Emulator Components.

During the Fetch stage, instruction associated with the current PC is fetched from the text section of the application program's simulated virtual memory 1.3.4.

In the Crack stage, the instruction is checked for type - ARM, Thumb-16 and Thumb-32, and then cracked into two or more emulator instructions accordingly 1.3.2.

During the Execute stage, each of the cracked emulator instructions is executed using the SCOORE execution engine being developed at the MASC lab 1.3.3. The only exception to this case is when an instruction belongs to an IT block and does not satisfy the IT block condition. In this scenario, not all the cracked instructions may execute.

Validate is responsible for validating the current PC, instruction, modified registers, modified memory and condition codes for every instruction.

1.3.2 Cracked Emulator Instructions

In the emulator, an ARM instruction is cracked into multiple *scinsts - Santa Cruz Instructions* instructions during the Crack stage. Emulator class *scuop* is an abstraction for each of these emulator instructions with members shown in Table 1.13.

1.3.2.1 *scuop - Santa Cruz Micro Operation*

Format of the *scuop* is as follows:

- *CrackInst::setup(RAWDInst *rinst, InstOpcode iop, Scopcode sop,*
  uint8_t srcA,uint8_t srcB, uint32_t immA, uint8_t dstA,
  unsigned seticc, unsigned setRND, unsigned getRND);

Where:

- *rinst* - ARM instruction abstraction in the emulator
- *iop* - Operation type, examples
  - iRALU, iAALU for ALU operations
  - iBALU_BRANCH for control operations]item iBALU_JUMP for jump
  - iBALU_CALL, iBALU_RET for sub-routine call and return
- sop - OPCode used by *sccore* engine to execute *scinst*, examples
  - OP_S32_ADD: Performs 32-bit Arithemetic ADD.
  - OP_S64_ADD: Performs 64-bit Arithemetic ADD.
  - OP_U16_ADD: Performs 16-bit unsigned ADD.
  - OP_S08_SUB: Performs 8-bit SUB.
- srcA: An abstraction for Rn, the Source Register.
- srcB: An abstraction for Rm, the Second Source Register.
- immA: An abstraction for immediate field.
- dstA: An abstraction for Destination Register, Rd.
- seticc: Flag to set condition codes.

1.3.2.2 How Crack Works

In this section, we show a step by step how-to on cracking an ARM instruction into multiple emulator instructions.

Our example ANDS<c ><Rd >, <Rn >, <Rm >,<shift > instruction performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

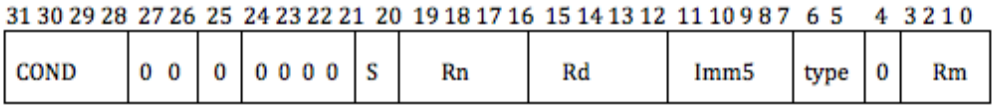

Figure 1.3: AND (Register) Instruction.

Fig. 1.3 shows a bit-level detail of AND (register) instruction. Bits [5:6] - *type* and Bits [7:11] - *imm5* determine the type of shift applied on register Rm, as shown in Table 1.14. The result of bitwise AND of shifted Rm and, Rn is saved to the destination register Rd. Here is the crack implementation for this instruction. Note that the code is shown only for *type* bits [5:6] '00' (LSL).

```
if(imm5 != 0) {
  if(type == 0) {
    CrackInst::setup(rinst, iAALU, OP_S32_SLL, RM, 0, SHIFT_IMM,
                                RM, 0, 0, 0);
  }
}
CrackInst::setup(rinst,iAALU,OP_S32_AND,RM,RN,0,RD,0,0,0);
```

1.3.3 SCOORE execution engine

SCOORE, the Santa Cruz Out-Of-Order RISC Engine, is a high-performance out-of-order execution engine being developed at the MASC research lab at UCSC to execute each of the *scinsts* generated during the crack stage. The goal of the MASC emulator research infrastructure is to support multiple ISAs, namely SPARC, X86 and ARM architectures. Instructions

from these ISAs are cracked into emulator *scinsts* and the SCOORE execution engine executes these instructions independent of the ISA type.

1.3.4 Application Program's Virtual Memory Simulation

The ARM emulator allocates memory using `malloc()` to simulate application program's virtual memory sections stack, text and heap. Note that initialized data section and heap are maintained together by the same memory block in the emulator.

The simulated stack is initialized with the stack dump file which is passed in to the emulator as one of the input parameters.

The ARM binary is encoded in standard ELF format [8] [2] which allows the use of programming interface provided by BFD package [5] to read the input ARM binary sections to initialize text, data and heap segments. In addition, this package provides an API to return the start PC of the instruction execution sequence of the ARM binary, which is used by the emulator to start executing the instruction at this PC.

1.3.5 SCOORE Address Space Translation

Since the application program is compiled on one computer and is executed through simulation on another computer, there is a need to translate addresses from the guest machine to the host machine. The guest machine refers to the computer with ARM processor on which we compile the application program and the host machine refers to the x86 computer where we run the emulator. A guest address refers to an address in the application program's virtual memory space on the guest machine and a host address refers to an address of the application program's simulated virtual memory space on the host machine.

As shown in Fig. 1.4, the first step to translate a guest address to a host address is to find the segment to which this address belongs to and then find the offset of the address from the start of the segment. Note that the start address of the segments in the application program binary are known and saved in the emulator at the time of program loading, see section 3.2 for details. This offset is then used to compute the corresponding address on the host machine. This computation is valid for any guest adress in stack, text and heap segments.

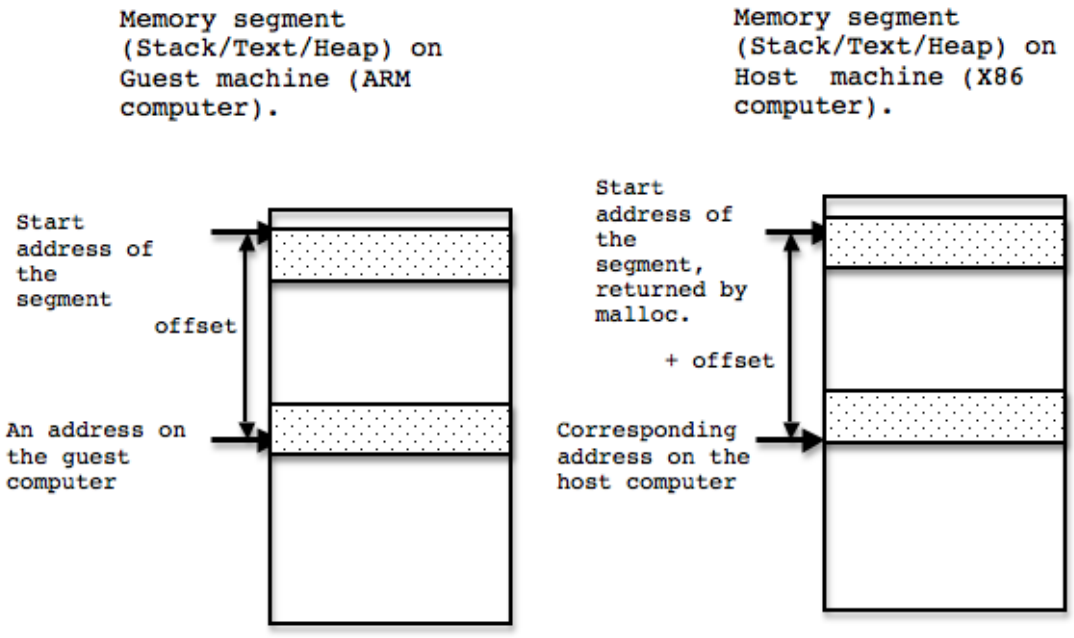

Figure 1.4: SCOORE Address Translation.

1.3.6 System Calls

The emulator handles most of the system calls by calling the underlying Linux system call interface with the same arguments. Some system calls require special handling, see section 3.3 for details.

Instruction Class	Examples
Register	STR <Rt>,[<Rn>, +/-<Rm >{, <shift>}] Store Register calculates an address from a base register Rn and an offset register Rm value, stores a word from register Rt to memory LDR <Rt>,[<Rn>, +/-<Rm >{, <shift>}] Load Register calculates an address from a base register Rn and an offset register Rm value, loads a word from memory to register Rt
Immediate	STR <Rt>,[<Rn>{,#+/-<imm12 >}] Store Register calculates an address from a base register Rn and an immediate offset imm12, and stores a word from register Rt to memory LDR <Rt>,[<Rn>{,#+/-<imm12 >}] Load Register calculates an address from a base register Rn and an immediate offset imm12, loads a word from memory to register Rt
Literal	LDR <Rt>, <label> Store Register calculates an address from PC and an immediate offset, and loads a word from memory to register Rt
Unprivileged	STRT <STRT Rt>,[<Rn>, +/-<Rm >{, <shift>}] Store Register Unprivileged stores a word from register Rt to memory. The memory access is restricted as if the processor were running in User mode LDRT <LDRT Rt>,[<Rn>, +/-<Rm >{, <shift>}] Store Register Unprivileged stores a word from register Rt to memory. The memory access is restricted as if the processor were running in User mode

Table 1.6: ARM Architecture - Load/Store Instructions.

Instruction Class	Examples
LDM/LDMIA/LDMFD	LDM<Rn>!,<registers>, Load Multiple Increment After loads multiple registers from consecutive memory locations using an address from a base register.
POP	POP<registers> Pop multiple loads multiple registers from the stack, loading consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.
STMDB/STMFD	STMDB<Rn>!,<registers>, Store Multiple Decrement Before stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.
PUSH	PUSH<registers> Push Multiple Registers stores multiple registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

Table 1.7: ARM Architecture - Load/Store Multiple Instructions.

Instruction Class	Description
MRS	Moves the contents of the Application Program Status Register (APSR) to a general purpose register.
MSR	Moves the contents of a general purpose register to APSR register.

Table 1.8: ARM Architecture - Status Register APSR Access Instructions.

Instruction Class	Description
VLDM	VLDM <Rn>!, <list> Vector Load Multiple loads multiple extension registers from memory locations using an address from an ARM core register.
VSTR	VSTR <Sd>!, [<Rn >, #+/-<imm >] This instruction stores a single extension register to memory, using an address from an ARM core register, with an optional offset.

Table 1.9: ARM Architecture - Advanced SIMD and VFP Load/Store Instructions.

Example Instructions Class	Description
VADD	VADD <dt>, <Qd>, <Qn>, <Qm> Vector Add adds corresponding elements in two vectors, and places the results in the destination vector.
VPADAL	VPADAL <dt>, <Qd>, <Qm> Vector Pairwise Add and Accumulate Long adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector.
VDIV	VDIV <Dd>, <Dn>, <Dm> This instruction divides one floating-point value by another floating-point value and writes the result to a third floating-point register.
VNEG	VNEG <dt >, <Qd >, <Qm > Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit.

Table 1.10: ARM Architecture - Advanced SIMD and VFP Data-Processing Instructions.

Instruction Class	Examples
Branch to target address	B <label> Branch causes a branch to a target address
Call a subroutine	BL <label> Branch with Link calls a subroutine at a PC-relative address
Call a subroutine, change instruction set	BLX <label> Branch with Link Exchange Instruction Sets calls a subroutine at a PC-relative address, and changes instruction set from ARM to Thumb, or from Thumb to ARM
Block Data Transfer	STMDA <Rn>,<registers> Store Multiple Decrement After stores multiple registers to consecutive memory locations using address from base register Rn LDM <Rn>,<registers without PC > In exception mode, this loads multiple user mode registers from consecutive memory locations using an address from a register Rn. This instruction is unpredictable in User or System modes

Table 1.11: ARM Architecture - Branch And Block Data Transfer Instrictions.

Cond	Mnemonic Extension	Meaning(Integer)	Condition Flags
1110	None(Al)	Always	Any
0000	EQ	Equal	Z == 1
0001	NE	Not Equal	Z == 0
0010	CS	Carry Set	C == 1
0011	CC	Carry Clear	C == 0
0100	MI	Minus, Negative	N == 1
0101	PL	Plus, Positive or Zero	N == 0
0110	VS	Overflow	V == 0
1000	HI	Unsigned Higher	C == 1 & Z == 0
1001	LS	Unsigned Lower or Same	C == 0 & Z== 1
1010	GE	Signed Greater Than or Equal	N == V
1011	LT	Signed Less Than	N != V
1100	GT	Signed Greater Than	Z == 0 & N == V
1101	LE	Signed Less Than or Equal	Z == 1 — N != V

Table 1.12: ARM Architecture - Condition Codes.

Member	Description
op	SCOORE operation type
src1	First Source Register of the instruction
src2	Second Source Register of the instruction
useImm	A flag to indicate if immediate is used
imm	Value of the immediate
dst	Destination Register
seticc	A flag to set condition codes N,Z,C,V

Table 1.13: MASC ARM Emulator - Santa Cruz Micro Operation (scuop) Class Members.

Type Field Bits [5:6]	Shift Type	Shift Value
00	LSL(Logical Shift Left)	UInt(imm5)
01	LSR(Logical Shift Right)	if imm5 == '00000' then 32 else UInt(imm5)
10	ASR(Arithmetic Shift Right)	if imm5 == '00000' then 32 else UInt(imm5)
11	imm5 == '00000' then RRX(Rorate Right With Extend) else ROR(Rotate Right)	1 UInt(imm5)

Table 1.14: Register Shift Using Type and Imm5 Field Values.

Chapter 2

Design And Implementation Of A Validation Framework

Having a validation framework for the ARM emulator serves two purposes. First, it helps identify incorrectly implemented or not yet implemented instructions and second, it helps find regression bugs quickly and easily. Regression bugs refer to those bugs that are introduced unintentionally into already working code during new feature development or while trying to fix other bugs. This chapter describes the design and implementation of a validation framework for the ARM emulator.

2.1 Overview

Every instruction execution in the ARM emulator may result in modified register or memory values and condition codes. Emulator results need to be validated against expected results obtained from the same program execution on ARM processor. Generation of the expected results and validation of the emulator results against the expected results are automated and are discussed in detail in the subsequent sections. While validating regular instructions is straightforward, handling system calls in the emulator is a bit more involved and is dealt with on a case by case basis. We make the following contributions

- Auto generate expected results.
- Validate the emulator results against the expected results.

With validation, the sequence of three steps to run the emulator shown in Fig. 1.1 slightly changes to the steps shown in Fig. 2.1.

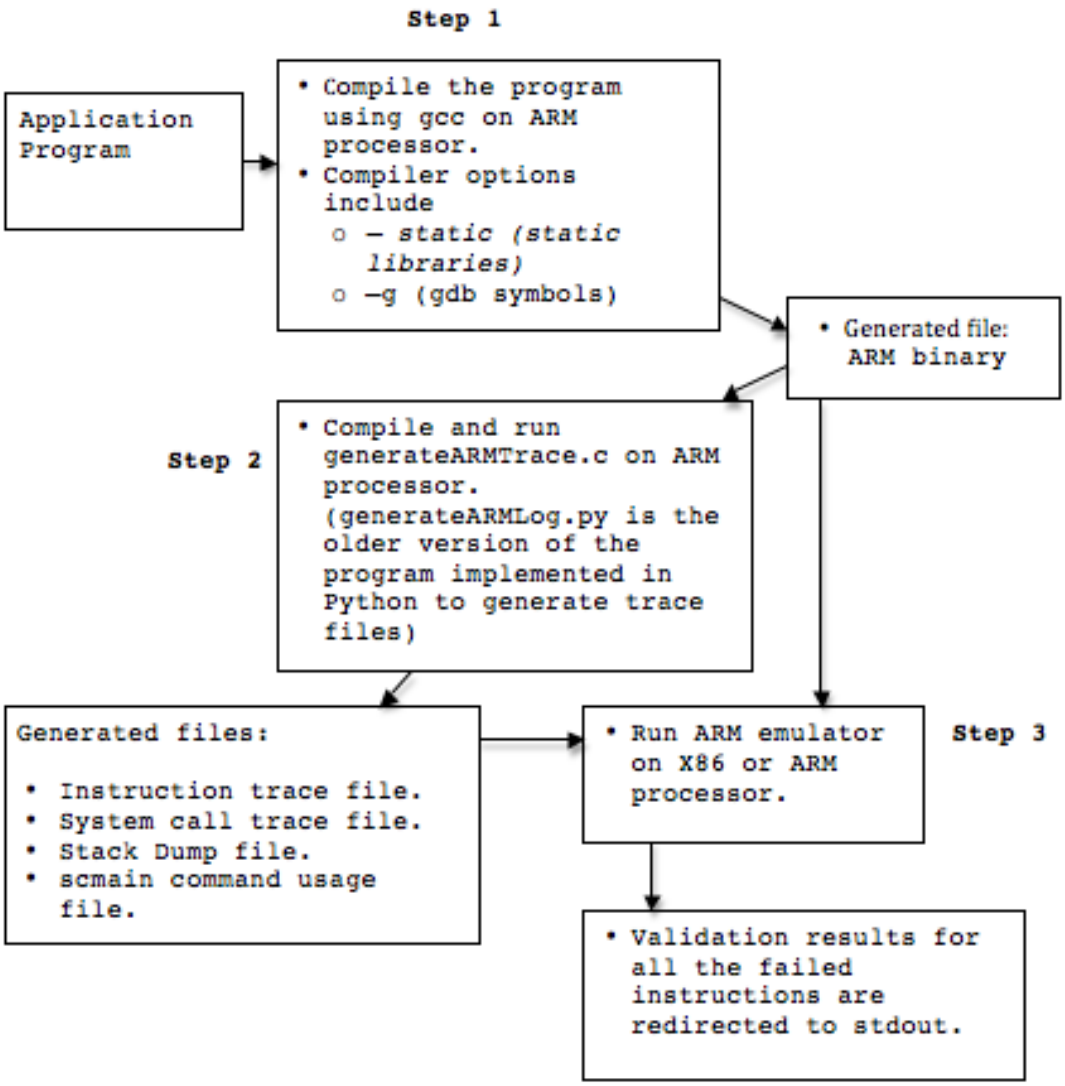

Figure 2.1: Steps To Run The ARM Emulator With Validation.

2.2 Generate Expected Results

The program to generate the expected results first attaches gdb to the program, steps through each instruction (with one exception, as described in section 2.3.5), sends commands

to gdb to dump register values and, processes the output returned by gdb. In summary, the program has two distinct but equally important tasks -

- Interact with GDB
- Process GDB Output

The program generates -

- An instruction trace file,
- A system call trace file and
- A file with 64kB binary dump of memory around the stack pointer.

2.2.1 Generating Expected Results Using Python - Version I

At first, to generate expected results, a Python Program 'generateARMLog.py' was implemented. It uses the 'pexpect' Python module to interactively communicate with gdb to step through each instruction. It imports the standard Python package 're' to use regular expression symantics of the language to process the output.

Even though the Python program 'generateARMLog.py' generates expected results reliably and was used extensively over the last few months, there is a need to speedup its execution time. With its current processing speed of about 100 instructions per minute, generating expected results for even a simple 'hello world' program takes about 4 hours of time.

2.2.2 Generating Expected Results Using Two-threaded Implementation - Version II

As outlined earlier, generating expected results has two tasks. The first task is to interact with gdb process to step through instructions and the second task is to process the gdb output and generate expected results.

To utilize the multi-core ARM processor in MASC lab, our first attempt at speeding up involved implementing a two threaded parallel program with one thread interacting with gdb and the other thread processing gdb output and generating expected results. But with bulk of the time spent on interaction with gdb, this technique did not yield favourable speedup.

2.2.3 Generating Expected Results In Chunks - Version III

This strategy was not so much to speedup the generation of expected results but to alleviate the long wait time before they are generated. In this method, the Python program generates multiple files with each file containing expected results for a predefined number of instructions. This allows the user to start running the emulator to validate a subset of instructions while the expected results for the rest of the instructions are being generated. This strategy works well when the emuator is still under development with the assumption that validation usually finds bugs in the subset and finding the root cause and fixing the bugs usually take time. Fixing bugs in one subset while generating expected results for another subset can happen in parallel. Unfortunately, this strategy does not work well for regression tests.

2.2.4 100x Speedup of Generating Expected Results Using C, GDB MI interface

Looking further into the issue of speeding up expected results generation, the key thing to note is that by speeding up the interaction with gdb the overall speedup of the program can be acheived. Python being an interpreted language with pexpect module processing every line returned from gdb considerably slows down the expected results generation.

Our third attempt at speeding up involved implementing a C program using gdb machine interface (MI) library to interact with gdb and process output. Gdb MI [1] is a line based machine oriented text interface to gdb. The library provides interfaces to spawn gdb process in interpreter mode, step through instructions and parse gdb output. With this implementation, generating expected results is significantly faster, with under 2 minutes for the simple 'hello world' program as opposed to 4 hours with the earlier version of the Python program.

2.3 Compiling And Running The Program *generateARMTrace.c*

Compile and run this program on an ARM processor. This program requires 'gdbmi' library. On masca1(an ARM computer in MASC lab), gdbmi library is installed at /root/gdbmi.

```
To compile the program on masca1:
-------------------------------


cc -O0 -Wall -gstabs+3 -I /root/gdbmi/libmigdb/src
```

```
       generateARMTrace.c /root/gdbmi/libmigdb/src/libmigdb.a
       -o generateARMTrace

To run:
------

./generateARMTrace

Example:
-------

arm-shell# ./generateARMTrace
Usage: generateARMTrace <binary> <results_dir>

arm-shell# ./generateARMTrace hello logs > output

 This program generates the following files...

 logs/scmain_cmd                   scmain (emulator) command
                                   usage help
 logs/stack_mem_dump.bin           stack memory dump
 logs/syscall_trace                system call trace
 logs/inst_trace                   instruction trace
```

2.3.1 Instruction Trace File

This file provides a trace of instruction execution sequence. Every instruction has a dedicated section in this file. Useful information about the executed instruction such as pc, instruction, modified single/double word registers and modified condition codes are saved in the section. Let us look at a couple of sections in detail in this file.

```
----------------
```

```
Instruction number: 3
pc: 0x8790
inst: 0x1b04f85d
r2=0x1
r14=0xbeffe9b4
----------------
Instruction number: 4
pc: 0x8794
inst: 0xf84d466a
r3=0xbeffe9b4
----------------
Instruction number: 5
pc: 0x8796
inst: 0x2d04f84d
r14=0xbeffe9b0
----------------
Instruction number: 6
pc: 0x879a
inst: 0x0d04f84d
r14=0xbeffe9ac
----------------
```

The first line line of any section is the instruction sequence number in the program execution. The second line is the PC value and the rest of the lines are for the modified registers and their contents. In the above example, for instruction 3, modified registers are R2 and R14 while for instruction 4, modified register is R3.

Note that in the emulator, the register numbering starts from R1 while on the ARM processor, the register numbering starts from R0. We take care of this by incrementing the modified register number before writing to the instruction trace file.

2.3.2 System Call Trace File

System calls are a means by which a user program can request for the underlying operating system services. Some system calls provide a pointer to a memory buffer as one of the input arguments and request the operating system to populate it with some specific information. The `uname()` is one such system call, it provides a pointer to the `utsname` structure and the operating system updates the memory buffer with name and other information about the kernel. Subsequent instructions may load values from this section of the memory to carry out their operations.

In the emulator, system calls are serviced by calling the underlying operating system system call interface. In such scenarios as explained above, where a section of the memory is changed when a system call is executed, the validator can yield false negative results for the subsequent instructions that use the information in the changed memory buffer. This is due to the fact that the validator validates the results by comparing the 'expected results' obtained from the program execution on ARM computer and 'emulator results' obtained from the emulator execution on x86 computer. Obviously, the information returned by the kernels of these two computers will not match in most of the cases. For such system calls, for the validator to work correctly, when generating the system call trace file on ARM computer, the system call resulting memory contents are copied over to the system call trace file. When the emulator finds such system calls, when validation is turned on, the emulator reads the system call results from the system call trace file and copies to the corresponding emulator virtual memory section. Note that this matching is not needed when validation is turned off and in that case, the emulator system call execution falls back to calling the underlying Linux system call interface.

The system call trace file records the sequence of system calls in the program. Every system call has a dedicated section in this file. Each section records the PC, instruction and the system call number for a system call. For system calls that modify the memory, the resulting memory contents are copied to this file as well.

In the current implementation, the resulting memory contents of the `uname()` and `fstat64()` system calls are copied to the system call trace file.

Here is an example contents of system call trace file highlighting three different types of information saved in system call trace file depending on the system call.

```
----------
```

```
pc: 0x25988
syscall: 122  // uname
Linux
masca1
3.0.0-1205-omap4
#10-Ubuntu SMP PREEMPT Thu Sep 29 03:57:24 UTC 2011
armv7l
----------
pc: 0x8de4
syscall: 45
----------
pc: 0x8de4
syscall: 197   // fstat64
000000000000000b
0000000400000000
0000000100002190
0000000500000dca
0000000000008801
0000000000000000
0000000000000000
0000000000000400
0000000000000000
20b585ad50636548
20b585ad50636548
37fdfc0150635fd5
0000000000000004
fffffffff00068128
00009dc500068128
0002101700068128
0000000c0004981c
000205e100068128
```

```
----------
pc: 0x10f04
syscall: 192
----------
```

2.3.3 Stack Dump File

The file 'stack_mem_dump.bin' provides a binary dump of the memory around the stack pointer. The binary dump is obtained using the gdb command `'dump binary memory mem start_addr end_addr'` with `start_addr` being `SP & 0xFFFF0000` and `end_addr` being `SP | 0x0000FFFF`, where `SP` is the stack pointer.

2.3.4 Scmain Command Usage Help File

The file 'scmain_cmd' provides the command to run the scmain (ARM) emulator. This file is useful as it eliminates the need to note down the values of stack pointer and start of the stack buffer which are both required to run the emulator.

2.3.5 Special handling of `ldrex/strex` Instructions

Load Register Exclusive `ldrex` and Store Register Exclusive `strex` instructions require the processor to have an exclusive access to memory. Stepping through either of these two instructions using gdb violates this exclusivity and the instruction execution waits in a tight loop for the lock, which never becomes available. To overcome this situation, a lookahead logic is implemented to not to step through these instructions.

```
89f6: f3bf 8f5f   dmb sy         // First data barrier instruction
89fa: e852 1f00   ldrex r1, [r2] // LDREX
89fe: 4299        cmp r1, r3
8a00: d105        bne.n 8a0e <__libc_start_main+0x1ce>
8a02: e842 0100   strex r1, r0, [r2]    // STREX
8a06: f091 0f00   teq r1, #0
8a0a: d1f6        bne.n 89fa <__libc_start_main+0x1ba>
8a0c: 4619        mov r1, r3
```

```
8a0e: f3bf 8f5f   dmb sy          // Second data barrier instruction
8a12: 428b        cmp r3, r1
```

The lookahead logic takes advantage of the use of two Data Memory Barrier `dmb` instructions that enclose the instruction sequence containing ldrex/strex instructions to provide the exclusivity. The logic looks ahead at the instruction next to dmb to determine if it is an ldrex instruction. If so, it finds the corresponding strex instruction and the closing dmb instruction. Once it is determined that the block of instructions is ldrex/strex, instead of stepping through each instruction, a break point is set for the second dmb instruction and gdb execution is continued using the command `continue` instead of stepping using the command `si` through individual instructions.

A similar look ahead logic is implemented in the validate component of the emulator as well.

2.4 Validate Component Of The Emulator

Now comes the time to validate the emulator. After every instruction execuiton in the emulator, if the user choses to validate, the validate component reads one section of the instruction trace file to get the expected PC, instruction, modified register contents, condition codes. Then compares these results to the results generated by the emulator and prints pass/fail results with some useful debug information about what was expected and what was generated by the emulator in case the validation fails for that instruction.

When the emulator encounters a system call in the program execution, it reads a section of information from the system call trace file to validate the PC, instruction, system call number. Those system calls for which the resulting memory contents are copied, emulator reads this information and copies over to the corresponding application program's virtual memory space.

If the instruction belongs to ldrex/strex block, the lookahead logic similar to that described in section 2.3.5 is implemented in the emulator to not to validate the set of instructions in ldrex/strex block.

The following is shows examples of the emulator output when an instruction passes validation and another instruction that fails validation.

```
1) Instruction execution with no failures

  Th[0] next PC=0X8b8c

  // CRACK Expansion
  expanded:
  +: TMP3 = ZERO OP_S64_COPYPCL ZERO useImm=1 imm=35732 seticc=0
                  setrnd=0 getrnd=0
  +: TMP2 = TMP3 OP_U32_ADD ZERO useImm=1 imm=12 seticc=0 setrnd=0
                  getrnd=0
  +: R3 = TMP2 OP_U32_LD_L ZERO useImm=0 imm=0 seticc=0 setrnd=0
              getrnd=0

  // Execution information
  executed:
  Th[0] -: modified state
    rf[78]=0x8b94 ccrf=0x0
  Th[0] -: modified state
    rf[77]=0xc00008ba0 ccrf=0x0
    addr(ldu32) 0x00008ba0
  ldu32 [0x8ba0] -> 0x00009428
  Th[0] -: modified state
    rf[4]=0x9428 ccrf=0x0

  // Validation information
  inITBlock 0, NOPStatus 0, flushDecode 0
  inst num: emul Instruction number: 10, trace file Instruction

  reg_val 9428
  reg_num 4 reg_val 9428 f_reg_val 0x00009428
```

2) Instruction execution with failures

```
  Th[0] next PC=0X8d10

  // CRACK Expansion
  expanded:
  +: R10 = R3 OP_S32_ADD ZERO useImm=0 imm=0 seticc=0 setrnd=0
                 getrnd=0
  // Execution information
  executed:
  Th[0] -: modified state
    rf[11]=0x9428 ccrf=0x0

  // Validation information
  inITBlock 0, NOPStatus 0, flushDecode 0
  inst num: emul Instruction number: 17, trace file Instruction


    FAIL: pc values do not match
      expected_pv_val 0x8d12 emul_pc_val 0x8d10

  reg_val 9428
  reg_num 11 reg_val 9428 f_reg_val 0x00009428

    FAIL: register values don't match
      reg_num 11. exp_reg_val 0x00009428. emul_reg_val 0x00009000
```

Chapter 3

Enhancements To The ARM Emulator

While the first part of the thesis work is to design and implement a validation framework, the second, equally important part, is to improve the ARM Emulator functionality inorder to facilitate executing ARM binaries through the emulator. This chapter describes the enhancements made to ARM emulator functionality in various areas.

3.1 Overview

At the beginning of this work, the emulator was capable of interpreting most of the instructions. However, the emulator was not ready to execute ARM binaries. To facilitate executing ARM binaries through the emulator, we make the following contributions:

- Implement Program Loader Functionality.
- Implement System Calls.
- Fix incorrectly-working instructions.
- Implement VFP load/store instructions using a rapid development technique.

3.2 Program Loading

The Loader is a part of the operating system responsible for loading the program binary from disk to memory and making it ready for execution. The ARM binary follows the

standard ELF format [8] [2] and contains sections such as .text, .data and .bss etc. One of the loader's function is to combine two or more sections of the binary into a segment and pad the segment to align it to the page size, which is 4k by default for ARM binaries.

In the emulator, the loader function is implemented by a software routine. This routine uses BFD programming package [5] to read the application binary sections, follows the guidelines specified in ELF standard [2] to create segments, pads the segments to align to the page boundaries and allocates memory for the segments using malloc().

3.2.1 Loader Implementation

After examining the readelf output of a few program binaries, the following assumptions are made about the ARM program binary sections

- The first page of the ELF formated application binary contains .text section.
- The first data section is either .tdata or .bss whichever has the lower virtual memory address.
- The last data section, which is also a part of heap, is either .bss or _libc_freeres_ptrs

These assumptions are working well for binaries compiled with static libraries. Program Loading for binaries with dynamic libraries was not studied as part of this work and may be considered future work.

In the emulator, the loader routine combines sections to creates two segments - text and data. Data segment contains initialized data and also heap. In other words, data and heap of the application program are maintined as one memory block in the emulator.

3.2.2 Emulator Virtual Memory Space and Application Program Simulated Virtual Memory Space

Fig. 3.1 shows a pictorial representation of the application program's simulated virtual memory space with respect to the ARM emulator program's virtual memory space. As shown, the application program's simulated text and data segments live in the heap space of the emulator's virtual memory.

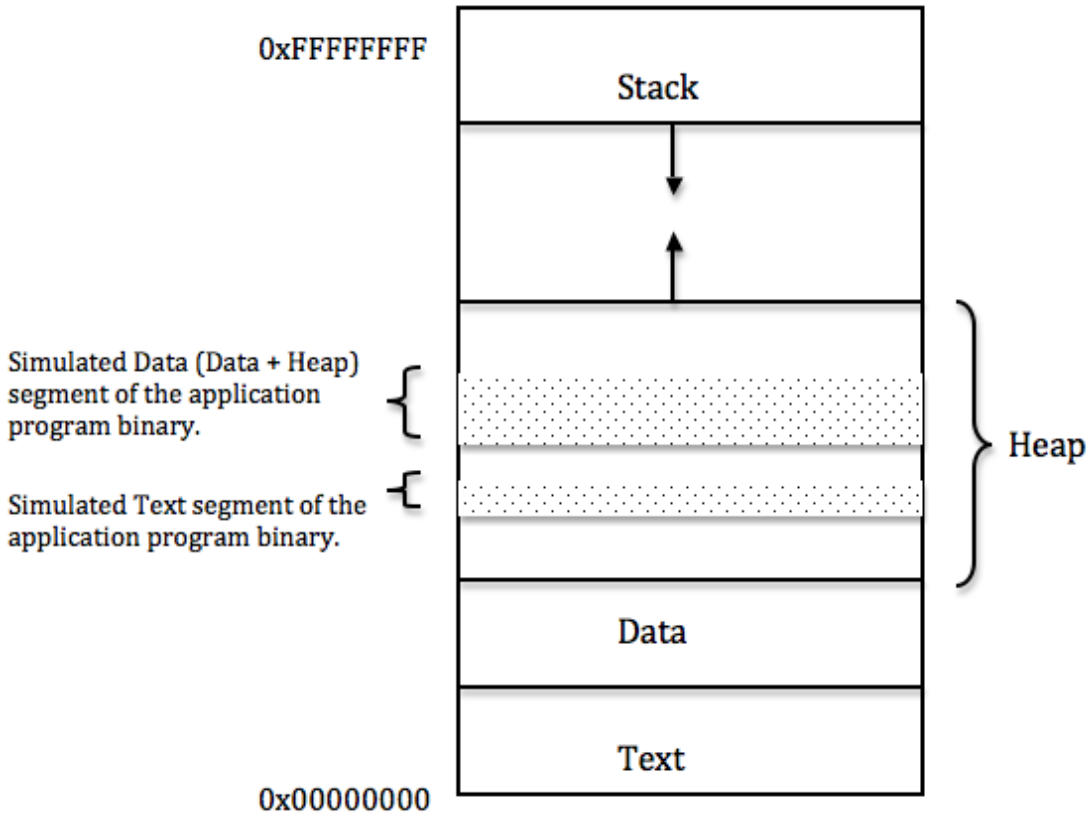

Figure 3.1: The Application Program Simulated Virtual Memory With Respect To The Emulator Program's Virtual Memory.

3.3 Emulator System Calls

In the emulator, depending on the system call functionality, its implementation falls into one of the three categories.

- The first type of system calls are serviced by simply calling the underlying Linux system call interface with the same arguments.
- The second type of system calls are serviced by calling the underlying Linux call interface just like in the first case, but the memory contents updated by OS are copied to the system call trace file and this content is copied over to the virtual memory of the application program during emulator validation, see section 2.3.2.

- The third type of implementation is to handle the system calls locally using `malloc()`.

3.3.1 Calling The Underlying Linux System Call Interface

The system calls `read()`, `write()`, `exit()` are some of the examples of the system calls for which the emulator calls the undelying Linux system call interface to get the work done.

3.3.2 Copying Contents From System Call Trace File To Application Program Virtual Memory Space Heap

When emulator encounters system calls in the program execution, it reads a section of information from the system call trace file for pc, instruction and system call number. For the system calls `uname()` and `fstat64()`, saved memory buffer information (refer to section 2.3.2) from the system call trace file is copied over to the application program's virtual memory space at the corresponding location.

3.3.3 Handling system calls locally using `malloc()`

The emulator needs to handle some of the system calls locally using `malloc()` without calling the underlying Linux system call API. These system calls are usually associated with application program's virtual memory space, specifically heap.

3.3.4 The `brk()` System Call

The system call `brk()` changes the data segment size by changing the location of the program break address. Calling the underlying Linux system call API changes the emulator program's break address, which is not the intended result, what we intend to change is the application program's break address. This is simulated by computing the heap space required by application program aligned up to the page size 4k and reallocating the memory using `realloc()`. Note that realloc, in addition to allocating the needed memory space, copies the contents from old memory block to the newly allocated one. The emulator keeps track of program break address throughout the program, it is first initialized to the top of heap during

program loading (refer to section 3.2) and adjusted to the top of new memory segment allocated to heap everytime this system call is executed.

3.3.5 The `mmap()` System Call

The system call `mmap()` creates a new mapping in the virtual address space of the calling process. One of the input arguments is the 'size' of the mapping. This system call is simulated in the emulator by allocating memory of 'size' using `malloc()`. There may be multiple such memory blocks allocated during the life of an application program and all the blocks are maintained and looked up just as text and heap.

3.4 Crack Code Optimization

An IT instruction is a feature only of Thumb instruction set and provides if-then-else functionality by having an associated condition upon which the next four instructional execution becomes conditional. As shown in Fig. 1.2, the emulator files Thumb16Crack.cpp and Thumb32Crack.cpp implement crack code for 16 and 32-bit Thumb instructions. By reorganizing code and making simple optimizations, these file sizes were reduced considerably, about 20k lines reduction in each of these two files. The following describes the details of the optimizations.

- Each instruction implementation separately checked to see if the instruction belonged to an IT block. Since there are hundreds of instructions, this consumes a lot of lines of code. One of the optimizations is to remove this check from every instruction implementation and have a common code block to determine if an instruction belonged to an IT block.
- Earlier, the cases for an instruction not-in-IT-block and in-IT-block&cond=Always are handled separately. Again by merging these two cases for hundreds of instructions, we were able to remove a lot of redundant lines in the code.

3.5 Bug Fixes

Using the validation framework a number of bugs were found in the ARM emulator. Subsequent sections describe the categories of issues found and fixed in the process.

3.5.1 Clear LSB of PC

Most of the data-processing and control instructions are capable of changing the program flow by changing the PC register i.e. these instructions have Rd, the destination register, set to R15 which is the PC register.

LSB of PC register value indicates the processor mode. A value of 0 indicates the instruction execution in ARM mode and a value of 1 indicates the instruction execution in Thumb mode.

However, the instruction itself is aligned to halfword (for Thumb-16 instruction) or word (ARM or Thumb-32 instructions).

As part of this bug fix, once it is determined that the instruction is an ARM/Thumb, we clear the LSB to access the correct instruction.

3.5.2 Issues With Instruction Decode

In these types of issues, the instruction decoding is wrong:

- The registers Rn, Rm and Rd are decoded incorrectly.
- Opcodes are decoded incorrectly.
- IT condition code is decoded incorrectly.
- Other fields such as register_lists, immediate fields are decoded incorrectly.

3.5.3 Issues With Instruction Cracking

In these types of issues, the instructions are implemented with a wrong set of *scinsts*:

- Instructions with one or two wrong *scinsts*
- Instructions for which the Crack functionality needed to be implemented completely.

3.5.4 Increment/Decrement Subsequent Memory Addresses

The instructions that perform block data transfers, usually transfer data to/from multiple registers from/to memory. This class of issues include:

- Not incrementing the register to subsequent registers in the list.
- Not incrementing the memory location for subsequent Load/Store operations.

3.5.5 Issues With Conditional Execution

The following items describe the types of issues fixed in the conditional execution of the instructions.

- An instruction is executed when its condition does not satisfy the condition code.
- An instruction is NOT executed when it satisfies the condition codes correctly.
- An instruction which belongs to an IT block and does not satisfy the IT block condition, is executed.
- An instruction which belongs to an IT block and satisfies the IT block condition, is NOT executed.

3.5.6 Size mismatch of structures defined in `unistd.h` between 32 and 64-bit architectures

The `unistd.h` header file defines the POSIX system calls and the structures used by the system calls.

Recall that system calls in the emulator, when validation is off, are handled by calling the underlying Linux system call interface with the same arguments. The system call `fstat()` provides a pointer to the `stat` structure and operating system fills this structure with relevant information.

One of the issues we ran into with this approach was the mismatch of the `stat` structure size on the 32-bit ARM architecture, where the application binary is compiled and the 64-bit x86 architecture, where the emulator is compiled. The size of the `stat` structure on 32 and 64-bit architectures is 88 and 144 bytes respectively. So when the application binary compiled for 32-bit ARM architecture is executed through emulator compiled for 64-bit x86 architecture and calls Linux system call interface, 144 bytes of the memory are modified leading to 144-88 = 56 bytes of memory corruption.

To overcome this issue, to run the emulator on x86 architecture, we compile the emulator on an x86 processor for 32-bit architecture.

3.6 Automatic Code Generation of VFP Load/Store Instructions

This section describes a rapid development technique to auto-generate code for a class of instructions. All the instructions that belong to a particular class of instructions generally perform a common set of operations. By exploiting this commonality, we are able to mass-produce code for a class of instructions rapidly. Specifically, we make the following contributions:

1. Programmatically parse ARM CPU specification manual.
2. Auto generate code for VFP Load/Store instructions.

Table 3.1 shows VFP Load/Store operations. Subsequent sections describe systematic parsing of the CPU specifications and auto generating the code for these instructions. The proposed methodology has proved its usefulness in:

1. Reduced development time at no extra cost.
2. Opportunity to generate optimized number of scuops in the crack stage for a class of instructions.
3. Ability to find and fix bugs across multiple instructions.
4. Opportunity to reuse parts of this code for another class of instructions, thus, speeding up the emulator development.

3.6.1 Programatic Parsing of the CPU Specification manual

In this section, we implement a rapid prototyping technique [10] to auto generate code for ARM and Thumb-2 Vector Floating Point (VFP) instructions. Table 3.2 shows the instruction operation psedocode taken directly from the ARM CPU specification manual. We grouped syntactically similar looking lines and designed match templates to facilitate effective parsing. We even make use of the psedocode indentation to remember the scope of the code and generate curly braces in the auto generated code. Match templates have optional values are enclosed in <>. Extracted values are enclosed in ( ).

3.6.1.1 Mining Rules

First mining rule:

(<if wback then>) Dest = if (OP) then (then_val) else (else_val)

Here are some example matching lines and their extracted fields using the template:

- address = if add then R[n] else R[n]-imm32;
  - <if wback then >= false
  - OP = add
  - then_val = R[n]
  - else_val = R[n]-imm32
- if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
  - <if wback then >= true
  - OP = add
  - then_val = R[n]+imm32
  - else_val = R[n]-imm32
- MemA[address,4] = if BigEndian() then D[d+r]<63:32>else D[d+r]<31:0>;
  - <if wback then >= false
  - OP = BigEndian()
  - then_val = D[d+r]<63:32>
  - else_val = D[d+r]<31:0>
- MemA[address+4,4] = if BigEndian() then D[d+r]<31:0>else D[d+r]<63:32>;
  - <if wback then >= false
  - OP = BigEndian()
  - then_val = D[d+r]<31:0>
  - else_val = D[d+r]<63:32>

Second mining rule:

for counter = (start) to (end)

Here are some example matching lines and extracted fileds matching this template:

- for r = 0 to regs-1
  - iteration_start = 0
  - iteration_end = regs-1
- for e = 0 to elements-1
  - iteration_start = 0
  - iteration_end = regs-1

Third set of mining rules:

- *if (single_regs) then*
- *else*

These templates are examples for handling special cases where certain lines cannot be grouped together with others to match a common template and need individual treatment. These templates match the following lines:

- if single_regs then
- else

Fourth mining rule:

lval = rval

This is an important template because it not only matches the psedocode lines but also is used to process the extracted fields from other templates. Every computational line is convered into a line matching this template and code is generated to compute the 'rval' and saved to 'lval'.

- Here is an example to show how this template is used on an extracted field,
  else_val = R[n]-imm32 is one of the extracted fields after applying the first mining rule.

- These are the example lines that come directly from the CPU manual psedocode:
  - MemA[address,4] = S[d+r];
  - address = address+8;

Fifth set of mining rules to handle Load/Store operations:

- *MemA(addr) = (Register) // For Store*
- *(Register) = MemA(addr) // For Load*
- *(S[d+r])*
- *D[d+r]<31:0>*
- *D[d+r]<63:32>*

Here are some examples of lines matching these templates and their extracted fields using the templates:

- MemA[address,4] = S[d+r];
  - Operation = Store
  - Memory Address = address
  - Source Register = Singleword register
- MemA[address+4] = D[d+r]<63:32>
  - Operation = Store
  - Memory Address = address+4
  - Source Register = Top 32-bits of the doubleword register

Instruction	Description
VSTM	Vector Store Multiple (Increment After, no writeback)
VSTM	Vector Store Multiple (Increment After, writeback)
VSTR	Vector Store Register
VSTM	Vector Store Multiple (Decrement Before, writeback)
VPUSH	Vector Push Registers
VLDM	Vector Load Multiple (Increment After, no writeback)
VLDM	Vector Load Multiple (Increment After, writeback)
VPOP	Vector Pop Registers
VLDR	Vector Load Register
VLDM	Vector Load Multiple (Decrement Before, writeback)

Table 3.1: ARM Architecture - VFP Load/Store Instructions.

3.6.2 Auto-generating The Crack Code

The parser 3.6.1 generates statements of type 'left_value = right_value'. The 'left_value' and 'right_value' can be simple expressions like R[n], imm32, address, S[d+r] or complex expressions like R[n] + imm32, R[n] - imm32, address + 8.

As shown before, generated simple and complex expressions have operands like R[n], imm32, address. These operands are of two types - for first type of operands, the values come from the instruction itself. Examples include imm32, R[n] etc. And the second type of operands are the temporary variables, like 'address'. In our implementation, we use a hash table to keep track of a mapping of temporary variables and the corresponding temporary registers to hold their values.

Here are some examples of how a statement of type 'left_value = right_value' is used to auto-generate the scuops.

- address = R[n]+imm32

  - CrackInst::setup(rinst, iAALU, OP_S32_ADD, RN, 0, imm32, LREG_TMP2, 0, 0, 0) // LREG_TMP2 is a temporary register which is anonymous for 'address', the destination register.

- address = R[n]+R[m]
  - CrackInst::setup(rinst, iAALU, OP_S32_ADD, RN, RM, 0, LREG_TMP2, 0, 0, 0) // LREG_TMP2 is a temporary register which is anonymous for 'address', the destination register.

- S[d+r] = MemA[address, 4]
  - CrackInst::setup(rinst, iLALU_LD, OP_U32_LD_L,LREG_TMP3 , 0, 0, DV, 0, 0, 0)

VSTM

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);     NullCheckIfThumbEE(n);
    address = if add then R[n] else R[n]-imm32;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r]; address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian() then D[d+r]<63:32>else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian() then D[d+r]<31:0>else D[d+r]<63:32>;
            address = address+8;
```

VLDR

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    NullCheckIfThumbEE(n);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    if single_reg then
        S[d] = MemA[address,4];
    else
        word1 = MemA[address,4]; word2 = MemA[address+4,4];
        // Combine the word-aligned words in the correct order for current endianness.
        D[d] = if BigEndian() then word1:word2 else word2:word1;
```

VLDM

```
    if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    NullCheckIfThumbEE(n);
    address = if add then R[n] else R[n]-imm32;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
    for r = 0 to regs-1
    if single_regs then
        S[d+r] = MemA[address,4]; address = address+4;
    else
        word1 = MemA[address,4]; word2 = MemA[address+4,4]; address = address+8;
        // Combine the word-aligned words in the correct order for current endianness.
        D[d+r] = if BigEndian() then word1:word2 else word2:word1;
```

Table 3.2: ARM Architecture - Psedocode Specification From The CPU Manual.

Chapter 4

Random Instruction Test Sequence Generation

This chapter describes a framework for regression testing of the ARM processor emulator by generating `RIT` (Random Instruction Tests). Human intervention is limited to copying the assembly syntax of all the instructions to an instruction macro library. An instruction is picked at random and an intelligent parser expands the instruction with all possible combinations of the options from its assembly syntax. By testing all the instructions thus generated, it can be guaranteed that all the code paths associated with that instruction are exercised.

4.1 Related Work

Instruction Randomization Self Test (IRST) [4] is a test methodology to acheive stuck-at-fault coverage for embedded processors. IRST has a test hardware and a test software function. Test hardware function performs three functions - it prompts the test software to provide a pseudo random sequence of instructions, monitors the data to determine if the behavior indicates faulty logic and it provides a source of randomized seed data which the test software uses to randomize register operands. Test software executes a variety of control and data path logic in the processor core. Test software provides observability on the central bus as the program executes.

Random test program generation for the class of microprocessors embedded inside a SOC (System-on-a-chip) [6] uses an evolutionary paradigm to generate automatic test programs and provides for auto-updating internal parameters of the optimizer. This work proposes an instruction library and a genetic algorithm to evolve random test programs. Test programs are

internally represented as directed acyclic graphs and evolved to maximize the attained fault coverage.

System Validation (SV) [9], a post silicon functional validation methodology to meet aggressive deadlines, is based on two approaches, random instruction testing (RIT) and direct testing. Using Compatibility validation (CV), the goal is to validate components (processors and chipset) at the platform level by using commercially available operating systems, application software and hardware in a manner that is consistent with the usage by the end user.

4.2 Our Approach To Generating RITs

Fig. 4.1 Shows a sequence of steps involved in regression testing of the emulator.

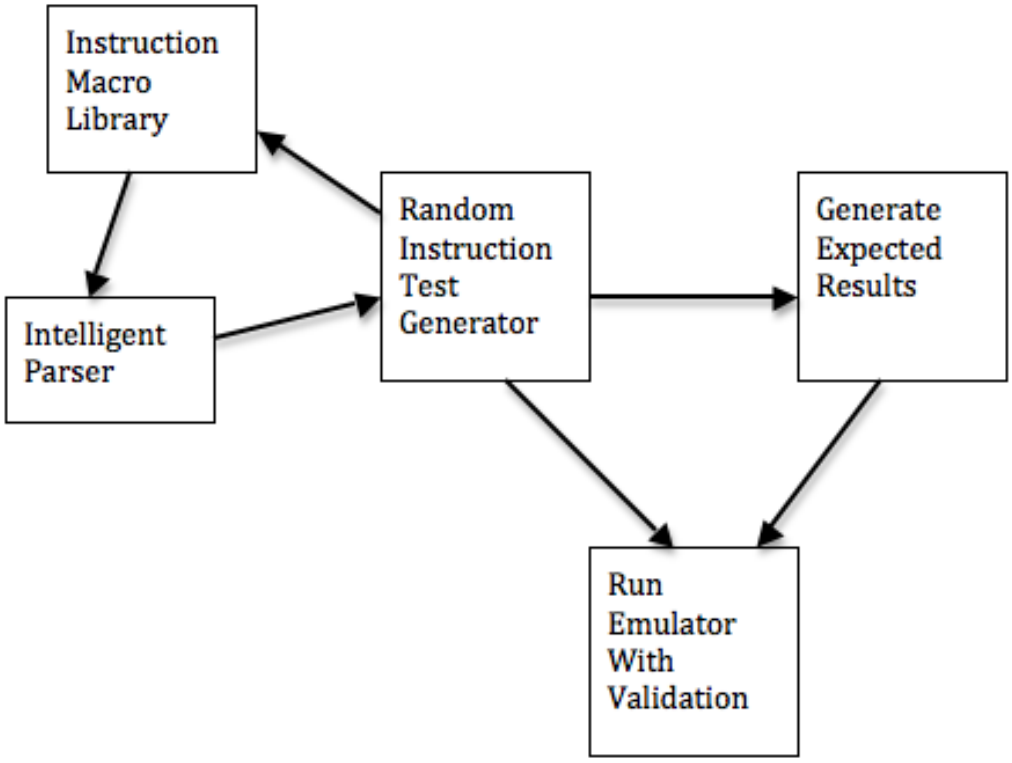

Figure 4.1: End-to-end Sequence Of Steps For The ARM Emulator Testing.

To ensure that an instruction in the ARM emulator is exercised through all code paths, the

following componets of its execution must be tested:

- Instruction Type i.e., the same instruction execution in ARM, Thumb-32 and Thumb-16 ISAs.
- Setting condition codes, if applicable for that instruction.
- Conditional execution.

To accomplish these goals, we make the following contributions:

- An Instruction Macro Library.
- An Intelligent Parser.
- A Random Test Generator.

The following sections describe these functionalities in detail.

4.3 Instruction Macro Library

Macro Library contains assembly syntax of all the instructions as specified in the CPU specification manual. Human intervention is limited to copying and pasting assembly syntax to this file.

The instructions are grouped by their type of instructions, see section 1.2.6 for all the instruction types.

The library provides methods to return a random instruction from a class of instructions. The caller functions have the option to provide an instruction class. The following shows some of the entries from the Instruction Macro Library.

```
ARM_data_processing_register =
    ["AND{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}",
    "EOR{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}",
    "SUB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}",
    "RSB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}",
    "ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}",
```

```
    "ADC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}"}

ARM_data_processing_immediate =
    ["AND{S}<c> <Rd>,<Rn>,#<const>",
    "EOR{S}<c> <Rd>,<Rn>,#<const>",
    "SUB{S}<c> <Rd>,<Rn>,#<const>",
    "ADR<c> <Rd>,<label>"}

ARM_load_store_word_unsigned_byte =
    ["STR<c> <Rt>,[<Rn>{,#+/-<imm12>}]",
    "STR<c> <Rt>,[<Rn>],#+/-<imm12>",
    "STR<c> <Rt>,[<Rn>,#+/-<imm12>]!",
    "STR<c> <Rt>,[<Rn>,+/-<Rm>{, <shift>}]{!}",
    "STR<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}",
    "STRT<c> <Rt>, [<Rn>] {, +/-<imm12>}"]
```

4.4 Random Test Generator

The Random Test Generator generates Random Instruction Sequences using Instruction Macro Library and Intellignet Parser.

For a given run, it generates two programs. A first program with assembly instructions encoded with ARM 32-bit instructions and a second program with assembly instructions encoded with Thumb 32-bit and 16-bit instructions.

The programs are generated with three sections. The first section contains ARM assembler directives to control the type of instructions being generated. The second section contains the instruction sequences and the third section contains data declarations that can be loaded into registers and used throughout the program.

```
Directives to generate ARM instructions

  .syntax unified
```

```
  .global main

Directives to generate Thumb instructions

  .syntax unified
  .global main
  .code 16
  .thumb_func

An example third section of the program that declares data

data1:
   double 2.0, 3.0

To load this data,

    adr r0, data1
    vldm r0, {d10-d11}
```

The Random Test Generator has a main loop which aims to generate about 1000 instructions for each of the ARM and Thumb test programs.

This module provides the Instruction Macro Library with an instruction class and requests for a random instruction. And the Library provides the random instruction to the intelligent parser. Parser, in turn, expands the assembly instruction with all the options and generates multiple instructions to cover all code paths associated with that instruction. Intelligent parser, thus, returns two sets of instructions for every request made by the Random Test Generator. The following shows psedocode of the Random Test Generator execution loop.

```
total_inst_count = 0

while total_inst_count < 1000
```

```
Pick an instruction class at random
Request Instruction Macro Library
Library selets an instruction and notifies Intelligent Parser
Intelligent Parser generates two sets of instructions

    set1 with ARM instructions
    count1 = number of instructions generated in set1

    set2 with Thumb instructions
    count2 = number of instructions generated in set2

total_inst_count += (count1 or count2 whichever is more)
```

4.5 Intelligent Parser

The assembly syntax of the ARM instructions as specified in the ARM specification manual follows a common pattern which allows us to parse the instruction for various fields. Here are some example assembly syntax of the instructions.

```
Format: INST<{s}><c> operands

ADD{s}<c> <Rd>,<Rn>,<Rm>{<shift>}
EOR{s}<c> <Rd>,<Rn>,<Rm><type><Rs>
SUB{s}<c> <Rd>,<Rn>,#<const>
STR <Rt>,[<Rn>[#+/-imm12]
POP <registers>
VLDM <Rn>!<list>
```

As shown in the instruction format above, the first field of the test vector is the name of the instruction. Option $\{s\}$ is a flag to set the condition codes. $<c>$indicates conditional execution. The rest of the operands indicate registers, shift values and immediate constants.

Option {s} produces two versions of the instruction - first instruction to set the condition codes and the second instruction not to set the condition codes. For those instructions that have no {s} flag option, only one instruction is generated, without {s} option.

<c > indicates that the instruction is capable of executing conditionally. Here we aim to test that the instruction works correctly for all 14 condition codes specified in Table 1.12. We also test the combinations where the condition codes do not satisfy the condition. This generates 14*2 = 28 instructions.

Registers with syntax <REG >Ex: <Rd >, <Rn >, <Rm >, <RS >: Parser picks any of the registers R0-R11 at random (PC register R15 is included for Rd, destination register selection).

<type >indicates the type of shift. One of the supported shifts (ASR, LSR, ROR, RRX etc.) is picked at random.

<shift >, <imm12 >, <const >generate a random value in the range of values supported by that field.

B <label >, BL, BLX, the label is generated with addresses of one of the instructions generated by the Random Test Generator. The address can belong to one of the previous or later instructions. Random Test Generator adds intelligence to not result in an infinite loop while generating these instructions.

4.6 Generating Random Instruction Tests (RITs)

This section describes how to generate Random Instruction Tests (RITs) using the framework.

```
shell# ls MacroLibrary.py MacroParser.py generateRIT.py
generateRIT.py MacroLibrary.py  MacroParser.py
shell# ./generateRIT.py

Usage:   generateRIT.py out_RIT_file

shell# ./generateRIT.py test_rit.s
```

Alternatively, here is a Makefile which generates a RIT file and compiles it to make a binary (Run this makefile on an ARM computer to generate ARM binary).

```
shell# more Makefile
rit.s: rit
        objdump -d rit > rit.s

rit: test_rit.s
        gcc -g -static test_rit.s -o rit

test_rit.s:
        ./generateRIT.py test_rit.s

clean:
        rm test_rit.s rit rit.s

shell# make clean
rm test_rit.s rit rit.s
shell# make
./generateRIT.py test_rit.s
gcc -g -static test_rit.s -o rit
objdump -d rit > rit.s
shell#
```

Chapter 5

Conclusion

Processor emulators allow researchers and designers evaluate design ideas quickly at no extra cost. To evaluate research ideas at MASC lab, we are developing an ARM processor emulator to execute ARM binaries.

5.1 Contributions

First, we developed a validation infrastructure to identify any functionality gaps in the emulator. Second, by implementing the needed functionalities and fixing problem areas, we are able to execute complete ARM binaries through the emulator. Finally, we developed a methodology for facilitating regression testing of the emulator.

5.2 Results

As a result of this work, we are able to:

- Execute ARM binaries through the emulator.
- Acheive a 100x speedup of the validation process since our original design.
- Reduce about 20k lines each from Thumb-16 and Thimb-32 Crack Implementation.
- Implement support for some system calls.

- Use a CPU spec mining technique to auto generate code for a class of instructions (VFP Load/Store instructions).
- Implement a framework to generate Random Instruction Sequence Tests to facilitate regression testing of the emulator.

5.3 Future Work

With validation framework and emulator functionalities in place, we can run longer programs including SPECInt and PARSEC benchmark programs through the emulator.

Another area of work would be to see if CPU spec mining can be used to auto generate Crack code for x86, SPARC instruction sets.

Crack code for ARM/Thumb instructions use `scinsts`. As part of this work, we would like to optimize the number of `scinsts` for the instructions.

Appendix A

Running the ARM Emulator

```
mada0:~/project/build$main/scmain
Usage:
    main/scmain stack_start stack_buffer_start stackdump.bin binary
    <arm_log  syscall_trace>

Mandatory arguments -

    stack_start                 Stack Pointer
    stack_buffer_start      Start address of the stack dump
                                    (SP && 0xFFFF0000)
    stackdump.bin           Path to the stack dump binary file
    binary                        Application binary

Optional arguments -

    arm_log                   Path to the instruction trace file
    syscall_trace           Path to the system call trace file

-------------------------------------------------------------------

Here is a step-by-step example to show how to run the emulator.

1) Clone the Project
```

```
> CONNECT TO YOUR X86 Linux machine (ubuntu or archlinux are fine)
shell> mkdir ~/projs
shell> cd ~/projs
shell> git clone ESESC repo (instructions may vary)
```

2) Now, you have cloned the project. Now, make a build directory to compile the code.

```
shell> mkdir ~/build
shell> cd ~/build
shell> cmake -DCMAKE_HOST_ARCH=x86_64 -DENABLE_SCQEMU=1 -
           DESESC_QEMU_ISA=armel -DCMAKE_BUILD_TYPE=Debug
           ~/projs/esesc/
shell> make scmain -j 16
```

To run the emulator, call

```
shell> ./main/scmain
```

and here is the message that you should get:

```
shell> Usage:
        main/scmain stack_start stack_buffer_start stackdump.bin
        binary <arm_log syscall_trace>
```

We have the emulator setup working. We will come back to running the emulator.

3) Now, let us see how to compile an example Application (hello.c) for ARM.

Open a new terminal.

```
> ssh YOUR ARM Machine (something like pandaboard or Samsung
   chomebook)
armshell> cd build/
```

```
armshell> mkdir kernels
armshell> cd kernels
armshell> vim hello.c

#include<stdio.h>
int main() {
  printf("hello world \n");
  return 0;
}
armshell> gcc -g -static hello.c -o hello
```

Now, we have the simulator (scmain) and the application (hello) compiled for ARM.

We should be able to call the "scmain" and pass the "hello" to start the emulation. However, since we want the emulator to match exactly the native execution, we try to reproduce the stack memory. It means that we dump the stack from the native ARM machine while running the application, and then pass it to the emulator.

4) Dump stack memory to a file

```
armshell> python2 ~/projs/esesc/conf/generateARMLog.py hello
Reading symbols from /home/cmpe202/build/helloworld_sccore/hello...
(no debugging symbols found)...done.

(gdb) b *_start
b *_start
Breakpoint 1 at 0x80f0
(gdb) run
run
Starting program: /home/cmpe202/build/helloworld_sccore/hello

Breakpoint 1, 0x000080f0 in _start ()
(gdb) p /x $sp
p /x $sp
```

```
$1 = 0x7efffcb0
(gdb) stack starts at 0x7efffcb0
dump binary memory hello.stack_0x7efffcb0 0x7efff000L 0x7fffffff
dump binary memory hello.stack_0x7efffcb0 0x7efff000L 0x7fffffff
(gdb) scmain usage:

./main/scmain 0x7efffcb0 0x7efff000L hello.stack_0x7efffcb0 hello

 Now, we have the memory dump of the stack.

    while we are on the ARM computer, objdump command in
    Linux is very useful:

    armshell> objdump -d hello > hello.s

    hello.s contains the assembly of the application now. You will
    need it for debugging to see what instruction is at what
    address, etc.

5) Now, to run the emulator, go back to the previous terminal where
   you logged in to your X86 Linux and compiled scmain.

 shell> ./main/scmain 0x7efffcb0 0x7efff000L hello.stack_0x7efffcb0 hello
```

Bibliography

[1] Gdb/mi- debugging with gdb. http://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html.

[2] ARM ELF file format- ARM DUI 00101-A. http://infocenter.arm.com/help/topic/com.arm.doc.dui0101a/DUI0101A_Elf.pdf, November 1998.

[3] ARMv7-M architecture reference manual. http://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARMv7-M_ARM.pdf, February 2010.

[4] Ken Batcher and Christos Papachristou. Instruction randomization self test for processor cores. In *Proc. VLSI Test Symposium*, pages 34–40, 1999.

[5] Steve Chamberlain. libbfd: The binary file descriptor library. http://www.gnuarm.com/pdf/bfd.pdf, April 1991.

[6] M. Sonza Reorda G. Squillero F. Corno, G. Cumani. Fully automatic test program generation for microprocessor cores. In *Proc. Design, Automation and Test*, pages 1530–1591, 2003.

[7] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator. http://sesc.sourceforge.net, January 2005.

[8] TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification. http://www.cs.princeton.edu/courses/archive/spr09/cos217/reading/elf.pdf, May 1995.

[9] Robert Mauri Tommy Bojan, Igor Frumkin. Intel first ever converged core functional validation experience: Methodologies, challenges, results and learning. In *Proc. Eighth International Workshop on Microprocessor Test and Verification*, 2008.

[10] Jyh-Charn(Steve) Liu Weiqin Ma. Alessandro Forin. Rapid prototyping and compact testing of cpu emulators. In *Proc. Rapid System Prototyping*, 2010.