Generating Tests to Verify Machine Descriptions

Abigail Lyons

March 30, 2018

Abstract

With the ending of Moore's Law, people are increasingly turning to other ways to leverage performance. x86 has been an industry standard for decades, but in recent years RISC ISAs like ARM and specialized hardware like FPGAs and GPUs are becoming widely used. Because of this, the already labor-intensive task of porting an operating system to different hardware needs a better solution. One possible approach to making porting easier is code synthesis through machine-level and semantic descriptions of hardware. Therefore, it is crucial that machine descriptions accurately reflect the hardware that they attempt to describe. Given a machine description, we generate a suite of tests that, when executed on the target platform, provides assurance that the description accurately reflects the platform's expected behavior.

Contents

1 Introduction

1.1 Motivation

Specialized processors are proliferating now more than ever. New applications, from smartphones to self-driving cars and delivery drones, are increasingly requiring higher performance, and we can no longer rely on Moore's Law to keep up with this. A good example of this is the single-core performance of iPhones. One study measures this relative to the Intel Core i7-6600U, which is given a baseline score of 4000; twice the score indicates a 2x increase in performance. According to this study, the iPhone 4S has a score of 284 and the iPhone 5, the very next generation, has a score of 754 which is close to triple the performance. However, the iPhone 7 has a score of 3387 while the iPhone 8 has a score of 4217, which is still an increase in performance but not by as much–only by a factor of 1.25[4].

Since neither battery power nor processor performance are growing as quickly as desired, the result is that we must have hardware that are highly tailored to their applications to best optimize performance. GPUs, FPGAs, TPUs, and exascale supercomputers are only a few examples of specialized hardware. These new hardware platforms need operating systems, but existing operating systems are challenging and tedious to port.

One attractive solution to this problem is to devise a method to describe hardware platforms in enough detail that operating systems could be synthesized to run on these new platforms. There are many different ways to describe hardware, from low-level hardware descriptions in Verilog to higher-level semantic machine descriptions in λ-RTL. Ideally, the machine description should be at the semantic level, providing enough information about the changes happening to the hardware at each instruction without requiring the programmer to write code for nit-picky

details like the bitwise operations involved in adding two numbers together.

Writing a complete machine description is challenging–not as time-consuming as manually porting every piece of machine-dependent software, but challenging nonetheless. As such, it's important to be reasonably confident that the resulting description is correct. A good way to do this is to generate tests to run on both the real hardware and a machine description interpreter, and then compare the resulting machine states to determine if the actual results from the hardware are what the description says they should be. This testing method is the focus of the rest of this paper.

1.2 Contributions

This paper presents a testing methodology for machine descriptions that generates the test suite from scratch. First, it describes a design and implementation of a program that can initialize equivalent states and run equivalent instructions on a virtual test machine and a machine description language interpreter and directly compare the resulting states. Second, it details a method of generating randomized tests from scratch, pruning the tests for validity, and selecting important corner cases to catch common errors. And lastly, it shows that this testing method reveals programmer errors in a machine description of an ARM processor.

The remainder of this paper is structured as follows:

Chapter 2 provides some background on the key infrastructure pieces on which this work depends.

Chapter 3 contains an overview of related research in machine descriptions, test generation, and their intersection.

Chapter 4 is a detailed description of the test infrastructure's design and implementation.

Chapter 5 discusses the errors that were found from running the testing program on a sample machine description.

Chapter 6 concludes with a brief summary of this paper.

2 Background

2.1 Machine Descriptions

A machine description language is any language that describes hardware. Hardware description languages (HDLs), such as Verilog, describe properties of machines and are commonly used when designing digital circuits, and as such they fit this general definition. However, in this paper I will instead focus on semantic machine description languages, which describe at a high level how each instruction changes the state of a machine [13]. A semantic machine description language consists of two main components. The first is a state definition, which defines what types are available, the number of registers and their types, flag bits, the amount of memory available, and possibly even more information describing the machine's state. The second component is the instruction definitions, which define instructions like shift right, add, etc. in terms of how the machine's state changes. When testing a completely accurate machine description, the values in the defined state should always match the actual state of the machine it's describing.

One example of a buggy machine description is the following definition of add. Ignoring flag bits and overflow behavior, this is supposed to add the contents of registers `r1` and `r2` and store the result in `r2`, but it actually just assigns `r2` to whatever is stored in `r1`:

```
add r1, r2:
```

```
 r2 ← r1
```

This obviously doesn't produce the intended result, and this bug can be easily found by running one add instruction in isolation with any nonzero value for `r2`. Compare this with the following description for `add`:

```
add r1, r2:
 r3 ← r1 + r2
 r2 ← r3
```

This now stores the intermediate from `r1` + `r2` inside `r3`, another register, and then assigns `r2` to the value in `r3`. This is less obviously wrong, because running one add instruction in isolation will not show any bugs just from examining the result in `r2`. However, the value that was originally in `r3` has now been overwritten. This bug could be found in two ways. One is by using `r3` in a subsequent instruction which would then return an incorrect value. Another is by examining the machine's state right after the erroneous add instruction is run, which would reveal that `r3` had unexpectedly changed.

2.2 Cassiopeia

Cassiopeia is a new semantic machine description language developed by the PRINCESS Evolving Operating Systems Project at Harvard[2]. A description in this language consists of an specification of a machine's state and a specification of operations in terms of the machine's state. For any given processor, much of this information can be found in its documentation to varying degrees of accuracy. Someone who wants to use Cassiopeia would then need to manually translate the documentation

into Cassiopeia code.

Cassiopeia has a few primitive types, including a length n bitvector, logical integer, logical boolean, pointer, and others. All other types used in the machine description must be defined by the user in terms of already existing types. Following are some examples of these type definitions; the first line creates a type `word` that is equivalent to a bitvector of length 32, while the second line creates a type `register` that is a mutable location that must contain word, or a length-32 bitvector.

```
type word = 32 bit
type register = word loc
```

To describe a machine state, Cassiopeia has a `letstate` which names variables of given types, along with their initialized values, as part of the state. These statements can be used to represent condition bits like N (indicating a negative result on an ARM processor), general registers like R0 (which stores return values on an ARM processor), and memory, represented here as a single word to shrink the size of the state space.

```
letstate N:1 bit loc = 0x0
letstate V:1 bit loc = 0x0
letstate R0:register = 0x00000000
letstate R1:register = 0x00000000
letstate MEM:word loc = 0x0
```

Cassiopeia describes how instructions change the machine's state using a `defop` for each instruction. Here is an example of the logical shift left (LSL) operation

in ARM; note that `arm32_16touint(sh)` is a built-in Cassiopeia function that translates a 16-bit bitvector into an unsigned integer.

```
defop LSL Rd:register Rn:register sh:halfword {
 sem = [
  let sh_int:int = arm32_16touint(sh) in
  let dRn:word = *Rn in
  let shgtz:bool = sh_int > 0 in
  if shgtz then
   let shRn:word = dRn << sh_int in
   let iszero:bool = shRn == 0 in
   *N <- shRn[31:32];
   *Z <- iszero ?  0x1 :  0x0;
   *Rd <- shRn
  else
   let iszero:bool = dRn == 0 in
   *N <- dRn[31:32];
   *Z <- iszero ?  0x1 :  0x0;
   *Rd <- dRn
 ]
}
```

The first line contains the name of the operation, followed by its named inputs and their types; the first two inputs are registers, while the third input is a halfword, or a bitvector of length 16. The second line declares the beginning of the operation's semantics. Temporary variables are defined using `let...in` statements. Assignment of state variables requires dereferencing them using $\star$ and then assigning them

using <−. Logical operations like +, >>, etc. are all built into the language, allowing the programmer to focus on the changes to the machine state instead of having to completely re-implement an add over individual bits.

Ultimately, what the above example aims to do is check whether the given shift value is greater than zero. If it is, it shifts the value in `Rn`, sets the `N` flag to the highest bit, and sets `Rd` to the shifted value; otherwise, it sets the `N` flag to the highest bit of `Rn` and sets `Rd` to `Rn`. In both cases, the `Z` flag is set to 1 if the result is 0, and set to 0 otherwise. Note that, compared to this brief statement of pseudocode, the actual Cassiopeia code appears to be much more verbose. This is because Cassiopeia does not yet allow multi-step statements, such as `if sh_int > 0 then`, where an implicit step calculates the intermediate anonymous boolean `sh_int > 0`.

In summary, a Cassiopeia file consists of type definitions, followed by state definitions and then instruction definitions.

There are two useful tools for writing Cassiopeia code; one is a parser that builds an abstract syntax tree out of a machine description, while the other is an interpreter that runs a given Cassiopeia program using a machine description. The machine description testing program delineated in this paper uses the interpreter to run instructions and output the expected state of the machine. The Cassiopeia program that the interpreter uses consists of statements written in the same form as the following example:

```
(ADD R0 R0 0x0000abcd)
(ADD R1 R0 0x00001111)
```

Assume that all registers and other components of state are initially set to `0x00000000` and an `ADD` function is correctly defined. If the above program interprets success-

fully, the interpreter will output that `R0 = 0x0000abcd`, `R1 = 0x0000bcde`, and all other components of state are equal to `0x00000000`.

2.3 ARM v6 Architecture

The ARM has 31 general-purpose registers, all of which are 32 bits wide, the first 16 of which are the user mode registers available to all unprivileged code. The first four registers (R0-R3) hold function call arguments, while R4-R8 do not have any designated purpose. R13 is typically the stack pointer, R14 is the link register which holds the address of the next subroutine call instruction, and R15 is the program counter which is two instructions after the instruction being executed. The ARM also has a current program status register (CPSR), which contains information such as endianness, interrupt masks, and other condition flags. Most importantly for this application, in the highest 4 bits it holds condition code flags representing a **N**egative, **Z**ero, **C**arry, and o**V**erflow result[15]. For this purpose, I model a partial state consisting of the first 9 registers, the four condition code flags, and one 32-bit piece of memory. This restricts which instructions I can model, making instructions that change the instruction pointer register (like branch) impossible to represent.

Because ARM is a Reduced Instruction Set Computer (RISC)[15], it has fixed-length instruction fields, making it ideal for modeling in Cassiopeia. Most ARM instructions are capable of setting condition flags by ending the instruction name with an s (i.e. `adds` instead of `add`); noteworthy exceptions are the branch instruction and its variants, which never set condition flags, and the compare instruction and its variants, which always set all of the condition flags. As such, modeling the condition flags in Cassiopeia is an important to getting the semantic instruction definitions exactly right.

To run one test of a machine description of ARM, the same instruction must be

run with the same initial state in both the Cassiopeia interpreter and on an ARM processor. Only if their final states are the same does the test pass.

2.4 Simulating ARM

The program explained in this paper was not run on the physical ARM processor that the Cassiopeia machine description is written for. Rather, it was run on a fixed virtual platform (FVP) that intends to model the underlying state of an ARM v7 processor and memory[3]. This differs from an emulator, which attempts to mimic the outward behavior of its target without necessarily mimicking the underlying state. For this application, an emulator that is not a simulator is not sufficient, since the tests rely heavily on extracting that underlying state.

The ARM FVP can easily be booted on an x86 machine running Ubuntu 17.10, and it can share the host's filesystem, making communication between programs on the FVP and the host much simpler. For purposes of this paper, I will continue to treat the ARM FVP as though it accurately represents a real ARM v7 processor, even if it has not been proven to be completely correct.

2.5 Test Generation Challenges

Because testing each instruction definition in isolation does not guarantee a correct machine description, we must introduce numerous more instructions to determine that each instruction actually being tested is correct. This leads to the state explosion problem that many testing problems face: as the number of instructions in a program, or the number of definitions in a machine description, increases, the number of tests required to guarantee correctness grows exponentially[6].

However, even when testing instructions just in isolation, the state space is still too big to test exhaustively. Suppose there is a very simple Turing-complete ma-

chine description for a 32-bit processor that just consists of load and store. For either of those instructions, the first register can contain any value between 0 and $2^{32}-1$. The second register can contain any valid address. The third value, an immediate representing the offset from the address stored in the second register, can contain any number of values between the address and the end of valid memory[15]. Assuming that the size of user-accessible memory is around 4 MB, this means that exhaustively testing the load instruction for two fixed registers alone would require $2^{32} \times \sum_{x=0}^{4M-1} x(4M-x) = 2^{32} \times 1.067 * 10^{19} \approx 4.5 \times 10^{28}$ tests.

To give an idea of what this means to someone trying to test their machine description, running 10000 tests on a computer with a 2.7 GHz processor and 8 GB RAM takes approximately 2 minutes and 30 seconds; running all the tests for the load instruction for those two predefined registers would then take $2.5 \times 4.5 \times 10^{24} \approx 1.1 \times 10^{2}5$ minutes, or 2.1×10^{19} years! Even though exhaustive testing in isolation is just as hopeless as exhaustively testing combinations of instructions, I opted to start with a new state for each instruction rather than track the changes in a single state across multiple instructions.

3 Related Work

3.1 Machine Descriptions

Machine descriptions based on register transfer lists (RTLs) have been frequently used for building retargetable compilers, but RTL languages are typically verbose and difficult for a human reader to understand. λ-RTL was developed to make machine descriptions easier to use by adding some human-readable shortcuts, such as allowing implicit fetches and assignment to bitvector slices. Ultimately, the goal of this is that machine descriptions will be more widely used and reused to build

other retargetable tools, not just compilers but also simulators, assemblers, linkers, and debuggers[14].

One interesting application of semantic machine description languages is verification of high-performance cryptographic assembly code. The Vale language was developed for this purpose. A programmer first describes the syntax and semantics of their architecture in Dafny, and then uses Vale to describe how the architecture's state is manipulated. The Vale tool generates an abstract syntax tree (AST), and builds proofs that are verified using a SMT solver[5]. Cassiopeia has some similarities to Vale, since it primarily describes how state is manipulated and also ultimately generates an AST.

Another application of machine description languages is testing individual components of a compiler, such as the architecture specifications used by code generators. GCC has its own RTL specification which describes the semantics of each instruction, but it cannot be assumed to be correct. Whenever a semantic difference is found in the testing process, it must be manually verified that the RTL semantics match the corresponding assembly instruction[9].

Notably, all these applications assume that the programmer provides an accurate description of their machine or rely on manual verification.

3.2 Test Generation

There have been many attempts to address the path-explosion problem of testing with well-defined heuristics of generating tests. However, most of these involve using existing source code as tests, and using a combination of path exploration and pruning heuristics to figure out which tests are most effective.

One of the classic methods of generating tests is symbolic execution, which is designed to test a given program. It replaces the program's inputs with symbolic

variables that have no constraints. It then gradually constrains the inputs more and more as it attempts to do two things: hit every possible line of executable code, and check each line against all possible input values. To address the path-explosion problem, it uses search heuristics to select which path to choose at each instruction step, taking into account the minimum distance to an uncovered instruction, among other things. Because the highest-coverage tests are run first, KLEE can then be run with a time limit and still provide confidence that many of the test cases have been covered [6]. Symbolic execution could be used to test machine descriptions, but the test program would need to be written ahead of time. Even if KLEE could run to completion on the test program, the test program itself cannot cover all possible instructions and operands.

One way that KLEE has since become more useful for systems applications is that it can do symbolic execution at the binary level, and various other optimizations can be made for low-level applications. For example, if testing that a pagetable works as expected, one would only need to explore a single pagetable location and test a few random values in the table [10]. This same reasoning can be used to remove redundant tests by reducing the number of times the same instruction is called with different operands.

A different approach to testing is fuzzing. First, the target's state is first filled with random data. Next, operation codes are chosen and combined with predefined operand values. The resulting tests are then pruned for redundancy and validity. The actual and expected states are compared at the end of execution of each test case[8]. This test generation is similar to the one used in this paper, though I used a much smaller sample of the machine state. The biggest difference is the objective; others have used fuzzing to discover whether an emulator behaves like the machine it claims to emulate[11], find differences in disassemblers[12], or find instances in which hardware behavior does not match its online documentation. Many of

the solutions to emulator, simulator, and hardware testing can also be applied to machine description testing.

One technique that has already been applied to machine description testing is assembly-disassembly. First, a machine description is passed into a special toolchain that chooses a sequence of instructions and data. It then produces an assembly file and a binary file based on the specified instructions. An independent assembler is applied to the assembly file to produce a second binary file, or an independent disassembler is applied to the binary to produce a second assembly file. When the resulting two assembly or binary files are compared, it is expected that they will be the same; a difference suggests an error in the machine description. One drawback of this approach is that assemblers and disassemblers frequently have bugs, and other research has been directed toward finding those bugs by comparing the output of disassemblers[12], so it is not immediately clear whether a found bug originates from the machine description or the assembler or disassembler. This also raises the question of which instructions, inputs, and values should be tested. In the same paper that introduced this approach, test generation involved building a tree out of each instruction's possible inputs, selecting the branch that covers the most untested inputs, and then using heuristics to decide on the values of those inputs[7]. This works well for RISC descriptions, but in more complex machines that allow a large number of inputs in their instructions, using a similar method would require implementing and running a complicated integer linear program solver to achieve similar results.

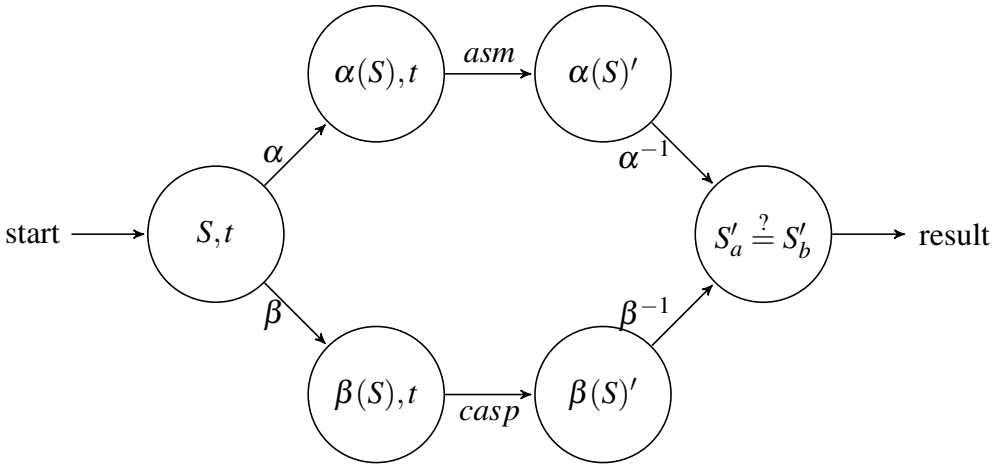

Figure 1: logical flow of information through a single test.

4 Design and Implementation

4.1 Problem Statement

My solution to testing machine descriptions is a program which generates initial states and test instructions, initializes the states and runs the instructions on both the test machine using inline assembly and the Cassiopeia interpreter, and compares the resulting states. The resulting states must match completely for the test to pass. Figure 1 is an illustration of this flow of information for a single test. S,t represent an initial state and test instruction from the test generator. α represents a function that translates the initial abstract state into its equivalent on the ARM processor, β represents a function that translates the initial abstract state into its equivalent in Cassiopeia, and their inverses undo those functions to get new abstract states in the same form as the initial state. *asm* and *casp* are functions that run the given test on translated states to get new translated states.

The diagram shows that there could be a few different sources of failure should S'_a and S'_b not match. One could be that α is not accurately translating the state,

and another could be that α^{-1} is not a perfect inverse of α; β is also subject to these possible errors. The one that's interesting for purposes of this paper, however, is that *asm* and *casp* do not do equivalent operations on the translated states, and the goal of the generated test states and instructions are to uncover these kinds of errors.

4.2 Implementation

Figure 2 is a road map for all components of the testing program, from the initial state generation step to the final test result, with the edges showing the information being passed from one part to the next. Completely machine-independent components are filled in crimson with white text, which is important to take note of when reusing code to test descriptions of other machines.

4.2.1 State and Test Generation

State itself is represented in the program as a C struct, containing an array of registers, a block of memory, and other components of processor state that are being measured. In the case of the ARM processor, I modeled 32-bit general registers 0 through 8, a single 32-bit memory location, and a 32-bit current program status register. Regardless of the individual parts that make up the state, the state generator fills the entire initial state struct with random bytes.

Next, the test generator picks an instruction, input registers, and immediate values to make up the test. I implemented two different test generators: one simply generates random instructions and picks random registers, with no memory of what tests were run before it. The other test generator picks instructions in a fixed order and sets certain requirements on its arguments. For example, it may require that all inputs be set to zero, or that two inputs for an ADD instruction have their highest

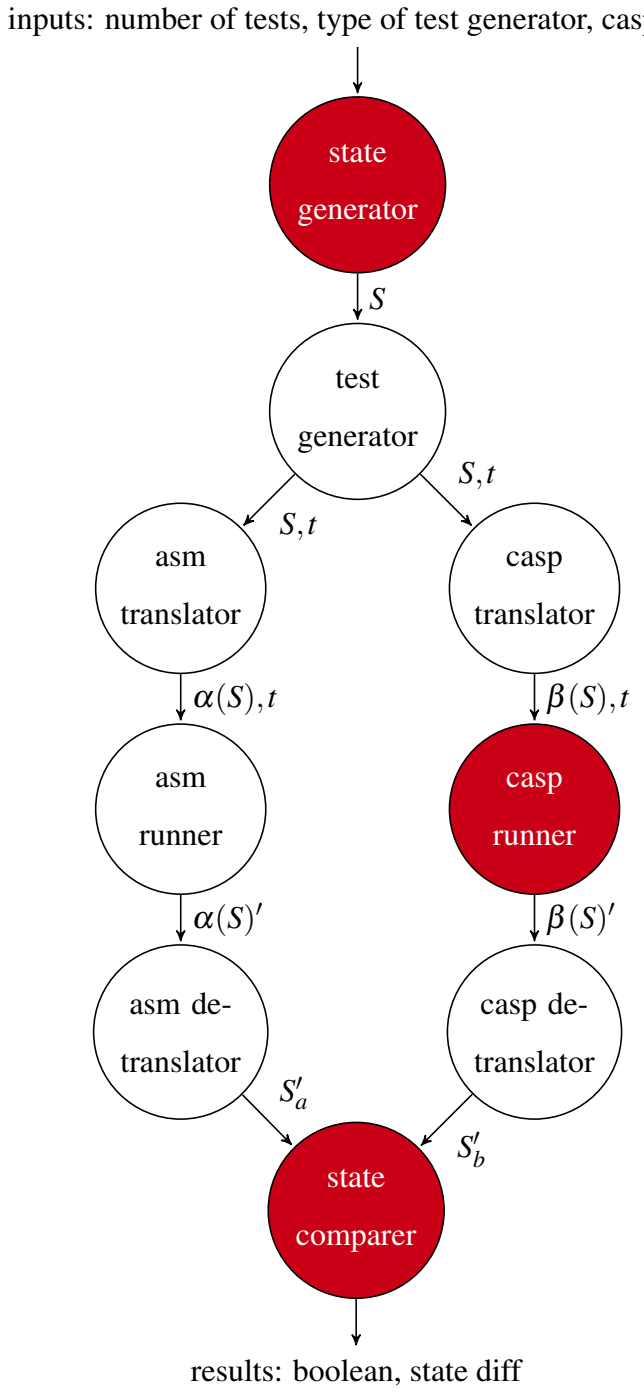

Figure 2: procedure for generating and running one test

bit set to deliberately cause an overflow. This test generator still picks registers at random, since it is unlikely that a bug lies within one specific register. (However, if all registers were being included in the abstract state representation, this should be adjusted so tests run on each register at least once.) Both test generators have the capability of modifying the initial state to fulfill processor requirements; for example, the ARM v6 documentation requires that in logical shift operations, the amount to be shifted must be between 0 and 31, so the third argument for those instructions must be changed. The random test generator always maintains the test's randomness, so even when part of the state is altered it takes on a random value within the valid range.

Then, the program splits off into two halves which can be run in any order: the hardware test and the Cassiopeia test. They can theoretically be run in parallel, but the ARM FVP does not support multithreaded programs or the fork system call, so I opted to run the hardware portion first.

4.2.2 Processor Test Execution

The state translation to assembly consists primarily of forcing state variables to their respective registers using statements like the following line, which requires that all operations on the 0th register in the abstract state take place on the 0th register of the processor.

```
register uint32_t r0_ asm("r0") = s->regs[0];
```

Next, the test is run via a volatile assembly instruction. Here's what an example instruction looks like:

```
asm volatile(
 "mov %[reg0],%[reg0]\n\t"
 "mov %[reg1],%[reg1]\n\t"
```

```
 "mov %[reg2],%[reg2]\n\t"
...
 "msr cpsr,%[f1]\n\t"
 "adds %[reg0],%[reg1],%[imm]\n\t"
 "mrs %[f1],cpsr\n\t"
 :  [f1] "=r" (s->cpsr), [reg0] "=r" (r0_),
   [reg1] "=r" (r1_), [reg2] "=r" (r2_),
...
 :  [imm] "r" (instr->imm), [reg0] "r" (r0_),
   [reg1] "r" (r1_), [reg2] "r" (r2_),
...
 :  "r0", "r1", "r2", "r3", ...
);
```

The first line declares that the following code is in an assembly language, and the volatile keyword disables certain optimizations from taking place, ensuring that the processor will behave exactly as specified. The processor's behavior was additionally verified using the DS-5 Debugger[1] to examine the processor state as the instructions were run. The following `mov` statements are no-ops that set the values of those registers to those in the abstract state's register variables. The `msr` instruction loads a value into the current program status register. Next, the actual test instruction is run, in this case an `add` instruction that adds register 1's value with the immediate and stores the result in register 0. The `mrs` instruction is the opposite of `msr`–it extracts the value of the current program status register. The first colon precedes a list of output operands; each variable is given a unique name in the assembly instruction, followed by `"=r"`, and then the variable's name in the surrounding C program. (Note that `instr` is a struct for storing an assembly test, and `imm` represents the immediate value passed into the instruction, if applicable.)

The second colon is similar, but instead begins a list of input operands. Lastly, the clobber list at the end of the instruction states which registers need to be restored after the inline assembly portion has finished running.

These inline assembly instructions cannot be written to dynamically select registers, so a different `asm volatile` call must be written for each combination of instruction, destination register, and source register. This quickly becomes too much code to write by hand, so I instead wrote a Python script that generates most of this code for me.

The results from running the test can easily be translated into an abstract state by setting all of the abstract state's register variables to the register variables that were initialized before:

```
s->regs[0] = r0_;
```

Special state components like the CPSR are handled separately, since the Cassiopeia interpreter does not handle finicky processor components like the program counter in the same way that the hardware does. In this case, since I only modeled the four uppermost bits, I simply applied a bitmask to zero out the rest of that register's contents.

4.2.3 Cassiopeia Test Execution

The Cassiopeia interpreter expects a special program file as an input, and it outputs a file detailing every step that was taken in producing the final state as well as the final state itself. As such, the abstract initial state must be translated and written into a program file that's readable by the interpreter. The Cassiopeia files written for these tests include initial values for each part of the state. However, they also include special `SET` operations that, unlike other operations, are meant just for dynamically initializing the state and do not actual model the processor. These `SET`

instructions are written to the program like such:

```
sprintf(buf, "(SETREG R%d 0x%08x)\n", i, s->regs[i]);
```

`i` in this case represents the register number being set. The buffer `buf` is then written to the output file. After all Cassiopeia state components are set to equal the initial state values, the instruction being tested is written to the end of the file. Reusing the assembly instruction from above, the correct Cassiopeia equivalent would be:

```
sprintf(buf, "(ADD R0 R1 0x%08x)\n", instr->imm);
```

The Cassiopeia interpreter is invoked using the following line, where arm.casp is the machine description, test.prog is the test program that was just written, and out.txt is the file to be parsed to find the final state.

```
system("./interp arm.casp test.prog > out.txt\n");
```

The output file out.txt is then read, checking for particular named pieces of state near the end of the output. When a CPSR bit is found, the individual bit corresponding to it in the abstract state is set to equal that bit. Similarly, when a register is found, its corresponding register is set to equal the value outputted by the interpreter. At the end, the remaining part of the state's CPSR is zeroed out to avoid any arbitrary conflicts between the unmodelled portions of the CPSR in test result comparison.

4.2.4 Test Result Comparison

Once both final states are received, they are simply compared using a call to `memcmp`. If a difference is found between the states, both states are printed out side-by-side with four bytes on each line; the programmer must then interpret the output based on the layout of the state struct.

5 Evaluation

The test program detailed above was run with a simplified ARM v6 machine description, which modeled registers 0 through 8, the upper 4 bits of CPSR, and four bytes of memory, and implemented the unsigned add, and, logical shift left, logical shift right, load, and store instructions. The test program was run on an ARM v7 FVP, and anywhere between 30 and 10000 tests were run in each iteration.

5.1 Bug Analysis

Many inconsistencies between the machine description and the hardware were found from simply running 50 completely random tests. The most common errors were inconsistencies in the CPSR flags, which were found in the add, shift left, and shift right instructions. These tests revealed that the C and V flags were sometimes being set in the Cassiopeia descriptions, but were never set by the processor. In addition, shift left and shift right were failing to set the N flag to the highest bit of the result.

The addition of fixed tests uncovered a few more inconsistencies. These tests covered basic corner cases for all instructions, such as setting the operand values to zero to ensure the zero flag was being set correctly, or deliberately setting the operands to be high or low to cause various behaviors like integer overflow, a zero result from nonzero operands, etc. One error that this testing method found was the behavior of shift right when the shift value is 0; the ARM v6 manual states that the result should then also be 0, but the FVP treated the shift right by 0 as a no-op other than setting the appropriate flags. One other error that was found was that setting the memory to equal zero in the load and store instructions would throw an error in the Cassiopeia interpreter, since a type error in the description caused it to interpret zero as an abstract value.

Overall, there are a few different sources of these errors: an imperfect version

match between the ARM documentation and the ARM FVP, misread instruction implementations in the documentation or poor organization of said documentation, and incorrect Cassiopeia code that halts the interpreter when certain branches are taken. All of these errors can realistically occur when writing machine descriptions.

5.2 Future Work

The program described above can be modified to be more programmer-friendly. Figure 2 demonstrates that most parts of the testing program built are machine-dependent, so a significant portion will need to be rewritten to test machine descriptions for other processors. As the problem is currently specified, many of these machine-dependent components are necessarily machine-dependent; for example, the test generator must be implemented with prior information about the kinds of corner cases that would be useful to cover for each instruction. This become particularly complicated when considering that more complicated architectures like x86-64 have variable numbers of arguments, which not only increases the number of test cases to cover and pieces inline assembly to write and compile, but also makes it difficult to know how large of a test state to generate. However, rethinking the infrastructure or the details of the test generator could reduce the number of machine-dependent components.

This testing program could also improve its test coverage by deliberately testing invalid instructions. In a brief experiment, I ran a few invalid instructions on the ARM FVP, including a load and store to address 0x0 and an add instruction with an invalid CPSR loaded prior to its execution. Surprisingly, the load and store instructions ran without complaint, while the add instruction caused the FVP to print a warning and hang. None of these were predictable or testable results, so I decided to stick with only testing valid instructions for purposes of this paper.

However, these invalid instructions might run in a more predictable way on a real ARM processor, in which case invalid tests should be run as part of the fixed test suite to ensure that both the Cassiopeia interpreter and the processor demonstrate errors.

Even though this program found many errors in a small machine description, it is currently unknown how many other errors may still be hiding. Other testing methods may be even more effective at uncovering bugs. For example, it is feasible that symbolic execution using a typical program written in assembly will result in a test that more closely mimics a real workload, so it is more likely to find errors that would impact a programmer. Not many solutions for testing machine descriptions have been found, so there are many opportunities for research in this area.

6 Conclusion

As the diversity of hardware increases, the problem of porting operating systems also grows in both frequency and tediousness. Machine descriptions can help to solve this problem–if it can sufficiently describe hardware, then it can aid in synthesizing operating systems for that hardware. The difficulty, however, lies in ”sufficiently describing hardware”, as machine descriptions are written by hand and typically undergo no testing. This paper describes a solution to this problem by generating a test suite comprised of both random and fixed tests for machine descriptions. When tested on a sample machine description, the test program revealed approximately one error for every instruction implemented, not to mention that the same errors often appeared in multiple instructions, which only underscores the importance of thoroughly testing machine descriptions before blindly using them.

7 Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisors, Professors Margo Seltzer and Stephen Chong, for their patient explanations of topics that I have no prior experience in, not to mention their immensely helpful feedback on my ideas, code, and writing. I would also like to thank everyone at Harvard working on the PRINCESS project for building the tools necessary for me to complete this research. And lastly, I'd like to personally thank Timothy for four remarkable years of friendship, and my mother, father, and brother for their lifelong support of my studies.

References

[1] Ds-5 development studio. https://developer.arm.com/products/software-development-tools/ds-5-development-studio/ds-5-debugger/overview. Accessed: 2018-03-30.

[2] Evolving operating systems. https://syrah.eecs.harvard.edu/darpa-princess-project?admin_panel=1. Accessed: 2018-03-30.

[3] Fixed virtual platforms. https://developer.arm.com/products/system-design/fixed-virtual-platforms. Accessed: 2018-03-30.

[4] iphone benchmarks. https://browser.geekbench.com/ios_devices/42. Accessed: 2018-03-30.

[5] BOND, B. Vale: Verifying high-performance cryptographic assembly code.

[6] CADAR, C. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.

[7] FERNANDEZ, M. Automatic checking of instruction specifications.

[8] GLAMM, B. Automatic verification of instruction set simulation using synchronized state comparison.

[9] HASABNIS, N. Checking correctness of code generator architecture specifications.

[10] MARTIGNONI, L. Path-exploration lifting: Hi-fi tests for lo-fi emulators.

[11] MARTIGNONI, L. Testing cpu emulators.

[12] PALEARI, R. N-version disassembly: Differential testing of x86 disassemblers.

[13] RAMSEY, N. Design principles for machine-description languages.

[14] RAMSEY, N. Machine descriptions to build tools for embedded systems.

[15] SEAL, D. Arm architecture reference manual.