



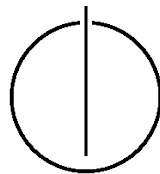
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

# **Validating the Decoding and the Translation into Value Semantics of X86 Machine Code**

Julian Kranz







FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Validating the Decoding and the Translation into Value  
Semantics of X86 Machine Code

Validierung der Dekodierung und der Übersetzung in  
eine Werte-Semantik von Intel x86 Maschinen Code

Author: Julian Kranz  
Supervisor: Prof. Dr. Helmut Seidl  
Advisor: Dr. Axel Simon  
Date: July 18, 2013





I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, the July 18, 2013

Julian Kranz



---

## Abstract

The static analysis of binary code is an established approach for the discovery of programming errors. The first step towards the analysis of a program is its *decoding*. To this end, a decoder scans the binary code and outputs a structured representation of the instructions using data structures defined in a programming language. Afterwards, the decoded data is translated into a standardized intermediate representation (IR). This process is referred to as the *semantic translation* since it expresses the semantics of the binary code using the generic constructs from the IR. The domain specific language *GDSL* is used for the Intel x86 decoder and semantic translator specifications that lay the foundations for this work. The implementation of a decoder and semantic translator for a complex architecture like Intel x86 almost certainly leads to a multitude of programming errors. In order to address this, the thesis presents an approach of autonomously validating the correctness of the respective specifications for the Intel x86 architecture using automatic test case generation.

---



# Contents

|   |             |
|---|-------------|
| <b>Abstract</b>   | <b>vii</b>  |
| <b>Outline of the Thesis</b>  | <b>xiii</b> |
| <br>  |             |
| <b>I. Introduction</b>  | <b>1</b>    |
| 1. Introduction   | 3           |
| <br>  |             |
| <b>II. GDSL</b>   | <b>7</b>    |
| 2. Overview   | 9           |
| 3. A Generic Decoder Specification Language for Interpreting Machine Language | 11          |
| 3.1. General Language Overview  | 13          |
| 3.2. Decoding x86 Prefixes  | 17          |
| 4. Using GDSL for Giving Semantics to Machine Language                        | 19          |
| 4.1. RReil Intermediate Representation  | 19          |
| 4.2. Writing Semantics using GDSL   | 20          |
| 4.2.1. An Example Intel Instruction   | 20          |
| 4.2.2. Generating RReil statements using GDSL monadic functions               | 21          |
| 4.2.3. The Translator   | 22          |
| 4.3. Development State  | 24          |
| <br>  |             |
| <b>III. Validation</b>  | <b>25</b>   |
| 5. Introduction to Validation   | 27          |
| 5.1. Approach   | 27          |
| 5.2. Limitations  | 29          |
| 5.3. Part Overview  | 29          |
| 6. Generating Intel Instructions  | 31          |
| 6.1. The Intel Instruction Format   | 31          |
| 6.1.1. The Classic Instruction Format   | 31          |
| 6.1.2. The AVX Instruction Format   | 32          |
| 6.2. Output Correctness vs. Implementation Simplicity                         | 33          |

|  |           |
|--|-----------|
| 6.3. The Decision Tree of the Generator . . . . .                            | 33        |
| 6.3.1. The X86 Generator Tree . . . . .                                      | 33        |
| 6.4. Invalid instructions . . . . .  | 35        |
| <b>7. The Tester Component</b>   | <b>37</b> |
| 7.1. Operation Modes . . . . .   | 37        |
| 7.1.1. Choice of Registers . . . . .   | 37        |
| 7.1.2. Crash handling . . . . .  | 38        |
| 7.2. Results . . . . .   | 38        |
| <b>8. Inspecting RReil Code</b>  | <b>39</b> |
| 8.1. Determining Register Usage . . . . .                                    | 39        |
| 8.1.1. Required Information about the Accessed Registers . . . . .           | 39        |
| 8.1.2. Handling of Conditional Execution . . . . .                           | 39        |
| 8.1.3. Read Registers . . . . .  | 40        |
| 8.1.4. Written Registers . . . . .   | 40        |
| 8.1.5. Dereferenced Registers . . . . .                                      | 40        |
| 8.1.6. Example . . . . .   | 41        |
| <b>9. The Execution Context</b>  | <b>43</b> |
| 9.1. Representation of Registers . . . . .                                   | 43        |
| 9.1.1. The Domain of a Register . . . . .                                    | 44        |
| 9.2. Representation of Accessed Memory . . . . .                             | 45        |
| <b>10. Generating the Testbed</b>  | <b>47</b> |
| 10.1. Overview . . . . .   | 47        |
| 10.2. Choice of Registers . . . . .  | 48        |
| 10.3. Allocation of Registers . . . . .                                      | 48        |
| 10.3.1. Allocation of Standard Registers . . . . .                           | 49        |
| 10.3.2. Allocating the Stack Pointer . . . . .                               | 49        |
| 10.3.3. Allocating the Flags Register . . . . .                              | 49        |
| 10.4. Initialization of Registers . . . . .                                  | 50        |
| 10.4.1. Initialization of Standard Registers and the Stack Pointer . . . . . | 50        |
| 10.4.2. Initialization of the Flags Register . . . . .                       | 50        |
| 10.4.3. Initialization of the Return Address Register . . . . .              | 51        |
| 10.5. Jumps . . . . .  | 51        |
| 10.6. Memory Read and Write Accesses . . . . .                               | 52        |
| 10.7. Writing back of Register Contents . . . . .                            | 52        |
| 10.7.1. Writing back of Standard Registers and the Stack Pointer . . . . .   | 52        |
| 10.7.2. Writing back of the Flags Register . . . . .                         | 53        |
| 10.8. Deallocation of Registers . . . . .                                    | 53        |
| 10.9. A Complete Example . . . . .   | 53        |
| <b>11. Preparation of the Execution Context</b>                              | <b>57</b> |
| 11.1. Initialization of General Purpose Registers . . . . .                  | 57        |
| 11.2. Initialization of the Flags Register . . . . .                         | 58        |

---

|   |           |
|---|-----------|
| 11.3. Initialization of the Program Counter . . . . .             | 59        |
| 11.4. Biased Random Value Generation . . . . .                    | 59        |
| 11.5. Discussion . . . . .  | 60        |
| 11.6. Examples . . . . .  | 60        |
| <b>12. Simulation of the Execution of RReil Code</b>              | <b>63</b> |
| 12.1. RReil Statements . . . . .                                  | 63        |
| 12.1.1. The Assignment Statement . . . . .                        | 63        |
| 12.1.2. Load and Store Statements . . . . .                       | 63        |
| 12.1.3. Control Flow Statements . . . . .                         | 64        |
| 12.1.4. Branching Statements . . . . .                            | 64        |
| 12.2. Domain Information . . . . .                                | 64        |
| 12.3. RReil Operations . . . . .                                  | 65        |
| 12.3.1. The Linear Operation . . . . .                            | 65        |
| 12.3.2. Addition and Subtraction . . . . .                        | 66        |
| 12.3.3. Multiplication, Division and Modulo . . . . .             | 66        |
| 12.3.4. Shift Operations . . . . .                                | 66        |
| 12.3.5. Bitwise Operations . . . . .                              | 67        |
| 12.3.6. Comparison Operations . . . . .                           | 67        |
| 12.3.7. Extension Operation . . . . .                             | 68        |
| 12.4. Memory Accesses . . . . .                                   | 68        |
| 12.4.1. Read Accesses . . . . .                                   | 69        |
| 12.4.2. Write Accesses . . . . .                                  | 69        |
| 12.4.3. Branching . . . . .                                       | 69        |
| 12.5. Simulation Errors . . . . .                                 | 69        |
| 12.5.1. Unaligned Memory Access . . . . .                         | 70        |
| 12.5.2. Undefined Addresses . . . . .                             | 70        |
| 12.5.3. Undefined Data to be Stored to Memory . . . . .           | 70        |
| 12.5.4. Undefined Conditions . . . . .                            | 71        |
| <b>13. Execution of the Instruction Under Test</b>                | <b>73</b> |
| 13.1. Memory Mapping and Initialization . . . . .                 | 73        |
| 13.1.1. Initilization of Read Memory . . . . .                    | 74        |
| 13.1.2. Initilization of Written Memory . . . . .                 | 74        |
| 13.1.3. Initialization of Jumped at Memory . . . . .              | 74        |
| 13.2. Execution of the Testbed Function and Cleanup . . . . .     | 74        |
| 13.3. Handling of Signals . . . . .                               | 74        |
| 13.3.1. Segmentation Fault (SIGSEGV) . . . . .                    | 75        |
| 13.3.2. Illegal Instruction (SIGILL) . . . . .                    | 75        |
| 13.3.3. Timer (SIGALRM) . . . . .                                 | 75        |
| 13.4. Intercepting Program Crashes . . . . .                      | 76        |
| 13.5. Execution Results . . . . .                                 | 76        |
| <b>14. Evaluation of the Test Results</b>                         | <b>77</b> |
| 14.1. Comparison of the Effects on Registers and Memory . . . . . | 77        |
| 14.1.1. Registers . . . . .                                       | 77        |

---

|  |               |
|--|---------------|
| 14.1.2. Memory . . . . .                                       | 78            |
| 14.2. Error Types . . . . .                                    | 78            |
| 14.2.1. Decoding Errors . . . . .                              | 78            |
| 14.2.2. Translation Errors . . . . .                           | 78            |
| 14.2.3. Simulation Errors . . . . .                            | 79            |
| 14.2.4. Execution Errors . . . . .                             | 79            |
| 14.2.5. Comparison Errors . . . . .                            | 79            |
| 14.3. Statistical Analysis . . . . .                           | 79            |
| 14.3.1. Instruction Abstraction . . . . .                      | 79            |
| 14.3.2. Collected Data . . . . .                               | 80            |
| 14.3.3. CPU Feature Dependence of Errors . . . . .             | 80            |
| <b>15. Examples and Statistics</b>                             | <b>81</b>     |
| 15.1. Example Test Case . . . . .                              | 81            |
| 15.2. Example Programming Error in the Translator . . . . .    | 84            |
| 15.3. Statistical Data on the Current Specifications . . . . . | 88            |
| <br><b>IV. Conclusion</b>                                      | <br><b>91</b> |
| <br><b>16. Conclusion</b>                                      | <br><b>93</b> |
| <br><b>Bibliography</b>  | <br><b>95</b> |

# Outline of the Thesis

## **Part I: Introduction**

### CHAPTER 1: INTRODUCTION

This chapter presents an overview of the thesis and its purpose.

## **Part II: GDSDL**

### CHAPTER 2: OVERVIEW

This chapter gives a summary of the design of the GDSDL programming language and why it is used for the decoder and the semantic translator.

### CHAPTER 3: A GENERIC DECODER SPECIFICATION LANGUAGE FOR INTERPRETING MACHINE LANGUAGE

This chapter describes the decoding of Intel instructions using a decoder written in GDSDL.

### CHAPTER 4: USING GDSDL FOR GIVING SEMANTICS TO MACHINE LANGUAGE

This chapter describes the semantic translation of decoded Intel instructions using a translator written in GDSDL.

## **Part III: Validation**

### CHAPTER 5: INTRODUCTION TO VALIDATION

This chapter introduces the reader to the general approach used for the validation.

### CHAPTER 6: GENERATING INTEL INSTRUCTIONS

This chapter details on the generation of Intel instructions used as test objects.

### CHAPTER 7: THE TESTER COMPONENT

This chapter describes the main component of validation software which controls the test process.

### CHAPTER 8: INSPECTING RREIL CODE

This chapter details on the gathering of data required for the execution of instruction under test from the translated semantics.

### CHAPTER 9: THE EXECUTION CONTEXT

This chapter describes the data structure used to represent the system state of both the

simulator and the actual processor.

### CHAPTER 10: GENERATING THE TESTBED

This chapter presents the generation of the test environment the instruction under test is embedded in for its execution.

### CHAPTER 11: PREPARATION OF THE EXECUTION CONTEXT

This chapter details on the initialization of the virtual system state before the simulation of semantics and the execution of the instruction under test.

### CHAPTER 12: SIMULATION OF THE EXECUTION OF RREIL CODE

This chapter describes the simulation of the instruction semantics obtained from the GDSDL toolkit.

### CHAPTER 13: EXECUTION OF THE INSTRUCTION UNDER TEST

This chapter details on the execution of the instruction under test embedded in the test environment on the actual processor.

### CHAPTER 14: EVALUATION OF THE TEST RESULTS

This chapter describes the evaluation of the test results. It also illustrates error conditions that cause the test to fail.

### CHAPTER 15: EXAMPLES AND STATISTICS

This chapter demonstrates how the validation software can be used to trace errors or obtain statistical data using examples.

## Part IV: Conclusion

### CHAPTER 16: CONCLUSION

This chapter summarizes the findings of the thesis.

## **Part I.**

# **Introduction**





# 1. Introduction

The static analysis of binary code is an established approach for the discovery of programming errors. The first step towards the analysis of a program is its *decoding*. To this end, a decoder scans the binary code and outputs a structured representation of the instructions using data structures defined in a programming language. Afterwards, the decoded data is translated into a standardized intermediate representation (IR). This process is referred to as the *semantic translation* since it expresses the semantics of the binary code using the generic constructs from the IR. Particularly, the implementation of the decoding of binary code is cumbersome using general purpose programming languages. For this reason, the domain specific language *GDSL* [10] specifically geared towards the implementation of instruction decoders is used for the x86 decoder implementation that provides the basis for this work. Since the GDSL language is a fully featured functional language which incorporates major features of Standard ML, it also qualifies for the implementation of the corresponding semantic translator.

The GDSL toolkit contains a decoder specification as well as a semantic translator specification for the Intel x86 architecture. The architecture has got a long history of extensions - by now, it incorporates a lot of different concepts that lead to a complex rule set for the decoder. Examples for pitfalls include a growing number of prefixes and different default values for instruction and address sizes depending on the execution mode of the processor. The Intel architecture is a CISC architecture; therefore, it consists of a large number of instructions, some of which are powerful and complex. Since the intermediate representation which the semantic translation translates the decoded instructions into only offers simple standard operations, the implementation of the semantic translator is considerable. By now, the decoder is able to handle all of the almost 900 different instructions and the semantic translator offers translations for at about half of them.

The implementation of the great number of routines needed to handle all these instructions almost certainly leads to a multitude of programming errors. An additional cause for programming errors is the quality of the Intel manual itself which occasionally is vague. The description of the instruction semantics uses an ad-hoc pseudo language that follows no precise syntax definition but instead uses different syntactic elements for logically equivalent constructs. While it is very hard to prove a program to be correct, for all practical purposes *testing* its functionality using example inputs for which correct output values (the *reference* values) are known has proven to be a valuable way to increase the code quality. For the instruction decoder and semantic translator, the test input data consist of a binary instruction which is decoded and translated. This results in a semantic translation expressed in the chosen intermediate representation. In order to evaluate the test result, it is necessary to verify the correctness of the output of the translation.

In general, the test cases used need to feature a thorough coverage of the functionality of the tested software program in order to allow for a reliable estimate of the code correctness and the dependable detection of programming mistakes. Because of that, the test cases

should be easily generable since a lot of them are needed: For the x86 specifications from the GDSL toolkit, it is not only necessary to test the decoding and the semantic translation of each instruction, but also include different operand combinations. A manual implementation of a vast amount of unit tests is very cumbersome. Besides that, the construction of reference values is also difficult when it comes to the semantic translator. Basically, a manually created reference semantics would again rely on the instruction semantics given by the Intel manual. Thus, errors made in the implementation of the semantic translator would very probably be repeated during the test case construction.

In order to address the issues stated above, this work presents an approach of validating the correctness of the specifications for the Intel x86 architecture written in GDSL using automatic test case generation. The validation software utilizes the actual processor it is running on for the construction of reference values. This allows for a fast and autonomous generation of numerous test cases. A single test case consists of the following main steps performed in the order given:

- **Instruction generation**

In the beginning, a random Intel instruction is generated (which is called the *instruction under test*). This does not only include a random opcode, but also arbitrary operand combinations and immediate values.

- **Decoding and semantic translation**

Next, the instruction under test is handed to the GDSL toolkit for its decoding and semantic translation. This yields the semantics to be validated for correctness expressed in the intermediate representation.

- **Simulation of the semantics**

The semantics itself is not compared to reference semantics since it is not possible to obtain such reference semantics automatically. Instead, the effects of the execution of instruction under test and simulation of the translated semantics on the system registers and memory are compared. For this, the execution of the semantics obtained from the GDSL toolkit is simulated.

- **Execution of the instruction under test**

In order to obtain a reference value for the effects of the instruction under test on the system registers and memory, the instruction is executed.

- **Comparison of the system registers and memory**

Finally, the changes on the system registers and memory induced by both the simulation of the semantics obtained from the GDSL toolkit and the execution of the instruction under test on the actual processor are compared. If they match (and no other errors occurred so far), the test succeeds. Otherwise, the test fails. In that case, a detailed description of the deviation of the two resulting states regarding registers and memory is displayed.

Using the method presented above, errors in the specifications can be easily found by repeatedly running test cases until the first error occurs. As stated above, the output of the validation software includes data on the kind of error which helps to locate the problem within one of the specifications. Furthermore, running a large number of test cases allows

---

for an analysis of the correctness of the current implementation by examining the output data statistically.

The thesis is divided into four parts. After this introduction, the next part describes the implementation of an instruction decoder and semantic translator specification using the GDSDL programming language. It is followed by Part **III** which is the main part of the thesis. This part details the validation of the decoder and semantic translator specifications. Finally, Part **IV** summarizes the findings of the thesis.



## **Part II.**

# **Using GDSL for the Decoding and Semantic Translation of Machine Language**



## 2. Overview

In order to perform static analysis of binary programs, machine code needs to be interpreted. The first step consist of the *decoding* of the application code. To this end, a decoder transforms the byte sequences of the binary program into structured objects of a given programming language. The decoder outputs an object for each instruction which allows a programmer to easily access the different properties of the instruction, like, for example, the type of operation and its operands. Even though these objects are a more generic representation of the instructions than the binary code, they are still highly architecture dependent. An analysis framework implemented using these objects would therefore be bound to that architecture. The support of different architectures would require the framework to be adapted to each of them which would result in a massive programming overhead and also lead to code duplication. In order to address these problems, an additional level of abstraction is introduced: The analysis is implemented using a small set of basic operations that forms an intermediate representation (IR). An example for such an intermediate representation is the *RReil* language [11]. In order to allow an analysis framework to be implemented architecture independent using RReil, the more complex and machine dependent operations given by the a concrete processor architecture need to be transformed into a program consisting of RReil statements. Since this models the semantics of the machine instructions, the process is referred to as *semantic translation*.

Implementing a decoder for binary code using common general purpose programming languages is inconvenient and error prone. This is primarily caused by the fact that the decoding involves a lot of bit operations used to to extract the different properties of an instruction from the byte stream. This led to the design of a domain specific language called *GDSL* [10] which is particularly geared towards implementing instruction decoders. Decoding an instruction involves numerous case distinctions; for example, the bit sequence representing the opcode of an instruction needs to be mapped to the type of the instruction. Functional languages are in general best suited for pattern matching and, thus, for case distinctions. Therefore, the GDSL language is designed to be a functional language.

As metioned above, the binary decoder outputs the decoded instructions in a representation that is specific to the programming language used. Even though it is possible to interface between different programming languages, this additional step requires programming effort and implies a loss of type safety at the point of handover. Therefore, it is preferable to implement the processing of the objects in the same language they were originally built in. For the matter in hand, this means to implement the semantic translation using the GDSL language. Even though GDSL is designed to ease the implementation of instruction decoders, it is a fully featured programming language which borrows major concepts from Standard ML. For this reason, it also allows the specification of a semantic translator.

The following chapters of this part describe the implementation of an instruction decoder and semantic translator for the Intel x86 architecture using the GDSL programming

## 2. Overview

---

language. They are derived from the two papers *GDSL: A Generic Decoder Specification Language for Interpreting Machine Language* [10] and *GDSL: A Universal Toolkit for Giving Semantics to Machine Language* [6].



### 3. A Generic Decoder Specification Language for Interpreting Machine Language

The reconstruction of assembler instructions from an input (byte) sequence that comprise the program is the first step towards static program analysis. The second step is to map each statement to a meaning which may be a value-, timing- or energy semantics, etc., depending on the goal of the analysis. Both aspects are commonly addressed by writing an architecture-specific decoder and a translator to some internal representation expressed in the implementation language of the analysis. The goal of this chapter is to build an infrastructure to specify decoders and translations to semantics using a domain specific language (DSL) that can be compiled into the programming language of existing analysis tools. To this end, GDSDL is presented and its design is motivated by the task of specifying decoders for Intel x86 machine code.

The incentive for creating a DSL to specify decoder and semantics of assembler instructions was a discussion at a Dagstuhl seminar on the analysis of executable code. Here, it was realized that many research groups implemented prototypes analyses using an architecture specific decoder and a hand-written semantic interpretation. Besides duplication of work, these approaches are usually incomplete, are bound to one architecture and are hard to maintain since their representation of instructions is geared towards a specific project. In the presence of steadily increasing instruction sets and the need to adapt an analysis to new targets such as virtual machines contained in malware, maintainability and simplicity of decoder specifications is of increasing importance.

To this end, it is desirable to group instructions logically or, when converting a manufacturer's manual, in alphabetical order; this is introduced as mnemonic-centric specification. For the sake of efficiency, however, a decoder must make a decision based on the next value from the input sequence (opcode-centric dispatch) which precludes testing opcode patterns one after the other. While a classic scanner generator like `lex` can convert a mnemonic-centric specification to an opcode-centric decoder, it allows and encourages overlapping patterns. Consider the following `lex` scanner specification:

|   |                                   |                                   |
|---|-----------------------------------|-----------------------------------|
| 1 | <b>while   do   switch   case</b> | { printf("keyword %s", yytext); } |
| 2 | <b>[a-zA-Z][a-zA-Z0-9]*</b>       | { printf("ident %s", yytext); }   |

Here the patterns for the keywords and the identifier are overlapping: the input `while` matches both rules. In this case, `lex` uses the rule that appears first in the specification file. Thus, a keyword is returned. Overlapping patterns are desirable in a scanner specification since they improve readability and conciseness. In an instruction decoder, however, overlapping patterns are undesirable since the sequence in which the rules are written starts to matter which, in turn, precludes a mnemonic-centric specification. Hence, a DSL for maintainable decoder specifications must provide a concise way of writing non-overlapping patterns to exactly match an instruction.

| Opcode | Instruction  | Description            |
|--------|--------------|------------------------|
| 00 /r  | ADD r/m8, r8 | Add r8 to r/m8.        |
| 28 /r  | SUB r/m8, r8 | Subtract r8 from r/m8. |

Figure 3.1.: Two typical instructions in the Intel x86 manual.

Another challenge is the processing of non-constant bits of an instruction that are used to specify parameters. Since parameter bits often follow re-occurring patterns, an abstraction mechanism is required to keep the specification concise. For example, the `ModR/M`-byte in Intel x86 instructions follows many opcodes and determines which register to use. Figure 3.1 shows an excerpt of the Intel manual where the first column shows the two bytes that together form an instruction. The second byte `/r` is the `ModR/M`-byte that determines which 8-bit registers `r8` and which pointer `r/m8` stand for. Within the decoder specification language, the functions `r/m8`<sup>1</sup> and `r8` are defined to generate the arguments of an instruction. The content of the `ModR/M`-byte are read by a sub-decoder named `/r` that stores the read byte in an internal decoder state. This sub-decoder can be re-used in the decoder for `add` and `sub`:

```

1 val main [0x00 /r] = binop ADD r/m8 r8
2 val main [0x28 /r] = binop SUB r/m8 r8

```

Here, the decoder `main` is declared as reading `0x00` (resp. `0x28`) from the input before running the sub-decoder `/r`. The `binop` function is a simple wrapper that executes functions `r/m8` and `r8` (which access the values stored by `/r`) and applies the results to the passed-in constructor (here `ADD` and `SUB`). By using sub-decoders such as `/r` that communicate via the internal state, the `main` decoder comes very close to the specification in Fig. 3.1 of the Intel manual.

Since the DSL is an ML-like functional language, it is powerful enough to describe all parts of a decoder, even `r/m8` and `r8` that are often hand-coded primitives in other decoder frameworks. This comprehensive approach enables users to add instructions that have not been anticipated in the original design of `/r`. In summary, GDSDL improves over existing approaches as follows:

- Its abstraction mechanisms enable the definition of instruction decoders that are very close to the syntax used in manufacturer’s manuals, thereby ensuring maintainability even by the end users of the decoder framework.
- The specification is type checked during compilation and overlapping patterns are detected. This ensures high reliability of the resulting decoder, especially in the presence of mistakes in the manuals of the manufacturer.
- The DSL is flexible enough to accommodate a variety of architectures. Due to its general nature, it is possible to add translations from native instructions to some abstract semantics which will enable binary analysis tools to analyse code for any architecture that is described with the framework.

---

<sup>1</sup>The slash (/) is allowed as part of an identifier to accommodate the Intel nomenclature.

- We provide a prototype compiler that generates C code which is competitive with other decoders. The specifications can be translated to other languages or used for other purposes (e.g. test generation) by writing a new backend.

After the next section presents the design of GDSDL, Sect. 3.2 illustrates its expressiveness by detailing the decoding of Intel prefixes.

### 3.1. General Language Overview

This section discusses the design of GDSDL by illustrating the use of the various syntactic constructs. The general idea is that the decoder specification is an executable functional program that consumes the input sequence and produces a heap containing the *abstract syntax tree* (AST) that represents the recognized instruction. After the AST in the heap has been processed, the heap can be reused for decoding the next instruction, thereby avoiding the need for a garbage collector or for allocating memory with each instruction.

The grammar of GDSDL is shown in Fig. 3.2. A file consists of a sequence of definitions given by *Decl*. The **granularity** statement can be given once and defines the size of the tokens that the decoder consumes. A token is measured in bits and is the smallest granularity that a processor reads from memory. For Intel x86, the token size is 8 (and each instruction can have between one and fifteen tokens). For standard ARM instructions, the token size is 32 (and no instruction is longer than one token). Other processors are in between these extremes, for instance, MicroChip’s PIC architecture has a token size of 14. The optional **lsbfirst** keyword states that bit sequences in decoders start with the least significant bit, a notation used for e.g. PowerPC.

The **export** keyword states which of the decoders are publicly visible to the client code. Line 3.3 shows the production for algebraic data types that introduce (or extend) the type **t-id** with constructors **con**. As in ML, each constructor takes zero or one argument, allowing the definition of enumerations such as the following:

```
1 type register = AX | BX | CX | DX
```

AST nodes such as shown below also are possible:

```
1 type op =
2   Reg of register
3   | Mem of {size : int, reg : op}
4   | Imm8 of [8]
```

Here, the argument to the `Mem` constructor is a record while `Imm8` takes a bit vector of 8 bits, written `[8]`. Bit vectors and **int** are the only basic data types with singleton bit-vectors acting as Booleans. Abbreviations for complex types can be introduced using the syntactic construct in line 3.4.

Productions 3.5, 3.6, and 3.7 introduce functions, decoders and decoders with guards, respectively. Functions and decoders differ in that functions take arguments and have exactly one definition whereas decoders read from the implicit input stream and definitions with the same name augment each other. Consider the decoder snipped in Fig. 3.3. Here,

|              |  |        |
|--------------|--|--------|
| $Decl ::=$   | <b>granularity</b> = <b>num</b> [ <b>lsbfirst</b> ]  | (3.1)  |
|              | <b>export</b> <b>id</b> *  | (3.2)  |
|              | <b>type</b> <b>id</b> = <b>con</b> [ <b>of</b> <i>Type</i> ] (  <b>con</b> [ <b>of</b> <i>Type</i> ])*                           | (3.3)  |
|              | <b>type</b> <b>id</b> = <i>Type</i>  | (3.4)  |
|              | <b>val</b> <b>id</b> <b>id</b> * = <i>Expr</i>   | (3.5)  |
|              | <b>val</b> <b>id</b> [ <i>TokPat</i> *] = <i>Expr</i>  | (3.6)  |
|              | <b>val</b> <b>id</b> [ <i>TokPat</i> *] (  <i>Expr</i> = <i>Expr</i> ) <sup>+</sup>  | (3.7)  |
| $Type ::=$   | <b>int</b>     <b>num</b>     <b>id</b>  | (3.8)  |
|              | { <b>field</b> : <i>Type</i> (, <b>field</b> : <i>Type</i> ) <sup>*</sup> }  | (3.9)  |
| $TokPat ::=$ | <b>hex-num</b>   <b>id</b>   ' <i>BitPat</i> * '   | (3.10) |
| $BitPat ::=$ | <i>BitStr</i> (  <i>BitStr</i> ) <sup>*</sup>  | (3.11) |
|              | <b>id</b> @ <i>BitStr</i> (  <i>BitStr</i> ) <sup>*</sup>  | (3.12) |
|              | <b>id</b> : <b>num</b>   | (3.13) |
| $BitStr ::=$ | ( <b>0</b>   <b>1</b>   <b>.</b> ) <sup>+</sup>  | (3.14) |
| $Expr ::=$   | <b>case</b> <i>Expr</i> <b>of</b> <i>Pat</i> : <i>Expr</i> (  <i>Pat</i> : <i>Expr</i> ) <sup>*</sup>                            | (3.15) |
|              | <b>if</b> <i>Expr</i> <b>then</b> <i>Expr</i> <b>else</b> <i>Expr</i>  | (3.16) |
|              | <b>let</b> ( <b>val</b> <b>id</b> = <i>Expr</i> ) <sup>+</sup> <b>in</b> <b>Expr</b>   | (3.17) |
|              | <i>Expr</i> <i>Expr</i>   <b>num</b>   ' <i>BitStr</i> '   <b>id</b>   <b>con</b>  | (3.18) |
|              | { <b>field</b> = <i>Expr</i> (, <b>field</b> = <i>Expr</i> ) <sup>*</sup> }  | (3.19) |
|              | <b>\$field</b>   @ { <b>field</b> = <i>Expr</i> (, <b>field</b> = <i>Expr</i> ) <sup>*</sup> }                                   | (3.20) |
|              | <b>do</b> ( <i>Expr</i> <sub><i>i</i></sub>   <b>id</b> <- <i>Expr</i> <sub><i>i</i></sub> ) <sup>*</sup> <i>Expr</i> <b>end</b> | (3.21) |
|              | <b>update</b> <i>Expr</i>   <b>query</b> <i>Expr</i>   <b>return</b> <i>Expr</i>   | (3.22) |

Figure 3.2.: Syntax of the GDSDL language.

```

1  granularity 8
2  export main
3  type instr = ADD of {op1:op, op2:op}
4
5  val binop cons giveOp1 giveOp2 = do
6    operand1 <- giveOp1;
7    operand2 <- giveOp2;
8    return (cons {op1=operand1, op2=operand2})
9  end
10
11 val /r [ 'mod:2 _reg:3 _rm:3 ' ] =
12   update @{mod=mod, reg/opcode=reg, rm=rm}
13 val /0 [ 'mod:2 _000 _rm:3 ' ] =
14   update @{mod=mod, reg/opcode='000', rm=rm}
15 val r/m8 = do # similar for r8, r/m16, r16, ...
16   r <- query $rm;
17   return (case r of '000': Reg AL | '001': Reg BL )
18 end
19
20 val main [0x80 /0] = binop ADD r/m8 imm8
21 val main [0x00 /r] = binop ADD r/m8 r8
22 val main [0x01 /r] | opndsz? = binop ADD r/m16 r16
23                       | rexb? = binop ADD r/m64 r64
24                       | otherwise = binop ADD r/m32 r32

```

Figure 3.3.: Specification for decoding the Intel add instruction.

### 3. A Generic Decoder Specification Language for Interpreting Machine Language

---

`binop` and `r/m8` in lines 5 and 15 are functions taking three and no arguments, respectively. In contrast, lines 11, 13 and 20 define decoders whose right-hand-side is evaluated if the token sequence in the square brackets matches the current input. Tokens can be specified in three ways (Production 3.10): either as a hexadecimal number (c.f. the first token of `main`), as a call to another decoder (c.f. the second token of `main`) or as a bit pattern (as used in the `/r` and `/0` decoders). Bit patterns, in turn, are enclosed in ticks and are given by Productions 3.11, 3.12, and 3.13:

- strings of 0,1,. (c.f. 000 in `/0`); the dot acts as a wildcard; a set of bit strings can be specified by separating them using a vertical bar, e.g. 00|01|10
- as above, with a leading variable separated by @; the variable is bound to the actual bits in the input; for instance, `/0` could have been written

```
1  val /0 [ 'mod:2 _reg@000 _rm:3 ' ] =  
2    update @{mod=mod, reg/opcode=reg, rm=rm}
```

- a variable with a width in bits; the notation `v:3` is syntactic sugar for `v@...;` examples are `mod`, `reg` and `rm` in the decoders `/r` and `/0`

The semantics of “calling” another decoder within a token sequence is that the pattern of the called decoder is substituted where it appears and that its body is prepended to the right-hand-side of the decoder. For instance, `main [0x80 /0]` is translated internally as follows:

```
1  val main [0x80 'mod:2 _000 _rm:3 ' ] = do  
2    update @{mod=mod, reg/opcode='000', rm=rm};  
3    binop ADD r/m8 imm8  
4  end
```

After inlining sub-decoders, the patterns of all `main` rules are translated using a `consume` primitive that reads one token from the input stream:

```
1  val main = do  
2    opcode <- consume  
3    case opcode of  
4      0x80 : do  
5        \r <- consume  
6        case (\r & 00111000 >> 3) of  
7          000 : do  
8            update @{mod=, reg/opcode='000', rm=rm};  
9            binop ADD r/m8 imm8  
10         ...  
11      0x00 : ...
```

During this translation overlapping patterns are detected. For token sizes larger than 8 bits, nested `case`-statements are generated.

The bodies of functions and decoders are given by the *Expr* production. Here, Productions 3.15, ..., 3.18 give the standard constructs found in a functional language with *Expr Expr* in line 3.18 denoting function application. The language allows the creation of

compound values using records which are collections of field names bound to a value. Production 3.19 allows the construction of new records (used in line 8 of Fig. 3.3). The value of a field `foo` is extracted using `$foo` which itself is a function. Thus, `$foo {foo=7}` evaluates to 7. Analogously, `@{foo=x}` is a function taking a record and setting the field `foo` to `x`. For instance, `@{bar='110'} {foo=7}` evaluates to `{bar='110', foo=7}`.

In order to allow for an internal state, each decoder is a monad, a concept borrowed from the pure functional language Haskell [8]. A monad is an abstract type containing a function from an input state to an output state and a result. The motivation for monads is to chain together computations that operate on a state without requiring side-effects in the language. Production 3.21 details the **do**-statement which threads together monadic actions whose result can be bound to an identifier. The result of the **do**-statement is that of the last action. Production 3.22 presents the three monadic actions of the language: **update** `f` applies `f` to the internal state (and is usually a record update); **query** `f` returns the result of applying `f` to the internal state (and is usually a record field selector); and **return** `x` that returns `x` as a result.

Besides **query**, the internal state can also be accessed using guards: the first guard of `opndsz?`, `rexw?`, and `otherwise` in line 22 that evaluates to '1' determines which right-hand-side is evaluated. Guards are functions taking the internal state as argument. Thus, `opndsz` and `rexw` are record fields in the internal state and `otherwise` is a function always returning '1'.

## 3.2. Decoding x86 Prefixes

One challenge in decoding x86 instructions is the correct handling of prefixes: they either serve to modify the following instruction or they are part of the following opcode (a so-called *mandatory* prefix). In the latter case, other prefixes are allowed between the mandatory prefix and the actual opcode. For example, both instruction sequences `67 f3 45 0f 7e d1` and `f3 67 45 0f 7e d1` encode `movq xmm10, xmm9` where `67` is an `ADDRSZ` prefix and `f3` is a `REPNE` prefix, but used here as mandatory prefix to the opcode `0f 7e`. Moreover, `45` is another “standard” REX prefix and `d1` the `ModR/M` byte. Confusingly, the REX prefix must immediately precede the opcode, otherwise it is ignored.

Certain instructions such as `mulss`, `mulsd`, and `mulpd` share the same opcode, here `0f 59`, but have different mandatory prefixes, namely `f2`, `f3`, and `66`, respectively. As a consequence, the *order* in which prefixes occur becomes important. Moreover, while the last occurrence of `f2` and `f3` determines the mandatory prefix, an occurrence of `66` is only recognized as mandatory prefix if `f2` and `f3` cannot start an instruction. A correct decoder recognizes:

|                                |                               |                               |
|--------------------------------|-------------------------------|-------------------------------|
| <code>66 f3 f2 0f 59 ff</code> | <code>mulsd xmm7, xmm7</code> | Mandatory prefix: <b>0xf2</b> |
| <code>66 f2 f3 0f 59 ff</code> | <code>mulss xmm7, xmm7</code> | Mandatory prefix: <b>0xf3</b> |
| <code>66 0f 59 ff</code>       | <code>mulpd xmm7, xmm7</code> | Mandatory prefix: <b>0x66</b> |
| <code>f2 66 0f 59 ff</code>    | <code>mulsd xmm7, xmm7</code> | Mandatory prefix: <b>0xf2</b> |

Mandatory prefixes can easily be handled in GDSL by using different decoders, depending on the last relevant prefix. Prefixes are decoded as follows:

```
1 | val prefixes [0x66] = p/66
```

### 3. A Generic Decoder Specification Language for Interpreting Machine Language

---

```
2 val prefixes [0xf2] = p/f2
3 val prefixes [0xf3] = p/f3
4 val prefixes [] = main
5 val p/66 [0x66] = p/66
6 val p/66 [0xf2] = p/66/f2
7 val p/66 [0xf3] = p/66/f3
8 val p/66 [] = after /66 main
9 val p/f3 [0x66] = p/66/f3      # f3 dominates 66
10 val p/f3 [0xf2] = p/f3/f2
11 val p/f3 [0xf3] = p/f3
12 val p/f3 [] = after /f3 main
13 val p/f3/f2 [0x66] = p/66/f3/f2 # f3/f2 dominates 66
14 val p/f3/f2 [0xf2] = p/f3/f2
15 val p/f3/f2 [0xf3] = p/f2/f3
16 val p/f3/f2 [] = after /f2 (after /f3 main)
17 ... # analogous for p/f2, p/66/f2, p/66/f3, p/f2/f3,
18      # p/66/f3/f2, p/66/f2/f3
19 val /66 [] = continue
20 val /f2 [] = continue
21 val /f3 [] = continue
22 val /66 [0x0f 0x59 /r] = binop MULPD xmm xmm/m128
23 val /f2 [0x0f 0x59 /r] = binop MULSD xmm xmm/m64
24 val /f3 [0x0f 0x59 /r] = binop MULSS xmm xmm/m32
25 val main [...] = ...
```

The entry point that is exported to the user is `prefixes`. When reading the sequence `f3 f2 0f 59 ff`, it dispatches to `p/f3` which itself reads `f2` and enters the `p/f3/f2`. Since the next byte `0f` has no match in `p/f3/f2`, the expression `after /f2 (after /f3 main)` is executed. The `after` function calls the decoder `/f2` and, if it fails, continues with `(after /f3 main)`. The latter expression runs `f3` and, if this decoder fails, runs `main`. On the example byte sequence, the `/f2` decoder succeeds in consuming the remaining bytes `0f 59 ff` and returns the `mulsd` instruction. By construction of the prefix decoders, at most four lookups can lead to failure: one prefix decoder, `/66`, `/f2`, `/f3`. Thus, at most one byte of the sequence is examined more than once.

Observe that `after` and `continue` can be defined directly within GDSL:

```
1 val after fst snd = do update @{cont=snd}; fst end
2 val continue = do decoder <- query $cont; decoder end
```

Here, `after` stores its argument `snd` in the decoder state and executes the decoder `fst`. The `continue` function retrieves the stored decoder and dispatches to it. This completes the design of the prefix decoders.



## 4. Using GDSL for Giving Semantics to Machine Language

The basis for analyzing executables is the decoding of byte sequences into assembler instructions and giving a semantics to them. The challenge here is one of scalability: a single line in a high-level language is translated into several assembler (or “native”) instructions. Each native instruction, in turn, is translated into several semantic primitives. These semantic primitives are usually given as an *intermediate representation* (IR) and are later evaluated over an abstract domain [3] tracking intervals, value sets, taints, etc. In order to make the evaluation of the semantic primitives more efficient, a *transformer-specification language* (TSL) was recently proposed that compiles the specification of each native instruction directly into operations (transformers) over the abstract domain [7], thus skipping the generation of an IR. The approach presented follows the more traditional approach of generating an IR that an analysis later interprets over the abstract domains. This allows to perform optimizations on the IR program that represents a complete basic block rather than on a single native instruction.

The next section shortly describes the intermediate language used. Thereafter, Sect. 4.2 illustrates how GDSL can be used to specify the semantics of instructions by discussing the translator for an example Intel x86 instruction. Finally, Sect. 4.3 details on the current development state of the GDSL toolkit.

### 4.1. RReil Intermediate Representation

Many intermediate representations for giving semantics to assembler instructions exist, each having its own design goals such as minimality [1, 4], mechanical verifiability [5], reversibility [9], or expressivity [1, 11]. The RReil IR [11], presented in Fig. 4.1, was designed to allow for a precise numeric interpretation. For instance, comparisons are implemented with special tests rather than expressed at the level of bits which is common in other IRs [4, 5, 7].

|   |   |
|---|---|
| <pre> <b>stmts</b> ::= <b>stmt</b>   <b>stmt</b> ; <b>stmts</b> <b>stmt</b>  ::= <b>var</b> = : <u>int</u> <b>expr</b>             <b>var</b> = : <u>int</u> [ <b>addr</b> ]             [ <b>addr</b> ] = : <u>int</u> <b>expr</b>             <u>if</u> ( <b>sexpr</b> ) { <b>stmts</b> } <u>else</u> { <b>stmts</b> }             <u>while</u> ( <b>sexpr</b> ) { <b>stmts</b> }             <u>cbranch</u> <b>sexpr</b> ? <b>addr</b> : <b>addr</b>             <u>branch</u> ( <u>jump</u>   <u>call</u>   <u>ret</u> ) <b>addr</b>             ( <b>var</b> : <u>int</u> ) * = " <u>id</u> " ( <b>linear</b> : <u>int</u> ) * <b>cmp</b>   ::= <u>≤<sub>s</sub></u>   <u>≤<sub>u</sub></u>   <u>≤<sub>s</sub></u>   <u>≤<sub>u</sub></u>   <u>≡</u>   <u>≠</u> </pre> | <pre> <b>var</b>    ::= <u>id</u>   <u>id</u> . <u>int</u> <b>addr</b>   ::= <b>linear</b> : <u>int</u> <b>linear</b> ::= <u>int</u> . <b>var</b> + <b>linear</b>             <u>int</u>   <b>var</b> <b>sexpr</b> ::= <b>linear</b>   <u>arbitrary</u>             <b>linear</b> <b>cmp</b> : <u>int</u> <b>linear</b> <b>expr</b>  ::= <b>sexpr</b>             <b>linear</b> <b>bin</b> <b>linear</b>             <u>sign-extend</u> <b>linear</b> : <u>int</u>             <u>zero-extend</u> <b>linear</b> : <u>int</u> <b>bin</b>   ::= <u>and</u>   <u>or</u>   <u>xor</u>   <u>shr</u>   ... </pre> |
|---|---|

Figure 4.1.: The syntax of the RReil (Relational Reverse Engineering Language) IR. The construct ": int" denotes the size in bits whereas ". int" in the **var** rule denotes a bit offset. The statements are: assignment, read from address, write to address, conditional, loop (both only used to express the semantics within a native instruction), conditional branch, unconditional branch with a hint of where it originated, and a primitive "id".

## 4.2. Writing Semantics using GDSL

As a purely functional language with algebraic data types and a state monad, GDSL lends itself for writing translators in a concise way as illustrated next.

### 4.2.1. An Example Intel Instruction

The following GDSL example shows the translation of the Intel instruction `cmov`. The instruction copies the contents of its source operand to its destination operand if a given condition is met. The instruction contains a condition (which is part of the opcode) and two operands, one of which can be a memory location. The translation of the instruction instance `cmovz ebx, eax` (using the Intel x86 architecture with the 64 bit extension) into RReil is shown in Fig. 4.2b). In order to illustrate the translation, the output of the GDSL decoder is first detailed which is a value of the algebraic data type `insn` that is defined as follows:

```

1 type insn =                                     # an x86 instruction
2   CMOVZ of {opnd1: opnd, opnd2: opnd}
3   | ...      # other instruction definitions omitted

```

Thus, the `CMOVZ` constructor carries a record with two fields as payload. Both fields are of type `opnd` which, for instance, carry a register or a memory location:

```

1 type opnd =                                     # an x86 operand
2   REG of register
3   | MEM of memory
4   | ...      # immediates, scaled operands

```

5

# and operands with offsets omitted

Note that all variants (here `REG` and `MEM`) implicitly contain information about the access size. In the example above, the instruction `cmovz ebx, eax` is represented by `CMOVZ {opnd1 = REG EBX, opnd2 = REG EAX}` where `EAX` is of size 32-bits. The following section details helper functions that operate on `opnd` values.

#### 4.2.2. Generating RReil statements using GDSL monadic functions

Each semantic translator generates a sequence of RReil statements. The sequence is stored inside the state of a monad. Each RReil statement is represented by a respective GDSL monadic function which builds the RReil statement from its arguments and appends it to the current list of statements. The following RReil generator functions are used in the example:

- `mov size destination source`  
The `mov` function generates the RReil `mov` statement that copies the source RReil register to the destination RReil register.
- `_if condition _then statements`  
This function generates the RReil `if.then.else` statement (with an empty `else` branch). The special mix-fix notation `_if c _then e` is a call to a mix-fix function whose name is a sequence of identifiers that each commence with an underscore. For instance, the `_if_then` function above is defined as follows:

```

1  val _if c _then a = do
2    ...      # add if c then a else {} to statement list
3  end

```

Furthermore, the following functions operate on x86 operands. In the context of the generation of RReil code, this expression is a short form for the RReil register, memory location, or immediate value associated with the x86 operand:

- `sizeof x86-operand`  
Returns the size of an x86 operand in bits; here, `sizeof (REG EBX) = 32`.
- `lval size x86-operand`  
The `lval` function turns an x86 operand into an RReil left hand side expression, that is, either `var` or `[addr]`. Here, `lval 32 (REG EBX)` yields the RReil register `B` that contains the 32 bits of the Intel `EBX` register.
- `rval size x86-operand`  
The `rval` function turns an x86 operand into an RReil `expr`. In the example, `rval 32 (REG EAX)` yields the RReil register `A`.
- `write size destination source`  
The `write` function emits all statements necessary to write to an x86 operand. The operand is specified using the *destination* parameter; it is the return value of an associated call to `lval`. In Fig. 4.2b) lines 6 through 7 originate from the call to `write`.

Finally, the `mktemp` function is used to allocate a temporary variable.

##### 4.2.3. The Translator

The translation function for `cmovz ebx, eax` is shown in Fig. 4.2a). The `do ... end` notation surrounding the function body is used to execute each of the enclosed monadic functions in turn. The decoded Intel instruction is passed-in using the `insn` parameter; the condition is determined by the caller depending on the actual mnemonic. The condition is an one-bit RReil expression. In the `cmovz ebx, eax` example, it is `FLAGS/6` which corresponds to the zero-flag.

The translation itself starts with a code block that is very common in instruction semantics: The operation's size is determined by looking at the size of one operand (line 2) and the respective operands are prepared for reading (using the `rval` monadic function) and writing (using the `lval` monadic function). Next, a new temporary RReil register is allocated and initialized to the current value of the destination operand (lines 7 and 8). This completes all preparations; the actual semantics of the instruction is implemented by the code lines 10 through 11. The condition is tested and, if it evaluates to *true*, the source operand is copied to the destination operand. It is important to note that the condition is not evaluated at translation time, but at runtime by RReil. Finally, the (possibly) updated value of the temporary RReil register is written to the corresponding Intel register by code line 13.

One might think that the instruction pointlessly reads the source operand and writes the destination operand in case the condition evaluates to *false*. It is, however, necessary since the writeback can also cause further side effects that still need to occur, even if no data is copied. This is visible in Fig. 4.2 (b). Since the instruction uses a 32 bit register in 64 bit mode, the upper 32 bits of the register are zeroed even if the lower 32 bits are unchanged (see line 7). This is done by the `write` function.

|    |  |    |   |
|----|--|----|---|
| a) | <pre>1  val sem-cmovcc insn cond = do 2    size &lt;- sizeof insn.opnd1; 3    dst &lt;- lval size insn.opnd1; 4    dst-old &lt;- rval size insn.opnd1; 5    src &lt;- rval size insn.opnd2; 6 7    temp &lt;- mktemp; 8    mov size temp dst-old; 9 10   _if cond _then 11     mov size temp src; 12 13   write size dst (var temp) 14 end</pre> | b) | <pre>1  t0 =:32 B 2  if(FLAG.6) { 3    t0 =:32 A 4  } else { 5  } 6  B =:32 t0 7  B.32 =:32 0</pre> |
|----|--|----|---|

Figure 4.2.: The translator function a) and a translation result b)

### 4.3. Development State

The GDSL toolkit contains a compiler for GDSL as well as decoders, semantic translations and optimizations written in GDSL. The benefit of specifying optimizations in GDSL is that they can be re-used for any input architecture since they operate only on RReil. Besides a few instruction decoders for 8-bit processors, the toolkit provides an Intel x86 decoder for 32- and 64-bit mode that handles all 897 Intel instructions. In terms of translations into RReil, it provides semantics for 457 instructions. Of the 440 undefined instructions, 228 are floating point instructions that are not handled since floating point computations currently are not included in the analysis performed on the resulting RReil code. Many of the remaining undefined instructions would have to be treated as primitives as they modify or query the internal CPU state or because they perform computations whose RReil semantics is too cumbersome to be useful (e.g. encryption instructions).

**Part III.**

**Validation**





## 5. Introduction to Validation

The Intel instruction set is large and complex. Furthermore, the Intel Software Developer's Manual [2] occasionally is vague. For this reason, programming errors are very likely to be present within the decoder specification as well as within the semantic translation specification which is used to translate the decoded instructions into RReil. Handwritten unit tests are cumbersome to implement because of the vast number of instructions and their complexity. Moreover, they are error prone because they rely on the ability of the programmer to construct correct reference values for the translated semantics. In order to address this problem, an approach is needed that allows to generate a large number of test cases (i.e. Intel instructions including their parameters) and automatically obtain reference values for the effects of the instructions. To this end, Intel instructions are generated and executed on the actual system processor. Afterwards, the effects of the execution are analyzed and compared to the effects of the semantics obtained from the translator. That way, the correctness of the decoder and the semantic translator can be tested autonomously.

The following section gives an overview of the approach used to create and validate a test case. Section 5.2 explains limitations to the level of correctness that can be achieved using the presented validation strategy. Finally, Sect. 5.3 gives an overview of the subsequent chapters.

### 5.1. Approach

Running a test case consists of the following steps which are performed in sequence:

1. **Test case generation**

As mentioned above, a test case starts with the generation of a byte sequence that represents an Intel instruction, the *instruction under test*. The byte sequence is then decoded and translated into RReil using the decoder and semantic translator specifications from the GDSDL toolkit.

2. **Code inspection**

Next, all accessed registers<sup>1</sup> are gathered from the translated code. The information is later used to reserve the required registers for execution of the instruction under test and assign appropriate values to them. Thus, even though the correctness of this translation is the primary test object, some basic assumptions about the correctness of the RReil code are made. Essentially, it is important for the translated code to write to the same registers and memory locations as the instruction under test during its execution on the processor, because it is not possible to compare the values of every register and every memory location before and after the execution of the instruction.

---

<sup>1</sup>Accessed memory locations are not yet considered because, in general, the memory addresses cannot be directly extracted from the semantics.

For the most coding errors, it is sufficient for the RReil code to modify an arbitrary register or memory location for the tester to detect a wrong translation; the test will, however, falsely succeed in case a subset of the written registers and memory locations, that are modified by the instruction under test, are modified by the RReil code in a correct way.

### 3. Testbed function generation

The set of registers obtained from the semantics is used to generate the so-called *testbed function*. The testbed function prepares the execution of the instruction under test by backing up all accessed registers on the stack and setting its input registers to values copied from the so-called *virtual registers* located in memory. The virtual registers are contained in a data structure referred to as the *execution context*. The testbed function then executes the instruction under test. After the execution, all registers taking part in the instruction test are written back to their location in memory, i.e. to the corresponding virtual register. Finally, all backed up registers are restored.

### 4. Mapping of the testbed function

A mapping to an executable memory region is created in order to store the testbed function. After its generation, the testbed function is written to that memory region, but not yet executed. While storing the function in memory, the address of the instruction under test is determined; it can be used to calculate a correct value for the program counter at the time of the execution of the instruction under test (as described by Sect. 11.3).

### 5. Initialization of virtual registers

The operations performed by the simulation of the RReil code and the execution of the instruction under test on the machine both are performed using data from the virtual registers. When it comes to the execution of the instruction under test, the values of the virtual registers are copied to the system registers beforehand and written back to the respective memory locations afterwards. Two independent instances of the execution context data structure exist for both the simulator and the actual processor. Before the simulation of the RReil code and the execution of the instruction under test, the virtual registers are initialized. Initialization means copying previously generated random values to them. Accessed memory regions are not yet initialized; this is done during the simulation of the RReil code.

### 6. Simulation of the RReil code

Using the initialized virtual registers, it is now possible to simulate the RReil code. During the simulation, all memory accesses are recorded. The data for read accesses is again generated using a random number generator. Besides the virtual registers, the execution context also stores accessed memory regions and their data. All register operations are performed on the virtual registers.

### 7. Execution of the testbed function

After determining the effects of the RReil code on the system registers and memory, the effects of the execution of the instruction under test are obtained. In order to

do that, mappings<sup>2</sup> for the accessed memory addresses are set up (as described by Sect. 13.1) and the read memory is filled with the same data as used during the simulation. Then, the testbed function is called. As described above, the testbed function cares for the initialization and the write back of accessed system registers. Most importantly, it also executes the instruction under test. Upon return of the testbed function, all written memory is copied back to the execution context and the mappings are removed.

## 8. Evaluation

During the simulation and the execution, both the processor and the RReil simulator altered their *system states*, that is, (virtual) registers and memory. These modifications were recorded and can now be compared. The test case succeeds if the changes match and fails otherwise.

The presented strategy has some limitations as pointed out by the next section.

## 5.2. Limitations

The validation approach is not able to prove the total correctness of the decoder or the translator. This is first of all caused by the amount of instruction and parameter combinations possible - it is infeasible to systematically test all of them. Furthermore, the state of the system cannot be examined completely. Instead, some assumptions about the correctness of the translation are made. For example, the addresses of modified memory cells are found using the output of the translator rather than including the whole memory in the test. There also are instructions that effect changes to processor internal registers that cannot (or at least not easily) be accessed for comparison. Finally, since the instructions are generated without paying attention to details of the Intel instruction format (this is described by Sect. 6.2), decoding failures cannot be directly used to deduce programming errors in the decoder. It is possible to validate the correct identification of invalid instructions by executing failed byte sequences on the processor. In case the byte sequence in fact does not correspond to a valid instruction, an exception is generated by the processor. However, this approach does not allow to reliably detect that a prefix of the byte sequence in question corresponds to a valid Intel instruction. This is problematic since the instruction generator appends additional bytes to the byte sequence that can be used as immediates if needed. Therefore, decoding failures are currently not examined further.

## 5.3. Part Overview

After this introduction, the next chapter describes the generation of Intel instructions used as test objects. Thereafter, Chap. 7 illustrates the main component of the validation software which controls the different steps taken for an instruction test. Chapter 8 describes the inspection of the semantics obtained from the GDSL toolkit. This results in sets of accessed registers. The data structure used for storing the virtual registers and accessed

---

<sup>2</sup>A memory mapping maps the virtual address used by the application to some hardware address; the actual hardware address used is unimportant here.

memory locations, the execution context, is detailed by Chap. 9. Afterwards, Chap. 10 describes the generation of the testbed function. Before Chap. 12 details on the simulation of the semantics and Chap. 13 illustrates the execution of the instruction under test, Chap. 11 explains the steps that are necessary to take in order to prepare the execution context. Chapter 14 describes the evaluation of the test result. Finally, examples and statistics are presented by Chap. 15.

## 6. Generating Intel Instructions

In order to validate the correctness of the translation of machine code, appropriate instructions need to be obtained somehow. There are different approaches for that: One might think of using the output of compilers, that is, taking the instructions from binaries. However, compilers are likely to use a limited set of instructions. Furthermore, the instruction types and parameters depend on the program selected - a lot of instructions of the Intel architecture have got a very special purpose and, for this reason, cannot be found in the binaries of standard software. In order to address these problems, instructions are instead generated randomly. This chapter describes the implementation of a simple generator that outputs byte sequences that correspond to correct Intel instructions with a probability of at about 50%.

The chapter starts with a brief overview of the Intel instruction format. Thereafter, the structure of the generator is described in Sect. 6.3. Sections 6.2 and 6.4 address the level of correctness (in terms of the fraction of valid Intel instructions) achieved by the generation.

### 6.1. The Intel Instruction Format

The description of the instruction generator uses a lot of terminology that is related to the Intel instruction set. For a better understanding, the Intel instruction format is briefly explained in the following. The information is taken from the Intel Software Developer's Manual. Basically, there are two different instruction formats (other encodings are possible; special cases are not considered) as detailed in the next sections.

#### 6.1.1. The Classic Instruction Format

A schema using the classic instruction format has got a structure as illustrated in the following. It consists of the elements shown in the order given:

|               |            |        |        |     |              |           |
|---------------|------------|--------|--------|-----|--------------|-----------|
| Legacy Prefix | REX Prefix | Opcode | ModR/M | SIB | Displacement | Immediate |
|---------------|------------|--------|--------|-----|--------------|-----------|

Every element is comprised of zero or more bytes. The purpose of each element is explained below.

##### 1. Legacy prefixes

Legacy prefixes are used to configure an instruction. Each prefix consist of one byte. Among others there are the *operand size* prefix, the *address size* prefix, the *repeat* prefix and the *repeat not zero* prefix. The former two are used to set the size of the operands and the length of addresses, the latter two allow an instruction to be repeated automatically. Legacy prefixes are optional.

### 2. The REX prefix

The REX prefix is used to access new features introduced with the 64 bit extension of the instruction set architecture. Specifically, the REX prefix can be used to switch to an operand size of 64 bits or to access additional registers (`r8 - r15`). The REX prefix is optional.

### 3. The opcode

The opcode encodes which instruction is selected. It has got a size of one to three bytes.

### 4. The ModR/M byte, the SIB byte, and displacement bytes

The operands of an instruction are encoded using the ModR/M, SIB, and displacement bytes. These bytes encode whether to use registers or memory locations and how addresses are calculated. The SIB and displacement bytes are optional and only used for memory accesses. Certain bits of the ModR/M byte are used as an opcode extension by some instructions. The ModR/M byte is mandatory for many instructions.

### 5. Immediates

Several instructions use immediate bytes. Immediates usually have got a size of one to four bytes; in rare cases up to eight bytes are used. Whether immediate bytes are used or not depends on the operation code.

#### 6.1.2. The AVX Instruction Format

An instruction using the AVX instruction format uses the VEX prefix. Since the VEX prefix contains other prefixes and parts of the opcode, AVX instructions are encoded slightly differently:

| Legacy Prefix | VEX Prefix | AVX Opcode | ModR/M | SIB | Displacement | Immediate |
|---------------|------------|------------|--------|-----|--------------|-----------|
|---------------|------------|------------|--------|-----|--------------|-----------|

Some elements at the beginning of an instruction changed in comparison to the classic instruction format; the differences are explained below.

#### 1. Legacy Prefixes

Legacy prefixes can be used in combination with AVX instructions. It is, however, important to note that some legacy prefixes are part of the VEX prefix. These legacy prefixes must not be used in combination with AVX instructions outside the VEX prefix.

#### 2. The VEX Prefix

The VEX prefix is the identifying characteristic of AVX instructions. It combines certain legacy prefixes, the REX prefix, parts of the opcode and additional register operands.

#### 3. The AVX Opcode

The AVX opcode determines the type of the instruction to be used. Parts of the opcode are, however, included in the VEX prefix. Thus, when it comes to AVX instruc-

tions, opcode bits can be found in the VEX prefix, the opcode itself and the ModR/M byte.

## 6.2. Output Correctness vs. Implementation Simplicity

The most straightforward way to generate instructions randomly is to simply use random byte sequences. Unfortunately, this leads to lots of invalid instructions. The approach is particularly ill-suited for the Intel instruction format: The Intel format makes use of instruction prefix bytes that have a special function. These prefixes may be combined to configure the properties of an instruction. It is, however, very unlikely to generate a specific combination of prefix bytes randomly. For this reason, it is necessary for the generator to output instructions with a certain structure. However, one would need to put a lot of effort into the generator to make it produce valid instructions only. Therefore, a compromise is made which leads to an acceptable portion of valid instructions that also feature a certain diversity in respect to prefix combinations. This strategy is described in the following.

## 6.3. The Decision Tree of the Generator

The generator uses a certain kind of decision tree in order to generate instructions that are likely to be valid. While generating one instruction a path from the root of the tree to one of its leaves is traversed. Each node of the tree either is a

- generator node or a
- branch node.

Whenever a *generator node* is found, its corresponding generator function is called. The generator function then generates a specific byte sequence. For example, an arbitrary valid prefix combination is generated. Every generator node has got at most one child node. In case the child node exists, the traversal is continued at that child, otherwise the generator terminates. Generator nodes are depicted as shown in Fig. 6.1. *Branch nodes*, in contrast, do not generate any output. Instead, they represent a decision. For this reason, they may be the parent of an arbitrary number of children. There is a weight value associated with each child defining the likelihood for it to be selected as the next node on the path (see Fig. 6.2). A higher weight value for a child results in a higher probability that this child is selected for the continuation of the generation.

### 6.3.1. The X86 Generator Tree

The generator uses the tree shown in Fig. 6.3 in order to generate x86 instructions. It corresponds to the basic structure of x86 machine code. The task of each node is described in the following:

---

□ **Figure 6.1.: Generator node**

---

---

Figure 6.2.: Branch node

---

---

Figure 6.3.: X86 Generator Tree

---

### **vex?**

The *vex?* branch decides whether an AVX instruction is generated. In that case no legacy and no REX prefix is generated, but only a VEX prefix. Otherwise an arbitrary number of legacy prefixes and up to one REX prefix is prepended in front of the opcode.

### **legacy prefixes**

The *legacy prefixes* generator is used to generate legacy prefixes. Currently, only the *operand size* prefix, the *address size* prefix, the *repeat* prefix, or the *repeat not zero* prefix is generated. Prefixes can occur multiple times.

### **vex legacy prefixes**

The *vex legacy prefixes* generator is used to generate legacy prefixes for AVX instructions. Currently, only the *address size* prefix is generated. Prefixes can occur multiple times.

### **rex?**

The *rex?* branch decides whether the instruction is an instruction that needs the rex prefix or not.

### **rex**

The *rex* generator generates a rex prefix. The rex prefix must be followed by the opcode directly and, thus, its generation is separated from the generation of the legacy prefixes.

### **opcode**

The *opcode* generator generates one opcode taken from an opcode table.

### **modrm**

The *modrm* generator generates a valid ModR/M byte and (if needed) an additional SIP byte and displacement bytes. The ModR/M and SIP byte are the primary way to encode register and address operands for Intel x86 instructions. The generator also adds immediate displacements if required by the generated ModR/M/SIB combination. Since displacements are used for address calculations it is important to consider alignment (because a few instructions allow aligned accesses only). Because an instruction can read up to 16 bytes from memory at a time, the lower 4 bits of the displacements are zeroed.

### **immediate?**

The *immediate?* branch decides whether to generate an immediate or not.

### **immediate**

The *immediate* generator generates an immediate of 8 bytes. Even though most instructions need 4 immediate bytes (or less) only, there also are instructions that use



up to 8 immediate bytes (like `mov`, for example). Unnecessary immediate bytes do not prevent the correct decoding of the instruction; they are ignored.

The choice of weights is not motivated by any particular strategy. The weights were chosen in such a way that more generic instruction types are slightly preferred to special ones. Since additional immediate bytes do not hinder the decoding, adding such an immediate is reasonable and has thus got a high weight.

## 6.4. Invalid instructions

The above scheme does not completely prevent the generation of invalid instructions. That is caused by the independence of the different generators: It is possible that a VEX prefix is generated for an AVX instruction but later the opcode generator chooses an opcode that does not belong to an AVX instruction. Further incompatibilities between certain prefixes and opcodes could exist. In contrast, in the most cases the instruction bytes following the opcode should be valid (since the generator only generates valid ModR/M/SIB byte combinations and almost always adds immediate bytes). Altogether, this leads to a success rate of at about 50%.

This concludes the generation of instructions to be used as test objects. After their generation, the instructions are handed to the *tester* component which is the main component of the validation software. The next chapter describes the tester component.



## 7. The Tester Component

The tester component is the main component of the validation infrastructure. It performs all steps necessary to obtain a test result for a given byte sequence encoding an instruction. To this end, the instruction is first decoded and translated. Then, the RReil code is inspected and, thereby, the register usage is determined. Using this information, the testbed function is generated. The testbed function is responsible for the setup of the test environment for the instruction under test. Next, the execution contexts of both the processor and the RReil simulator are initialized and the simulation of the instruction semantics is run. After that, the testbed function is executed (which, in turn, executes the instruction under test). Finally, the effects on the system registers and memory of the execution of the instruction under test and the simulation of the semantics are compared.

The chapter begins with an illustration of the operation modes in the next section. After that, Sect. 7.2 describes possible test results.

### 7.1. Operation Modes

The tester component allows the calling module to pass configuration parameters which control the mode of operation. There are two important parameters concerning the choice of registers included in the test and the way errors are handled. The following subsections describe them.

#### 7.1.1. Choice of Registers

Basically, there are two possible strategies to choose the registers to be included in the test. On the one hand, registers can be chosen using the data gathered during the inspection of the RReil code only. This is referred to as the *minimalistic mode*. The advantage of this strategy is that it is fast and straight forward. There is, however, a disadvantage: The approach relies heavily on certain assumptions about the correctness of the translation - the test will only detect a wrongly translated instruction if the translated RReil code modifies a register or memory cell which the instruction under test modifies differently or not all. Thus, on the other hand, a more robust approach includes all registers<sup>1</sup>. This mode is called the *greedy mode*. It is important to note that even if the tester component treats all registers as if they were included in the test, some registers need to be excluded from the test for practical reasons. Section 10.2 describes this limitation regarding the choice of registers in more detail. A configuration option allows the configuration of the strategy to use.

---

<sup>1</sup>It is not possible to include all memory cells.

### 7.1.2. Crash handling

As already mentioned above, the validation relies on certain assumptions about the correctness of the semantic translation. If these assumptions are not met for an instruction (for instance, if the instruction accesses additional registers or memory locations), the effects of its execution cannot be predicted. This may result in an exception generated by the processor. Such exceptions are handled by the operating system only. In case the exception is caused by an application error, it is reported to the corresponding process using a so-called *signal*. Depending on the kind of signal and the state of the process, the process is either able to handle the error and continue its execution or not. In case the process cannot recover from the error, it is terminated. Such an abnormal program termination results in an inability to indicate the result of the test (i.e., the crash during the execution of the instruction under test) to the user. Furthermore, in case the validation software collects the results of a number of instruction tests, a crash results in an abort of the whole test suite. In order to solve this problem, the tester component is able to execute the instruction to be tested inside a child process. If the child process crashes, the tester is still able to continue its work. In case this *forking* of child processes is enabled, the tester component has got an additional result type which indicates a crash of the program during the execution. Since forking child processes is costly in terms of processor time and also makes debugging more complex, it can be disabled.

## 7.2. Results

The test of a random byte sequence can fail at many positions as it passes through the different components of the validation software. The tester component reports whether an error occurred or not. If an error is reported, the failing component and its error code also are part of the result value. The following errors can occur during the validation:

**Decoding error** This error type indicates that an error occurred during the decoding of the generated Intel instruction.

**Translation error** This error type indicates that an error occurred during the translation of the decoded instruction into RReil semantics.

**Simulation error** This error type indicates that an error occurred during the simulation of the semantics.

**Execution error** This error type indicates that an error occurred during the execution of the testbed function on the actual processor - such errors should only occur at the time the instruction under test itself is executed.

**Comparison error** This error type indicates that the effects of the instruction under test executed on the actual processor and the effects of the simulation of the translated semantics do not match.

The above errors are detailed in Sect. 14.2. If none of them occurs, the success of the test is reported to the user of the tester component.

This concludes the chapter about the main component of the validation software. The next chapter details on the inspection of the translated RReil code.

## 8. Inspecting RReil Code

The inspection of the RReil code is an important step in order to gather required information for the generation of the testbed for the instruction under test. The Intel x86 registers which are used by the the RReil code need to be known. Furthermore, their kind of use, that is, whether they are read, written, or used for address calculations, is relevant. This chapter describes the process of extracting register usage data from the translated RReil code.

### 8.1. Determining Register Usage

The x86 registers used are determined before the actual simulation. This is necessary to build the testbed function prior to the simulation and, thereby, to allow the simulation to use a correct value for the program counter. From this point on, two different kinds of registers are intermingled in the text: RReil registers and x86 registers. It is important to keep them apart - x86 registers are represented using RReil registers in the RReil code, but there is an arbitrary number of additional RReil registers.

#### 8.1.1. Required Information about the Accessed Registers

RReil registers are memory areas of infinite size which can be accessed at any offset. The number of bits accessed can be chosen freely. For this reason, it is important to not only know the RReil registers used, but also the corresponding offsets and access sizes. The design of RReil is helpful here: The language does not allow calculated register offsets or access sizes, but expects these values to be a fixed part in every statement, embedded as constant values. For this reason, it is in the most cases possible to exactly determine the register usage by considering each operand of each statement. The only exceptions to this are formed by the RReil conditional and loop statement - their handling is described in the next section. From the set of all RReil registers, only those that represent x86 registers are actually of interest at this point. Within one RReil register, only the part of the register that lies within the bounds of the corresponding x86 register is considered. All other accesses can be ignored since they are only needed for the internal state of the RReil simulator.

#### 8.1.2. Handling of Conditional Execution

The conditional and the loop RReil statements both execute child statements depending on the runtime values of registers. Therefore, the register accesses of all possible branches need to be considered. This, of course, may lead to the recording of phantom accesses that will not be performed during the execution of the instruction under test or the simulation run, respectively. Since the number of system registers that take part in the test is limited, this could theoretically lead to the omission of accessed registers during the preparation

of the execution of the instruction under test. However, in practice only few and shallow conditional RReil statements are used to choose between x86 registers. In fact, the usage of conditional statements for the selection of the kind of register access can almost only be found in the context of processor flags, but all flags are part of one single x86 register. Furthermore, the maximum number of system registers that are able to take part in the test is much greater than the number of registers used by regular instructions. It therefore is reasonable to join the sets of all register accesses of all branches of conditional RReil statements.

### 8.1.3. Read Registers

The first type of register access that is considered is the read access. Read registers need to be known in order to assign initialization values to them. The assigned values are used by both the RReil simulator during the simulation and the instruction under test during its execution. Additional registers can be filled with initialization values and be included in the test in order to increase the test reliability, but since it is not possible to reserve all registers to be used by the instruction under test during its execution, it is necessary to know the read registers in order to give them priority.

### 8.1.4. Written Registers

In addition to read accesses, registers can also be written. Written registers need to be considered when evaluating the test results. Differentiating between read and write accesses is not essential; it allows, however, more specific outputs and thereby eases the debugging in case an error occurs. If a register is both read and written, it is added to the read as well as the write set.

### 8.1.5. Dereferenced Registers

The third and less obvious access type is the dereferencing access. This kind of access occurs whenever a register is used in address calculations. Differentiating these accesses from read and write accesses is important due to the fact that dereferenced registers should not be initialized with the same random constants as other registers. Chapter 11 describes the generation of random values for registers in more detail.

Unfortunately, it is not as easy to detect dereferencing accesses as to detect read or write accesses. This is caused by the fact that the contents of an x86 register can be copied to an internal RReil register and later be used for address calculations. In this case, the dereferencing access would be classified as a read access. The following RReil example demonstrates the problem; the syntax of RReil is explained in Sect. 4.1:

```
1 t0 =:64 rax
2 ...
3 t1 =:64 [1*rdx + 2*t0 + 42]:64
```

While inspecting the RReil statement in line 1, it seems as if the register `rax` is read only. In case `t0` is not written before line 3, the value from `rax` is also used for the memory

```

1 t0 =:64 rax
2 t0 =:64 t0 + rcx
3 t1:64 = [1*rdx + 2*rbx + 42]:64
4 rax =:64 t0 - t1

```

Listing 8.1: RReil Example

dereference in that line. In order to be able to correctly deal with such situations, the values would need to be traced as they flow through the RReil program. Currently this is not implemented; instead only registers directly involved in a memory access are added to the set of dereferenced registers. The problem is addressed further during the initialization of the registers as described by Sect. 11.1. Dereferencing accesses have got a higher priority, that is, a register is added to the dereferencing set only in case it dereferenced and additionally read or written.

### 8.1.6. Example

In order to achieve a better understanding, List. 8.1 contains a small example RReil program. During the inspection, the following sets of registers<sup>1</sup> are built:

1. **Set of read registers**

The set of read registers consist of `rcx` and `rax`. The register `rax` is read by the statement in line 1 while the register `rcx` is read by the statement in line 2. Both registers are not used for memory address calculations.

2. **Set of written registers**

The set of written registers consists of `rax`, since it is written in line 4 and dereferenced nowhere. No other registers are written.

3. **Set of dereferenced registers**

The set of dereferenced registers consists of `rdx` and `rbx`. This is because `rdx` and `rbx` are used to calculate the memory address used by the load statement (the statement in line 3).

This finishes the inspection of the translated RReil code. The preceding chapters already mentioned the *execution context*. The following chapter describes the data structure in detail.

---

<sup>1</sup>It is important to keep in mind that only RReil registers representing x86 registers are considered.





## 9. The Execution Context

The *execution context* is a data structure that contains all system state information, i.e. the contents of registers and memory regions, relevant for an instruction test. Instances of the data structure are used for both the RReil simulator and the actual system processor. The simulator directly performs operations on its instance of that data structure. In contrast, the CPU operates on its own registers and addresses the (virtual) system memory directly, that is, it does not look an accessed memory address up in the execution context in order to find the memory region associated with that address. The testbed function takes care of copying in the *virtual registers* residing in the execution context to the processor registers before the execution of the instruction under test. It also restores the system registers to the virtual registers afterwards. Before the testbed function is called, the execution component prepares the respective memory addresses for their access by the instruction under test and, upon return of the testbed function, it copies the written memory regions back to their representatives in the execution context.

The representation of registers is explained in the following section. Thereafter, Sect. 9.2 details on the representation of memory accesses.

### 9.1. Representation of Registers

At a first glance, there are two main types of registers - x86 and RReil registers. Since RReil registers are memory regions of infinite size while x86 registers have got a fixed size, RReil registers implement the more generic concept. For this reason, as mentioned above, it is possible to implement an x86 register as a part of an RReil register. Therefore, from now on the generic term *register* can be applied to both an RReil register and an x86 register, since both are represented the same way in memory. From an organisational point of view, the execution context contains registers of the following types:

1. **X86 registers**

The set of x86 registers contains all registers introduced by the Intel x86 architecture (using the 64 bit extension).

2. **RReil virtual registers**

RReil virtual registers are used as platform independent RReil status flags. They are not relevant for this work.

3. **RReil temporary registers**

RReil temporary registers are registers used within RReil programs used to save intermediate results. They are only important for the simulation of the RReil program; since the Intel processor does not know these registers, their values are not compared during the test evaluation phase.

A register primarily consists of two fields:

### 1. The data of the register

The first field is a pointer to the data stored in the register. An RReil register has an infinite length; it is, however, sufficient to store those parts of the register in memory that have been written. Due to the way the registers are used (there are no big gaps between used parts of the registers), it is reasonable to use a contiguous memory region.

### 2. The domain of the register

Furthermore, it is important to store the domain of the register. The domain determines which parts of the register have got a defined value. Since the domain information needs to be stored for every bit, the data structure of the register contains a pointer to an additional memory region of the same size as the memory region used for data of the register. Section 9.1.1 details on the domain of registers.

### 9.1.1. The Domain of a Register

The domain information needs to be stored for each of the bits of the register, because the value of some of the bits can be undefined. This is caused by two circumstances:

#### 1. Reading of never written data

One way to induce undefined bits is to read data which has never been written. Let us, for example, consider the following RReil program:

```
1 t0 =:16 rax
2 t0.48 =:16 rbx.32
3 rcx =:64 t0
```

For the example, it can be assumed that all data bits contained in Intel registers <sup>1</sup> are defined at the beginning of the RReil program. The RReil internal register `t0`, however, has never been written and is, thus, initially undefined. The statements in line 1 and 2 fill parts of that register with data from Intel registers. The bit range from 16 to 47 is not written, though. Because of that, at the time the statement in line 3 copies the first 64 bits of `t0` to `rcx`, it undefines bits 16 through 47 of register `rcx`.

#### 2. Explicitly undefining bits

Another way in order to undefine bits is to explicitly undefine parts of a register. This is useful in case the manual of the processor itself specifies values to be undefined under certain conditions. Some instructions of the Intel x86 architecture produce undefined bits, primarily inside the `flags` register. RReil provides an expression (*arbitrary*) usable to limit the domain of an RReil register:

```
1 rax.16 =:32 arbitrary
```

The given example undefines 32 bits of register `rax` beginning at offset 16.

---

<sup>1</sup>Intel register is used as a synonym for a part of an RReil register that corresponds to an Intel register.

## 9.2. Representation of Accessed Memory

In addition to the information about the registers, the execution context also needs to keep track of all memory accesses. This is done using a list of accessed memory regions. Each entry of the list is a data structure describing the memory access. The following fields are contained in the data structure:

1. **Access type**

Each memory access has got an access type. In contrast to register accesses, read and write accesses are not distinguished. Instead, there are *regular* accesses (that is, a standard read or write access) and *jump* accesses. A jump access needs to be handled specifically by the execution component before the instruction under test is executed on the physical processor (further details can be found in Sect. 10.5 and 13.1).

2. **The data**

A regular memory access also contains a pointer to a memory region saving the data either read from memory or written to memory. Jump accesses do not need the data field since the memory at the jump target is neither read nor written by the branching instruction.

3. **The size of the data**

Memory accesses can access any amount of memory. Some instructions of the Intel x86 architecture contain instruction level iterations that process an arbitrary amount of memory. Thus, the size of the memory region needs to additionally be kept. Memory can only be accessed byte-wise. Jump accesses always have their size field set to zero since they do not use the data field.

4. **The accessed address**

It is important to keep track of the memory address generated for the access. This is because the processor will later use this address while executing the instruction under test. Therefore, a memory mapping <sup>2</sup> is created before the execution (this process is explained by Sect. 13.1).

This concludes the description of the execution context. The next chapter illustrates the generation of the testbed function which contains the instruction under test.

---

<sup>2</sup>The creation of a memory mapping makes an address usable by the user application; it maps the virtual memory address to a physical one used by the hardware.



## 10. Generating the Testbed

In order to be able to verify the correctness of the simulation of the RReil code, the instruction under test is executed on the actual processor. It is, however, necessary to perform some preparations before the execution since the registers of the processor and the memory of the system can be accessed by the instruction under test. To this end, the instruction is enclosed inside a *testbed function* that performs necessary setup and cleanup operations. The testbed function needs to make sure that accesses to registers due to the instruction under test do not interfere with the execution of the test application. Accesses to the system memory, in contrast, are taken care of by the execution component which will be described in Sect. 13. Moreover, the testbed function must set up the input to the instruction and record its output by copying the appropriate values from and to the execution context.

### 10.1. Overview

The testbed function is emitted as a byte sequence representing machine instructions. It complies with the standard C calling convention and can thus be easily called from C code. The function is structured into five logical sections. In the following, they are described in the order of appearance:

1. **Backup of registers**

Like standard C functions, the testbed function begins by backing up certain registers to the stack. This process of making the register available to be used by the testbed function is referred to as *allocating* the register.

2. **Initialization of registers**

The instructions subsequently emitted copy the input values from the execution context to the registers of the processor. This process is called the *initialization* of registers. Note that at the time the testbed function is called by the execution component, all registers involved in the instruction test have already been set to the generated values in the execution context of the processor.

3. **Initialization of the return register**

After the registers have been initialized, all preparations for executing the instruction under test are finished. However, in case the instruction is a jump instruction, the processor branches somewhere and needs a way to return to the testbed function afterwards. For this reason, a dedicated return address register is used in which the code address of the instruction following the instruction under test is stored. This register is initialized directly before the execution of the instruction.

4. **Execution of the instruction**

Finally, the instruction under test is executed. Therefore, all bytes previously consumed by the instruction decoder are inserted into the testbed function.

### 5. Writeback of registers

In order to track the changes made by the instruction under test, the registers need to be written back to the execution context which makes the changes visible to the tester component.

### 6. Register deallocation

As a last step, the allocated registers are deallocated. This is done by restoring the original value they have had before the call to the testbed function to them.

The remainder of the chapter is organized as follows. After discussing implementation limits when using *greedy mode*, Sect. 10.3 describes the allocation of the different register types. Section 10.4 describes how values are copied from the execution context into the allocated registers before the instruction under test is executed. The special case of testing jump instructions is detailed in Sect. 10.5. Section 10.7 illustrates how the results of running the instruction under test are written back to the execution context. The chapter concludes with a comprehensive example.

## 10.2. Choice of Registers

As described in Sect. 7.1.1, there are two participation strategies for registers - the *minimalistic mode* and the *greedy mode*. The testbed function generator accounts for that by also offering these two strategies. The difference to the main component is, however, that the testbed generator is unable to make all registers available for the tested instruction. This is because some registers are used by the testbed function itself (like the return address register) and are thus not available for the validation of the instruction. The information about the actual choice of registers does not need to be propagated back to the tester component, since all registers, which are not chosen for validation, are untouched in the execution context of the processor and contain the same value as the corresponding registers in the execution context of the simulator. That is because the registers of both contexts are initialized to the same values and the virtual registers of the simulator are not changed by the simulation since they are not included in the write set of the RReil code. Naturally, all known to be accessed registers (i.e. the ones that are accessed by the simulator) always have got the highest priority - they need to take part in the test in any case.

## 10.3. Allocation of Registers

Making a register available to be used by the testbed function is referred to as *allocating* the register. Two allocation strategies are distinguished. First, a register can be allocated at the beginning of the testbed function; that allocation strategy is called *static* allocation. If, in contrast, the register is allocated when needed at later positions, *dynamic* allocation is used. Depending on the type of the register to be allocated, different steps need to be performed for the allocation. The following subsections describe the steps necessary in order to allocate a register.

### 10.3.1. Allocation of Standard Registers

The allocation of standard registers is straight forward - they need to be pushed onto the system stack. The following example allocates the register `eax`. It is assumed that the stack pointer is not yet allocated, i.e. it points to the top of the stack owned by the current thread:

```
1 push eax
```

### 10.3.2. Allocating the Stack Pointer

The stack pointer can be used (almost) the same way as other general purpose registers. It is, for example, possible for the instruction under test to add a value to the stack pointer:

```
1 add rsp, 42
```

In case such an instruction is tested, the stack pointer needs to be allocated. For its allocation, it is not possible to simply save the old value of the stack pointer on the system stack, because after overwriting the stack pointer, the system stack is not accessible any more. Thus, an additional register is used - the *stack pointer backup* register. The value of the stack pointer is backed up to it. For instance, if the register `r8` is used as backup register, the allocation of the stack pointer is implemented as follows:

```
1 mov r8, rsp
```

After the allocation of the stack pointer, further accesses to the system stack need to take the allocation of the stack pointer into account. An allocated stack pointer does not contain a pointer to the system stack. Instead, it contains its *allocated value*. Thus, before a `push` instruction can be issued, the original stack pointer needs to be restored in order to be able to access the system stack. After the `push` instruction, the new value of the stack pointer needs to be written to the *stack pointer backup* register and the allocated value of the stack pointer register needs to be restored to it. Both is implemented using the `xchg` instruction which exchanges the values of its operands. For the following example, it is again assumed that the register `r8` is used as *stack pointer backup* register and that the register `eax` needs to be pushed to the system stack:

```
1 xchg r8, rsp
2 push eax
3 xchg r8, rsp
```

The first `xchg` instruction restores the pointer to the system stack and saves the allocated value of the stack pointer to the stack pointer backup register. Then, the `push` instruction is issued. Finally, the new pointer to the system stack is written to the stack pointer backup register and the allocated value is copied to the stack pointer using another `xchg` instruction.

### 10.3.3. Allocating the Flags Register

Many instructions read or modify the `flags` register. The allocation of the `flags` register is very easy, since Intel offers an instruction in order to push the contents of the flags

register to the system stack: `pushfq`. However, one needs to take care that many bits of the register are either not used or not readable by a user process. These bits are masked during the push operation, that is, their values are ignored.

### 10.4. Initialization of Registers

As already mentioned above, a choice of registers (depending on the mode of operation) needs to be initialized before the execution of the instruction. *Initializing* a register means copying a predefined value from the execution context to it. These values act as input values to the instruction under test. At the time the testbed function is generated, the initialization values do not need to exist yet. The testbed function generator only needs to know the addresses of the memory cells that will contain the values for the registers when the testbed function later is executed. To this end, the memory of the execution context is allocated before the testbed function is generated and a reference to that memory region is passed to the generator. The respective memory addresses are included in the testbed function as immediate values. For this reason, the testbed function is specialized for a particular execution context object.

#### 10.4.1. Initialization of Standard Registers and the Stack Pointer

The initialization of standard registers requires two instructions. The same holds for the stack pointer, since it can be used like general purpose registers with respect to the operations needed. The following example initializes the register `rbx` to a value found at the address `0x63c700`:

```
1 mov r9, 0x63c700
2 mov rbx, [r9]
```

Thus, the address is first loaded into the temporary register `r9` (which needs to be allocated). Then, the value at the address specified by register `r9` is copied into the register `rbx`. It might seem odd not to directly use the address as an immediate value in the memory dereference operation. The reason is that a memory address can only be specified using a 32 bit immediate value. Since, however, the memory addresses are created randomly using up to 46 bits, that size might not suffice. In order to keep the testbed function generator simple, the approach introduced above is used for all cases.

#### 10.4.2. Initialization of the Flags Register

Initializing the `flags` register is a bit more complex since no variant of the `mov` instruction is able to directly copy a standard register or a memory location to the `flags` register. There is, however, an instruction (`popfq`) that pops the top of the stack into the `flags` register. In the following example, the initialization value is located at address `0x63c640` and `r9` is used as a temporary register:

```
1 mov r9, 0x63c640
2 mov r9, [r9]
3 push r9
```



4 | **popfq**

Apparently, the initialization value is first copied from memory to the temporary register (`r9`). This is done the same way as already discussed above. The temporary register is then pushed to the stack and, finally, the top of the stack is popped into the `flags` register. It is important to keep in mind that not every bit of the `flags` register is actually used by the Intel architecture and some of the bits in use cannot be written by a user process. Furthermore, some of the bits have special functions, like, for instance, the *Trap Flag* (which is used for instruction stepping). In order not to accidentally set them, the initialization value of the `flags` register has to be chosen with caution, i.e. all special function bits need to be zeroed in the initialization value. Section 11.2 describes the choice of an appropriate value for the `flags` register in detail.

### 10.4.3. Initialization of the Return Address Register

The return address register is used to save the return address for jump instructions. Independently of which instruction is executed, the processor needs to continue to execute the testbed function after the execution of the instruction to be tested. Since the execution has to continue directly after that instruction, the return address can be calculated by adding the size of the instruction to its address. In the following example, the instruction to be tested has got a size of 3 bytes and register `r11` is used as return address register:

```
1 lea r11, [rip + 0x03]
2 ... //tested instruction
```

As indicated in the code, the instruction to be tested needs to immediately follow the initialization of the return address register. The `rip` register is the program counter and always contains the address of the next instruction. Hence, it contains the address of the instruction under test during the initialization of the return address register. By adding 3 bytes to the value of the `rip` register the address of the instruction following the instruction to be tested is determined. The `lea` instruction does not actually perform the memory dereference, but instead saves the effective address of the source operand to its destination operand, that is, to register `r11`. The return address register therefore is loaded with the correct return address.

## 10.5. Jumps

Jump instructions change the value of the program counter and thereby make the program continue at a different address. The code located at that address needs to confirm that the processor actually jumped to the right location and then return to the testbed function. The former is implemented by saving the current value of the program counter (which is the `rip` register) to the virtual program counter located in the execution context. After that, the return address register is used for the return. In the following example, the virtual program counter (which is located inside the execution context) has got the address `0x639eb0`. The registers `r8` and `r9` are temporary registers. Register `r10` is the return address register:

```
1 lea r8, [rip - 0x07]
2 mov r9, 0x639eb0
3 mov [r9], r8
4 jmp r10
```

The first step again uses the `lea` instruction in order to access the current value of the program counter. Since the `rip` register points to the next instruction, the size of the `lea` instruction needs to be subtracted from the value of the program counter in order to calculate the address the processor jumped to (in the example, the `lea` instruction has got a size of seven bytes). The calculated address is saved to the register `r8`. Next, register `r9` is loaded with the address of the virtual program counter and then used to save the value held in register `r8` to it. Finally, a `jmp` instruction uses the return address register (`r10`) to return to the testbed function.

It is important to note that the code generated is position independent, since it uses the program counter in order to determine its location at runtime. This is very useful, because the actual memory address the code needs to be written to is not known at the time the testbed function is generated. It is, just like all other memory addresses, determined during the simulation of the RReil code.

### 10.6. Memory Read and Write Accesses

Memory accesses do not need to be considered during the generation of the testbed function. This is because the memory management for the instruction under test is implemented by the execution component. This component prepares the memory before the execution of the testbed function and handles possible changes to the memory afterwards.

### 10.7. Writing back of Register Contents

After the execution of the instruction to be tested, its effects on the system registers need to be propagated back to the execution context. Again, depending on the mode of operation, either all registers previously allocated to be used by the instruction under test (if operating in the *greedy mode*) or only those which are part of the write set of the instruction (in case the *minimalistic mode* is used) are written back. As in the case of the initialization, different register types need to be handled differently. The approaches used are described in the following sections.

#### 10.7.1. Writing back of Standard Registers and the Stack Pointer

Similar to its initialization, the stack pointer can be handled the same way as general purpose registers in the following. Writing back registers mirrors their initialization: Again, the address of the virtual register located in the execution context is loaded into a temporary register first. Using this address register, the contents of the register to be written back are then copied to memory. In the following example, the contents of `rbx` are written back. The address of the corresponding virtual register is `0x63c700` and `r9` is used as temporary register:

```

1 mov r9 , 0x63c700
2 mov [r9] , rbx

```

### 10.7.2. Writing back of the Flags Register

Writing back the `flags` register can also be done analogously to its initialization: An instruction called `pushfq` is available allowing to push the contents of the register onto the stack. In the following example, `0x63c640` is the address of the virtual `flags` register inside the execution context. Moreover, `r9` and `r10` are used as temporary registers:

```

1 pushfq
2 pop r10
3 mov r9 , 0x63c640
4 mov [r9] , r10

```

First, the `flags` register is pushed onto the stack. The value is then popped into the temporary register `r10`. Next, the address of the virtual register is loaded into `r9` and, finally, this address register is used to write the contents of `r10` to memory.

Again, unused or inaccessible bits of the `flags` register need to be considered. Their values are either fixed or undefined. Thus, these bits cannot be included in the test. This can easily be implemented by marking them as undefined in the execution context of the simulator - all bits which are undefined in the execution context of the simulator are not included in the comparison of the system states, i.e. the registers and memory. Section [11.2](#) describes the choice of an appropriate value for the domain of the `flags` register. The `testbed` function does not need to address this issue further, but instead stores the value read from the `flags` register as it is.

## 10.8. Deallocation of Registers

At the end of the `testbed` function all allocated registers need to be deallocated, i.e. their original values need to be restored. The stack pointer is restored first, if it has been allocated. For this, the value of the *stack pointer backup* register simply needs to be copied to it. Afterwards, all other registers are restored. This is done using the `pop` instruction (in order to restore standard registers) and the `popfq` instruction (in order to restore the `flags` register).

## 10.9. A Complete Example

The above sections described the different steps taken during the generation of the `testbed` function. In order to ease the understanding of how these parts work together, the following example shows the complete `testbed` function for the instruction `add rbx, rax` (using the *minimalistic mode* for the choice of registers allocated to be used by the instruction under test):

```
1 push rbx
2 push rax
3 pushfq
4 push r8
5 push r9
6 push r10
7
8 mov r8, 0x63c700
9 mov rbx, [r8]
10 mov r8, 0x63c6c0
11 mov rax, [r8]
12 mov r8, 0x63c640
13 mov r8, [r8]
14 push r8
15 popfq
16 mov r8, 0x63c700
17 mov rbx, [r8]
18
19 lea r10, [rip + 0x03]
20
21 add rbx, rax
22 nop
23 nop
24 ...
25 nop
26 nop
27
28 pushfq
29 pop r9
30 mov r8, 0x63c640
31 mov [r8], r9
32 mov r8, 0x63c700
33 mov [r8], rbx
34
35 pop r10
36 pop r9
37 pop r8
38 popfq
39 pop rax
40 pop rbx
41
42 retq
```

The example can almost completely be understood using the explanations given in the sections above. One detail, however, needs additional clarification: Following the instruc-

tion to be tested, a number of *nop* instructions is inserted (line 22ff). Their task is to improve the handling of wrongly decoded instructions: In case the instruction to be tested is followed by additional bytes, these bytes are interpreted by the processor as one or more additional instructions. This only happens if the decoder is unable to decode the instruction correctly, but does not recognize that. In case these additional instructions do not cause an error (a signal, for instance), the *nop* instructions make sure that the rest of the testbed function is executed. Otherwise, the bytes of one or more of the instructions belonging to the testbed function following the instruction to be tested could end up as operands of the unwanted additional instructions which lie in between. In that case the processor would not interpret these bytes as instructions and, thus, the operations would not be executed.

This finishes the description of the generation of the testbed function. The next chapter details on the initialization of the virtual registers in the execution contexts of both the RReil simulator and the processor.



# 11. Preparation of the Execution Context

The execution contexts of the processor and the simulator contain all information relevant for the test regarding their system states - that is, the values of the registers and the memory. The register contents need to be set before the simulation. In contrast, accessed memory addresses are not yet known. Therefore, the corresponding memory regions are instead set during the simulation using callback functions (Section 12.4 describes the simulation of memory accesses). This chapter details on the initialization of the virtual registers contained in the execution context using appropriately generated values. For the remainder of the chapter, the supplementary keyword *virtual* often is omitted if the kind of register is apparent from the context.

The chapter begins with an explanation of the initialization of general purpose registers in the next section. Afterwards, Sect. 11.2 details on how the `flags` register is initialized whereas Sect. 11.3 describes the initialization of the program counter. Section 11.4 illustrates the way random numbers are generated for the registers. The consequences of the approach taken are discussed by Sect. 11.5. Finally, Sect. 11.6 presents an example.

## 11.1. Initialization of General Purpose Registers

The initialization of general purpose registers uses the information gathered during the RReil code inspection. Three separate cases are differentiated if the test software is operating in the *minimalistic mode*:

### 1. Read registers

Registers which are read only can be initialized using random bytes. However, it is possible for a value originating from a register contained in the set of read registers to be used in memory address calculations. For this reason, in case there is at least one memory dereference in the RReil code, all read accesses are treated like memory dereference accesses. In this case, shorter and aligned<sup>1</sup> values are used.

### 2. Written registers

The values of written registers are not used; thus, they can be left uninitialized. In order to ease the understanding of the output presented to the user, the current implementation sets all registers that are contained in the write set only to zero.

### 3. Dereferenced registers

The values of dereferenced registers (that is, addresses) also are generated using random values. There are some pitfalls, though; first of all, addresses must not use more than 46 bits (the rest of the more significant bits need to be set to zero) and it might

---

<sup>1</sup>An aligned value has its four least significant bits set to zero.

be necessary for an address to be aligned to (at most) 16 byte boundaries<sup>2</sup>. Since it is not known whether a specific instruction requires alignment, all memory accesses need to be aligned.

In addition to the *minimalistic mode* mode, the tester can also be configured to operate in the *greedy mode*. In that case, all registers that are not dereferenced are added to the read set. Therefore, written and not accessed registers are not distinguished from the read registers any longer.

### 11.2. Initialization of the Flags Register

The `flags` register is used as status register in the Intel x86 architecture - therefore, each bit of the register has got a dedicated purpose. Some of the bits are used to describe details about an arithmetic operation - for example, the carry bit saves the carryover generated by an addition at the most significant bit. These bits can be read and set by the user process and need to be treated the same way as values contained in other registers. Other bits, however, have got special purposes. Some of them are read-only or write-only - such status bits can be used by the user process as a straight-forward way to read or write the current processor configuration. Other bits are not accessible at all. Furthermore, there are unused bits. All these kinds of status bits do not play well together with automatic testing, since the validation software expects a randomly generated value assigned to a bit of a register to stay unchanged if the semantics of the instruction does not change the bit. Additionally, the setting of some bits of the status register might influence the configuration of the processor in an unintended way - for example, the *Trap Flag* is used by the debugger to make the processor stop after the execution of the next instruction. In order to address these problems, the initialization of the `flags` register consists of two additional steps:

#### 1. Domain limitation

Usually, the domain of registers corresponding to Intel registers is set in such a way that all bits of the register which correspond to a bit in the Intel register have got a defined value. In contrast, the domain of the `flags` register is set so that only bits which can be read and written safely by the user process (that is, only bits that do not cause any unintended side effects) get a defined value. If the instruction semantics only touches these bits<sup>3</sup>, the values of the undefined bits stay undefined during the simulation. Therefore, all status bits set to an undefined value are excluded from the value comparison at the end of the validation.

#### 2. Value masking

Setting the domain of the register does not influence the testbed function. That is, even though some bits of the register are undefined, the testbed function copies the whole register to the actual Intel `flags` register (if the register is allocated for the test) before the instruction to be tested is executed. The respective Intel instruction

---

<sup>2</sup>This applies if the current implementation of the Intel x86 instruction set architecture including the 64 bit extension is used.

<sup>3</sup>This holds for every instruction the validation software is meant to handle.



takes care not to overwrite any read-only or system bits; naturally, this does not hold for the special function bits which are writable by the user process, like, for instance, the *Trap Flag* mentioned above. In order not to overwrite such a bit with a random bit, all special function bits are set to zero in the generated random value before the initialization of the virtual `flags` register using a bit mask.

### 11.3. Initialization of the Program Counter

The program counter is included in the comparison of the register state during the evaluation of the test result. This allows for an intuitive way of validating a correctly taken jump: the RReil code implements the jump by loading the program counter with the jump target. The testbed function also updates the virtual program counter associated with the Intel `rip` register of the processor to the jump target in case of a successful jump (as described in Chap. 10 and 13). Relative jumps are implemented by adding a value to the program counter. In order to make the new values of the two program counters (the program counter of the processor and the program counter of the RReil simulator) match in that case, the program counter of the RReil simulator needs to be initialized to the value the real program counter (the Intel `rip` register) will have during the execution of the instruction under test. As defined by the Intel manual, the value of the program counter always equals the address of the instruction following the instruction currently in execution. Since the testbed function is already mapped to memory at the time the execution contexts are initialized, the address of the instruction following the instruction under test is known. It can therefore be used to set the virtual program counter to the correct value.

### 11.4. Biased Random Value Generation

Random values used for read registers should not be chosen completely arbitrarily. This is because some instructions behave extraordinarily if certain bit sequence are present in their operands. Important examples for such bit sequences with a special effect are the sequence of zero bits and the sequence of one bits. The Intel instruction `packusdw` helps to develop a better understanding of the problem. It is used to compress integer values - therefore, it converts multiple (signed) doubleword (that is, 32 bit) integers to unsigned word (that is, 16 bit) integers using saturation - thus, if the value of the doubleword integer does not fit into the word integer, either the highest or lowest possible word value is chosen as the resulting word integer. In order for a 32 bit signed value to fit into a 16 bit unsigned storage location, the high 16 bits of the value need to be either all zero (in case of a positive number) or 1 (in case of a negative number). All other bit sequences lead to an overflow. However, the chance of coincidentally generating a doubleword integer with a high word consisting of zero or one bits only is just  $2 : 2^{16} \equiv 1 : 32768$  - as a consequence, practically every test input leads to saturated results only. In order to counteract this effect and thereby achieve a better test coverage, the generation of special bit sequences (like, for instance, the ones mentioned above) is preferred.

### 11.5. Discussion

The initialization of the registers of the execution contexts is not purely random - first, because of the way random values are generated and second because all registers are initialized using address-friendly values in case any memory access occurs. As a consequence of the complexity of the Intel architecture, a clear proof that these approaches do not lead to a wrong picture of the correctness of the decoder or semantic translation is not supplied. However, the following arguments indicate that the chosen approaches actually make sense.

First, the manipulation of the generated random values is considered. Here, certain bit sequences are used in preference. However, the chances of generating such a special bit sequence are chosen to be small. Thus, the major part of the generated operands are filled with arbitrary bit sequences. Therefore, the effects of uniformly distributed random values are tested sufficiently.

The usage of address-friendly register contents for all register is, in contrast, more problematic. This is because addresses require a big portion of the register to only contain zero bits; such operands lead to a changed behaviour of the instructions processing them. In order to understand that, the `add` instruction which sums up its operands can be considered. The instruction sets the carry bit of the `flags` register in case a carryover occurs at the most significant bit of the result. This never happens if the most significant bit of both input operands is set to zero.

However, the way the Intel architecture is designed is helpful here. It is a CISC architecture - because of that, most instructions are usable in combination with a lot of different addressing modes and may also operate on operands of different sizes. As a result, even though the correct setting of the carry flag of the `add` instruction is never tested for the combination of 64 bit operands together with a memory reference, it is tested both without a memory access as well as with a memory access and smaller (32 bit) operands. Because of that, the test coverage is still adequate.

### 11.6. Examples

The following two examples illustrate the initialization approaches used. Both examples assume the tester to operate in the *minimalistic mode* - thus, only those registers actually used by the instruction given are considered. The first example shows a virtual register initialization for the Intel instruction `add rbx, rax` which adds its two operands and saves the result to the first operand:

```
Read registers:
Register B: ffffffffffffffff
Register A: ffffffffffffffff
Register FLAGS: 0000000000000884
Written registers:
Register FLAGS: 00000000000008d5
Register B: ffffffffffffffff
Dereferenced registers: none
```

```

Register IP: 00007f06c1e2105c [defined: ffffffffffffffff]
Register FLAGS: 0000000000040084 [defined: 0000000000244cd5]
Register A: 4b5d000137b47f92 [defined: ffffffffffffffff]
Register B: ec936e00e0ff4d52 [defined: ffffffffffffffff]

```

The listing starts by recapitulating the results of the RReil code inspection. It yields a bit mask (given as a hexadecimal number) for each register - a 1 at a position in the binary representation of the mask indicates that the bit at the respective position in the register is accessed, a 0 indicates it is not accessed. The output of registers without at least one accessed bit is suppressed. Naturally, both input operands are read and the destination operand (first operand) is also written. Furthermore, a few bits of the `flags` register are read and written.

The second part of the listing shows the values generated for the registers and their domain. The domain is again given as a bit mask. All accessed registers (except the `flags` register) are fully defined in the bit range 0 through 63 (thus, the domain matches the size of the register). The domain of the `flags` register is set according to the associated mask used. Its value also is masked. Since the instruction does not access the memory, address-friendly initialization values are not used. The `ip` register is set to the proper address of the instruction following the instruction under test in the memory.

The second example is similar to the first one, except that it uses a memory access; the instruction `add [rbx], rax` is looked at:

```

Read registers:
Register A: ffffffffffffffff
Register FLAGS: 0000000000000884
Written registers:
Register FLAGS: 00000000000008d5
Dereferenced registers:
Register B: ffffffffffffffff

Register IP: 00007fb968c5c04f [defined: ffffffffffffffff]
Register FLAGS: 0000000000204890 [defined: 0000000000244cd5]
Register A: 000000f002c0b1b0 [defined: ffffffffffffffff]
Register B: 000000c118dd3410 [defined: ffffffffffffffff]

```

The register `B` now is part of the set of dereferenced registers, it is no longer written. Since the memory is accessed, memory-friendly values are used for the initialization: The lower four and the higher 24 bits<sup>4</sup> of the registers `A` and `B` are set to zero. The memory address used for the write access is not displayed since it is not yet known.

This finishes the preparation of the execution context. After the virtual registers have been initialized, the simulation of the instruction under test can begin. The next chapter describes the simulation in detail.

<sup>4</sup>This is because the value is zeroed bytewise; a more precise bit mask would allow the usage of a few more bits.



## 12. Simulation of the Execution of RReil Code

In order to compute the effects the translated RReil code carries out on the system registers and memory, the RReil code is simulated. The code consists of a list of RReil statements. The simulation processes one RReil statement of that list at a time. While simulating a statement, the execution context is directly read and modified. The right hand side of *assignment* and *store* statements contains an RReil operation. Operations do not only calculate new values, but also a domain for each value. Consequently, the RReil assignment statement also updates the domain information of a register. RReil operations include, among others, arithmetic, bitwise, and comparison operations. In the current implementation of the simulator, some operations are limited in terms of the maximum operand size. RReil also offers statements that enable memory accesses. Since the memory addresses are not known in advance, read memory regions are uninitialized (that is, they do not contain a value) at the time the simulation starts. Because of that, a callback function performs the initialization of the memory and the memory access itself.

The chapter starts with a description of the simulation of the different RReil statements in the next section. Thereafter, Sect. 12.2 details on how the domain information of registers is handled. Section 12.3 describes the simulation of the each RReil operation. Memory accesses are discussed in Sect. 12.4. Finally, not every sequence of RReil statements can be simulated - for this reason, Sect. 12.5 explains error handling.

### 12.1. RReil Statements

An RReil program consists of a list of RReil statements. During the simulation, the statements contained in the list are processed one at a time. The following subsections describe the different statment types and how they are simulated.

#### 12.1.1. The Assignment Statement

The assignment statement assigns the result of an RReil operation to a register. The simulation therefore first simulates the operation and then copies its result to the register. It is important to note that the result of an operation does not only contain the newly calculated value, but also a domain.

#### 12.1.2. Load and Store Statements

Load and store statements are used to access the memory. The load statement assigns the value of a memory cell to a register. The store statement, in contrast, executes an RReil operation yielding the value to be saved to memory. The addresses themselves are

given as linear expressions which need to be evaluated before the actual load or store can be performed. The load and store operations themselves are performed by invoking a callback function.

### 12.1.3. Control Flow Statements

RReil supports the conditional and repeated execution of RReil statements independent of the underlying address space. To this end, these control flow changes within the RReil program are not implemented using some kind of branch statement, but instead use dedicated RReil statements. Two control flow statements are available - the *if · then · else (ite)* statement and the *while* statement. The former executes one of two branches depending on a condition, the latter repeats the execution of its body as long as a condition holds. The condition can either be a comparison operation or a linear expression of size one. The different branches of the *ite* statement or the loop body, respectively, are given as a list of child RReil statements. The simulation first evaluates the condition and then recursively simulates the chosen child statement list if necessary.

### 12.1.4. Branching Statements

Branching instructions are used to change the control flow of the underlying machine program. Therefore, they update the program counter register of the processor to contain the address of the next machine instruction to be executed. In the simulator, a jump is modelled as special type of memory access to the target memory address. RReil offers two different branch statements - a conditional branch depending on a condition (which can either be a comparison operation or a linear expression of size one) and an unconditional jump. The unconditional jump is used for standard unconditional jumps, calls to subroutines and returns from subroutines.

## 12.2. Domain Information

All registers contained in the execution context do not only have got a value associated with them, but also a domain. As described by Sect. 9.1.1, the domain data works on the bit level, i.e. each individual bit can either have a defined or undefined value. RReil operations do not only calculate a new value given one or two operands, but also a new domain. The domain of the result can be larger or smaller than the domain of either of the input operands. In order to understand that, both cases are demonstrated using examples. The first example addresses the `shl` statement which shifts the value of a linear expression to the left:

|   |  |
|---|--|
| 1 | <code>destination =:32 source shl 4</code> |
|---|--|

The statement given shifts the register `source` four times to the left and saves the result to the register `destination`. During the shift, zero bits are inserted at the least significant bit of the result. Due to that, the domain grows by at most four bits. This happens if undefined bits are shifted out of the register. The shift does not affect any bits outside its range of operation (bits 0 through 31 in the example) - thus, in case a defined bit is

shifted out of the register, the domain does not grow. For instance, if the `source` register is defined in the bit range 0 through 16, the register `destination` will be defined in the bit range 0 through 20 after the operation. If, in contrast, the register `source` is defined in the bit range 0 through 30 before the operation, the size of the domain increases by one only.

The second example performs an addition of a register and an immediate value using the add operation:

```
1 destination =:8 source + 255
```

For the example, it is assumed that the `source` register is defined in the bit range one through seven and all these bits have got the value zero. All bits of the 8 bit result, however, depend on the value of the first bit of the input register - a value of zero would result in a sum of 255, while a value of one would produce an overflow and, thus, a sum of zero. Therefore, since the first bit of the input register is undefined, all result bits are undefined - the domain of the result is smaller than both the domain of the first and the second operand. The implementations of some of the RReil operations in the simulator are not exact in respect of the domain, that is, the domain of the result of the operation might be smaller than necessary for some inputs.

The domain information cannot be propagated to memory in case of a store statement, since memory cells do not have got a domain. That is because according to the Intel x86 architecture, the processor does not store undefined values. Therefore, the usage of an undefined value for a memory cell indicates an error in the semantic translation.

## 12.3. RReil Operations

All operations supported by RReil need to be implemented in the simulator. They operate on linear expressions which are evaluated before the actual operation (the next section regarding the linear operation provides more details about linear expressions). The evaluation of these linear expressions yields data structures which contain pointers to the data of the operands and their domain. These data structures serve as parameters to the actual operation implementations. The return value of an operation is of the same type. The following subsections describe the implementation of the different operations.

### 12.3.1. The Linear Operation

The linear operation primarily is used to allow for a single RReil register or immediate value to appear as a right hand side expression in an assignment or store statement. It therefore only has got a single linear expression associated with it. This expression is evaluated and the result is returned without modification. However, the linear expression can also contain additions, subtractions, and scales (which is equivalent to a multiplication with an immediate value). In case such a more complex operation inside the linear expression is applied, the respective implementation also used for the corresponding RReil operation is utilized. These operations are described below.

### 12.3.2. Addition and Subtraction

The RReil `add` operation adds two operands. In the simulator, the addition is performed bitwise. There is no limit to the size of the operands. The same applies to the subtraction since it is implemented by first calculating the two's complement of the second operand and then again performing an addition. The domain of an addition or subtraction is equal to the continuous range of defined bits in both operands starting from the least significant bit. In other words, the result of the addition is undefined from the first undefined bit in either operand on. As an example, an addition is considered. The first operand has got a domain of the bit range from zero to four, while the second operand only has got a domain of bits zero and one. Thus, just the lower two bits of the result are defined.

### 12.3.3. Multiplication, Division and Modulo

The multiplication (`mul`), division (`div` for unsigned division and `divs` for signed division) and modulo (`mod`) operations are implemented using the respective operators as offered by the C programming language. This allows for a simple implementation of these rather complex operations. The approach has got a drawback: the maximum size supported is 128 bits. Currently, the semantic translation does not need to perform these operations on operands bigger than that. The implementations of these operations also use a simple model for the domain - if at least the value of one bit of the input operands is not defined, all bits of the result are undefined; otherwise the domain of the result equals the range of bits the operation is performed on.

### 12.3.4. Shift Operations

RReil supports the three well-known shift operations `shl` (shift left), `shr` (shift right) and `shrs` (shift right signed). In their implementation, the shift amount is separated into an inter byte fraction (the number of complete bytes to shift) and an inner byte fraction (the rest of the shift amount, i.e. the shift amount modulo the size of a byte). The inter byte fraction is used as an offset into either the result data (in case of a left shift) or the source operand data (in case of a right shift). The result is initialized to -1 (in case of a shift right signed operation with a negative shift operand) or 0 (in all other cases). Next, the source operand data is copied into the result data (using the offset as described). Finally, the inner byte fraction of the shift needs to be performed. For this, the result is iterated through either starting at the least significant byte (in case of right shift) or the most significant byte (in case of left shift). In all cases, the inner byte shift can be performed locally by processing one byte after the other. However, the direct successor of a byte in the direction of the iteration is looked upon in order to determine the bits being shifted in. In case of a shift right signed operation, the sign only needs to be taken care of in the last step of the iteration.

In order to help the development of an understanding of the steps performed, Fig. 12.1 demonstrates a shift left operation. The shift amount is 21 bits. For this reason, the inter byte shift fraction is 2 byte and, thus, two zero bytes are appended, thereby chopping off the two most significant bytes since the size of the data does not change (the bytes are *shifted out*). After that, the individual bytes are moved 5 more bits as indicated in the



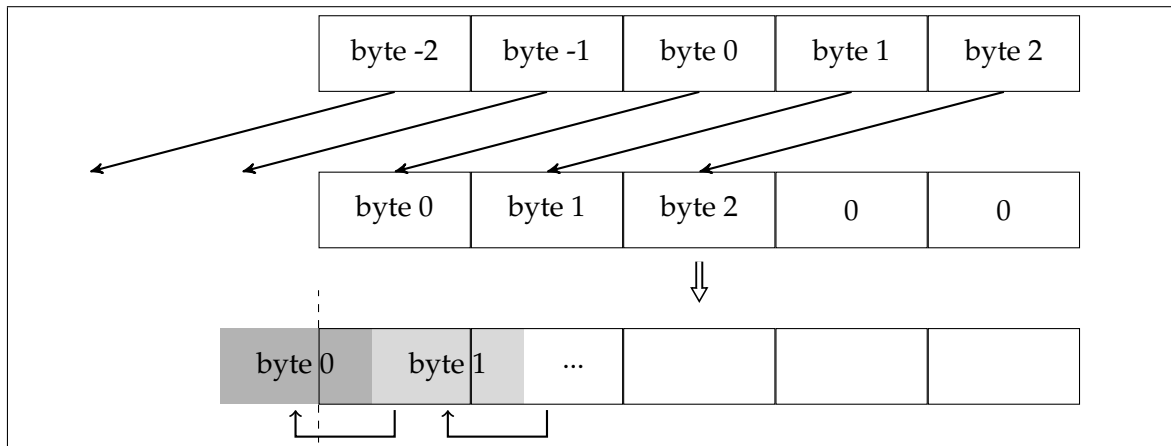


Figure 12.1.: Shift Left Operation

second step. Again, all bits shifted out of the most significant byte are cut off.

For the calculation of the domain of the result of a shift operation two cases are distinguished: Either the shift amount is fully defined or not. If it is not, all bits of the result are undefined. Otherwise, the domain of the source operand is first shifted (without taking care of the sign) into the direction of the shift operation and using the same amount as for the data. That way, the association between a bit of the source operand data and the corresponding bit the source domain data is preserved. Thereafter, the domain of the bits shifted in needs to be set. In case of an unsigned shift operation, all of them have got a defined value (zero). Otherwise, the sign bit of the original value of the source operand is considered. In case it is defined, all bits shifted in also are defined. Otherwise, they are marked as undefined.

### 12.3.5. Bitwise Operations

Bitwise operations combine each bit of the input operands independently. For this reason, their implementation is very straightforward: The input operands are divided into chunks processable by the physical machine and each of these chunks is handled separately. RReil offers the bitwise operations *and* (`and`), *or* (`or`), and *xor* (`xor`). These operations also are available in the C language and can thus be directly used for the resulting data chunks.

The domain of the result of a bitwise operation also is calculated bitwise: The value of the  $i$ th bit of the result is defined iff the value of the  $i$ th bit of both input operands is defined.

### 12.3.6. Comparison Operations

Comparison operations compare their operands arithmetically. RReil offers operations for signed and unsigned comparisons. The following table lists all available comparison operations, the corresponding mathematical expression, the signedness, a description, and the formula used by the implementation of the simulator for their evaluation:

| RReil Op. | Op.        | Signedness | Description            | Formula   |
|-----------|------------|------------|------------------------|---|
| =         | $a = b$    | Both       | Comp. for equality     | $\bigwedge_{i \in [0..s[} a_i \equiv b_i$         |
| $\neq$    | $a \neq b$ | Both       | Comp. for inequality   | $\bigwedge_{i \in [0..s[} a_i \neq b_i$           |
| $\leq_s$  | $a \leq b$ | Signed     | Comp. if less or equal | $((x \neg y) \& ((x \oplus y) \neg(y-x)))_{s-1}$  |
| $\leq_u$  | $a \leq b$ | Unsigned   | Comp. if less or equal | $((\neg x y) \& ((x \oplus y) \neg(y-x)))_{s-1}$  |
| $<_s$     | $a < b$    | Signed     | Comp. if less          | $((x \& \neg y) ((x \equiv y) \& (x - y)))_{s-1}$ |
| $<_u$     | $a < b$    | Unsigned   | Comp. if less          | $((\neg x \& y) ((x \equiv y) \& (x - y)))_{s-1}$ |

As shown in the table, all of these operations are implemented using bitwise operations. While this is trivial when it comes to the comparisons for equality and inequality, the formulas for the other operations are more complex - they originate from the book Hacker's Delight, Sect. 2.11 [12]. In the table, the notation  $x_i$  is used to access the  $i$ th bit of  $x$ . The variable  $s$  contains the size of the operands.

The domain of the result of comparison operations is calculated using the simple approach known from above: The one bit result value is defined if the values of both input operands are fully defined, otherwise it is undefined.

### 12.3.7. Extension Operation

Finally, RReil offers two extension operations - the zero (`zero-extend`) and the sign (`sign-extend`) extension. Extension operations prepend additional bits at the most significant bit of their operand. The zero extension always uses zero bits; the sign extension, in contrast, repeatedly duplicates the most significant bit of the operand, until the target size is reached.

The domain of the extended result depends on the kind of the extension operation. In case of a sign extension, the domain of the operand also is sign extended, i.e. if the value of the most significant bit of the input operand is defined, the bits used for the extension also have got a defined value, otherwise not. The domain of the result of a zero extension is equivalent to the original domain extended to the target size using one bits, that is, all bits prepended to the data always have got a defined value.

## 12.4. Memory Accesses

Memory accesses occur whenever the RReil program reads data from memory, writes data to memory, or changes the control flow of the underlying machine program. Since memory addresses and access sizes used by the RReil program are not known beforehand (sometimes this is not even possible: Both can depend on the RReil system state at the time of the access), the execution context cannot be prepared for the memory accesses before the simulation. For this reason, the actual memory access is outsourced to the calling component using a callback function. The callback function reads from memory, writes to it, or acknowledges a branch. The following subsections detail on how the accesses are handled by the callback function.

### 12.4.1. Read Accesses

The handling of read accesses depends on whether it is the first access to the given memory address or not. If the memory region has been accessed before, the memory read callback needs to read the old value from the execution context and return it. In contrast, if the current access is the first access to that address, new random data needs to be generated and both be saved to the execution context and be returned to the calling simulator. It is important to note that such newly generated data does not only need to be saved to the execution context of the simulator, but also to the execution context of the processor. That is because all data generated for reading needs to be preserved for the subsequent execution of the instruction on the processor. Since the Intel architecture does not allow memory indirect addressing <sup>1</sup>, the generated random data does not need to be address-friendly (Section 11.1 details on address-friendly random values). However, it is important to consider the preference of special bit sequences during the generation of random data as described by Sect. 11.4.

### 12.4.2. Write Accesses

Write accesses are easier to handle than read accesses since they do not need to generate any random data, but only check whether the given address has been accessed before. In that case, the corresponding entry in the execution context of the simulator needs to be updated to contain the newly written data. Otherwise, a new entry is inserted. Furthermore, the accessed address and access size need to be preserved in the execution context of the processor since this information is needed by the testbed function execution component and during the test evaluation. The write callback is not expected to return any value to the simulator.

### 12.4.3. Branching

A branch memory access is performed whenever the program counter register (associated with the Intel `rip` register) is updated to a new value (it is important to note that a jump to the current value of the program counter is ignored) using a *branch* or *cbranch* RReil statement. In the current semantic translation, a branch statement is only used as the last statement of the semantics of an instruction. The branch callback creates a new entry to be inserted into the list of memory accesses of both the execution context of the simulator and the processor. The branch entry does not contain any data, but only the address jumped to. The entry later is handled specially by the testbed execution component; this is described in Sect. 13.1.3.

## 12.5. Simulation Errors

Simulation errors are a sign of wrongly constructed semantics. They occur if the RReil program contains a statement sequence which would result in an ambiguous system state or which contains invalid statements. In most cases, it does not make sense to continue the

---

<sup>1</sup>That is, data read from memory never is used for an address calculation of another memory access within the same instruction.

simulation in case of such an error; thus, the simulation is aborted and an error message is returned to the caller, that is, the tester component. The following subsections describe the different error types.

### 12.5.1. Unaligned Memory Access

An unaligned memory access error occurs if the number of bits to be read from memory is not divisible by the size of a byte. Such an access is invalid because memory used in today's computers is unable to access single bits of memory cells, but instead addresses bytes or larger data units. Therefore, no Intel instruction semantics accesses individual bits inside a memory cell: If this is necessary, the whole byte (or more data) containing the bits in question is loaded to a temporary register, modified, and written back. The following example contains an RReil program trying to load 6 bits and thereby causing an unaligned memory access error:

```
1 t1 =:6 [rax]:64
```

### 12.5.2. Undefined Addresses

An undefined address error occurs if the address calculated for a memory access contains undefined bits. Such an access would access an arbitrary memory cell; this not useful, since the execution of the instruction under test on the actual processor most likely would access a different memory location. Furthermore, the semantics found in the Intel manual never uses undefined data for memory accesses. Thus, an undefined address indicates an error in the semantic translation. The following example RReil program stores data to an undefined memory location:

```
1 t0 =:32 arbitrary
2 [t0]:64 =:32 eax
```

The program first undefines 32 bits of `t0`. The value of the register is then used as address in the store statement. This generates an undefined address error.

### 12.5.3. Undefined Data to be Stored to Memory

An undefined store error occurs if the data to be stored to memory contains undefined bits. During the memory access, this information would be lost. The memory does not store the domain of its cells because the Intel manual does not store undefined data to the memory in its instruction semantics. Thus, again, such an access can only occur if the RReil program is a incorrect representation of the Intel semantics. The following example RReil program tries to store undefined parts of a register to memory:

```
1 t1 =:64 arbitrary
2 t1 =:32 eax
3 [rbx]:64 =:64 t1
```

The program first undefines 64 bits of the register `t1`. Then, it copies the value of `eax` to its lower 32 bits. Finally, it tries to store 64 bits of `t1`. Since the bits 32 through 63 have

not been written since they were undefined, the value to be stored to memory contains undefined data and an error is generated by the simulator.

#### 12.5.4. Undefined Conditions

An undefined condition error occurs if the condition of a conditional execution statement (*ite*), an iteration (*while*) or a conditional branch statment (*cbranch*) has got an undefined value. While such a situation could in theory be handled by tagging all written RReil registers as undefined in all child statements of the conditional statement, it makes more sense to forbid undefined conditions, since they are not needed by the semantic translation and, thus, again indicate some kind of error in the semantic translation. The following example RReil program uses an undefined condition in a conditional execution statement:

```
1 t0 =:1 arbitrary
2 if (t0) then {
3   ...
4 } else {
5   ...
6 }
```

This concludes the description of the RReil simulator. After the simulation of the RReil code, the instruction under test is executed on the actual processor. The next chapter describes all steps necessary to do that.



## 13. Execution of the Instruction Under Test

In order to obtain a reference value <sup>1</sup> for the effects of the semantics of the instruction to be tested on the system registers and memory, the instruction is executed on the physical processor. At that point in time, the execution context of the processor is already prepared, i.e. the virtual registers and memory cells contain the values to be used in the test. The testbed function is resident in memory and will take care of initializing the registers of the processor with the values from the execution context directly before the execution of the instruction under test. However, memory addresses accessed are only known since the simulation of the RReil code. The execution component needs to create memory mappings for the respective addresses and initialize the corresponding memory using the data from the execution context. Finally, the execution component takes care of handling signals received while executing the testbed function.

The chapter begins with an explanation of the memory mapping and initialization in the next section. Section 13.2 describes the execution of the testbed function and the cleanup of the memory mappings afterwards. The circumstances that lead to the triggering of signals and the types of signals are presented in Sect. 13.3. Section 13.4 briefly reiterates the handling of program crashes which is, strictly speaking, not a task of the execution component. Finally, Sect. 13.5 describes possible result values of the execution.

### 13.1. Memory Mapping and Initialization

The memory addresses accessed by instruction under test are expected to be known from the simulation of the RReil code. It is assumed that the Intel instruction to be tested will not access any further memory addresses during its execution; otherwise, either a signal is generated or the memory is corrupted in an unpredictable way <sup>2</sup>. In order to allow the instruction under test to access the memory using the given addresses, memory mappings that make the respective virtual addresses available to the executing process need to be created. The C function `mmap()` is able to map a number of memory pages for a specific virtual address and size. Naturally, it might be the case that the given address already is mapped; in that case the memory is in use and cannot be allocated for the test. There is no way to recover from that situation, the test needs to be aborted. Otherwise the new mapping is created. The following sections describe further steps needed to initialize the memory depending on the access type.

---

<sup>1</sup>A *reference value* is the comparison value for the test which is supposed to be the correct test result.

<sup>2</sup>In order to deal with such severe error situations, the validation software can be configured to outsource the execution of the instruction under test to a child process.

### 13.1.1. Initilization of Read Memory

Read memory regions need to be initialized with data read by the RReil program at the time the access occurred, i.e. during the simulation. The data is stored together with the address inside the execution context of the processor. The data needs to be copied from there to the target memory region.

### 13.1.2. Initilization of Written Memory

Written memory regions do not need to be initialized. Thus, at this point, they are simply skipped.

### 13.1.3. Initialization of Jumped at Memory

Memory addresses branched at also need to be initialized. Their presence in the execution context indicates the instruction under test to be a jump instruction. The processor might continue its execution at such a branch target. For this reason, the corresponding memory needs to be initialized using a program that makes the jump visible in the execution context of the processor and then returns to the testbed function. During the generation of the testbed function, the routine for jump targets has already been created and is now available to the execution component (Section 10.5 describes the generation of it in detail). Since the routine is position independent, the execution component simply copies the routine to the memory which is the jump target.

## 13.2. Execution of the Testbed Function and Cleanup

After the initialization of the memory, the testbed function is called. It administers the registers of the processor used for the test and executes the instruction to be tested. Upon return of the testbed function, the virtual registers of the execution context of the processor have already been updated with the changes induced by the instruction under test. The memory accessed, however, still needs to be propagated back to the execution context. This can be done by iterating over all memory accesses and copying the data at the respective addresses back to the memory image of the memory access data structure within the execution context. Afterwards, the mappings are removed using the `munmap()` C function.

## 13.3. Handling of Signals

Signals are events generated by the operating system. A signal can be caused by an exception during the execution of a program. Such an exception is, for example, triggered if an address is used which the processor does not know any mapping for. The processor reacts to an exception by jumping to an exception handler routine. Such an exception handler always is located inside the operation system kernel. In the example situation, the handler tries to find a mapping in the mapping table of the operating system which contains all



mappings<sup>3</sup>. If a mapping is found, the memory access of the user application is finished using that mapping - the user process does not notice that the exception was triggered in the first place. Otherwise, however, a signal (`SIGSEGV`) is sent to the process indicating the memory error. The user process is able to define handlers for signals. Such a handler is then invoked once a signal is received.

The execution component installs a number of handlers in order to react to possible errors during the execution of the instruction under test. The following sections describe the most important ones.

### 13.3.1. Segmentation Fault (`SIGSEGV`)

As indicated in the example above, a *Segmentation Fault* (`SIGSEGV`) is triggered in case of an invalid memory access. Such a memory error is very likely to happen if the RReil program fails to correctly calculate memory addresses used. In that case no mapping is created for the actual address accessed by the instruction under test and, thus, if the address is not already used by the same process for other reasons, its usage results in a memory error reported to the process by the `SIGSEGV` signal.

### 13.3.2. Illegal Instruction (`SIGILL`)

The *Illegal Instruction* signal (`SIGILL`) is triggered in case the user process tries to execute an illegal instruction. This can happen if the decoder erroneously decodes an illegal instruction generated by the random instruction generator. It is expected that the decoder reports an instruction to be invalid if the bytes handed to it do not represent an Intel instruction; in that case, the test byte sequence is ignored and another instruction is generated. In contrast, if the decoder fails to detect the invalid instruction, the decoding error is only recognized while trying to execute the instruction by the reception of the `SIGILL` signal. Another cause for the signal is the test of an instruction that is legal according to the Intel architecture, but requires a CPU feature that is not present in the processor configuration of the processor the validation software runs on.

### 13.3.3. Timer (`SIGALRM`)

The *Timer* signal (`SIGALRM`) is triggered in case of a timer expiration event. The timer is set using the `alarm()` function. The timer is used to cope with infinite loops caused by an invalid decoding or translation of an Intel instruction. As an example, the following instruction is considered:

|   |                     |
|---|---------------------|
| 1 | <code>jmp -2</code> |
|---|---------------------|

The instruction is assembled into the byte sequence `0xebfe`. It is a relative jump branching to an address calculated by adding the operand (`-2`) to the address of the instruction following the jump instruction. Thus, since the instruction has got a size of two bytes, it branches to itself. Because of that, the instruction repeats indefinitely. Of course, if the memory mapping phase correctly identifies the jumped at address to be in use, the test is

---

<sup>3</sup>The processor itself only caches some mappings inside the so-called translation lookaside buffer (TLB).

aborted before the execution of the testbed function and, thus, before the execution of the jump instruction. In case, however, this fails for some reason - for example, because of a wrongly calculated target address -, the testbed function is executed and never returns. By setting an alarm before the execution of the testbed function, a recovery from the error is possible. It is assumed that whenever the execution of the testbed function takes longer than one second, it is trapped in an infinite loop and needs to be aborted.

## 13.4. Intercepting Program Crashes

Even with signal handlers installed, it is possible that the execution of a test instruction leads to an irrecoverable error. These rather rare cases cause program crashes. This problem is not addressed by the execution component, but by the tester component. Section 7.1.2 describes the approach used in detail.

## 13.5. Execution Results

The execution component returns the result of the execution to the calling component. The result reports problems during the execution. In the following, all possible return values are listed and described briefly. An error during the execution leads to the failure of the test. The general error handling of the validation software is described in detail by Sect. 14.2.

### Mapping error

As mentioned above, the mapping of addresses can fail. In that case, the test cannot be performed; instead, the execution is aborted and a mapping error is returned to the tester component.

### Signal received

The reception of a signal during the execution of the testbed function indicates an error. It is not possible to evaluate the test results in that case. For this reason, the signal type is reported to the tester component.

### Success

If none of the above errors occurs, the successful execution of the instruction under test is reported to the tester component.

This finishes the description of the execution of the instruction under test. After a successful execution of the instruction the two instances of the execution context data structure which belong to the simulation and the execution contain different values regarding the virtual registers and the system memory. The next chapter describes how the execution contexts are used to evaluate the test result.

## 14. Evaluation of the Test Results

After the completion of a test case the results need to be evaluated. A single test may either succeed or fail. A failing test can either fail during the evaluation of the test results (i.e., because of deviating register or memory contents in the two execution contexts) or beforehand because of a decoding, translation, simulation, or execution error. The different error types are mostly caused by programming errors in the decoder or the semantic translation. There also are errors that do not indicate a programming error, but occur due to unrelated underlying conditions. An example for such an error is the memory mapping error that results from an already allocated memory address which cannot be used by the instruction under test.

There are two possible operational scenarios for the tester. First, it can be used to find errors in the decoder and translator specification. For this, the tester repeatedly generates test cases until an error occurs. Second, the tester can be used to collect statistical data. To this end, instruction tests can be run multiple times using different test byte sequences in order to obtain data about the frequency of certain error types for different instruction classes.

This chapter starts by describing the comparison of the effects of the simulation of the translated semantics to the effects of the execution of the instruction under test in the next section. After that, Sect. 14.2 details on possible reasons for a test to fail and also summarizes the origins of the errors described. Finally, Sect. 14.3 describes the statistical analysis of instruction tests.

### 14.1. Comparison of the Effects on Registers and Memory

The last step of an instruction test is the comparison of the effects on the system registers and the memory produced by the simulation of the translated semantics and the execution of the instruction under test. The effects are recorded in the respective execution contexts. If their contents deviate, the test fails because the real semantics as observed on the processor behave differently from the semantics given by the translated RReil code. For the test to succeed, both the accessed memory cells and all registers from the execution context of the simulator need to match the respective memory cells and registers from the execution context of the actual processor. The following two sections describe the approaches taken.

#### 14.1.1. Registers

The conformance of the registers is checked by comparing all values from all registers of the two execution contexts. It is not necessary to limit the check to the registers which actually took part in the test. This is because both the execution context of the simulator and the processor have been initialized to the same values. All registers which are changed

by the simulator also take part in the calculation of the reference value, i.e. their initial value is saved to the respective physical register before the execution of the instruction under test and written back to the execution context of the processor afterwards. Thus, all registers not considered during the execution of the instruction on the physical machine still contain their initialization value in both execution contexts.

During the comparison of the individual registers, it is important to pay attention to their domain. The values of undefined bits are allowed to vary from test run to test run, independent of whether they are generated by the real machine or by the simulator. Thus, their value is not useful in any way and needs to be ignored. In practice, the domain found in the execution context of the simulator <sup>1</sup> is used as a bit mask for the two register values to be compared.

### 14.1.2. Memory

The conformance of the memory accesses is confirmed by comparing all memory (write) accesses from the two execution contexts. Naturally, this only makes sure that those memory cells modified by the simulation of the RReil code are compared. Further memory cells which have been modified by the execution of the instruction under test on the processor are overlooked. This necessarily results in a loss of test accuracy. As pointed out in Sect. 5.2, the drawback is in the nature of the approach taken by the validation software.

## 14.2. Error Types

The validation of a byte sequence may fail for many reasons while the different steps of the test are performed. It is very important to precisely report the error which led to the abort of a test. Using that information, the respective specification (the decoder or the semantic translator) can be debugged or statistical information about the correctness can be collected. The following error types are distinguished; each error type has its individual causes and origins.

### 14.2.1. Decoding Errors

A decoding error occurs if the given byte sequence cannot be decoded by the decoder. Because of the way instructions are generated <sup>2</sup> such decoding errors are likely to happen and can in general be ignored, that is, by trying a new byte sequence.

### 14.2.2. Translation Errors

Translation errors occur if the semantic translation fails for a given instruction object. Since the semantic translator also works for instructions whose semantic translation is not yet implemented (the translator tries to guess the output operand of these instructions and assign an undefined value to it), translation errors should not occur and indicate a programming error in the semantic translation.

---

<sup>1</sup>The domain of the virtual registers contained in the execution context associated with the processor is not set since the processor does not calculate the domain separately while executing instructions.

<sup>2</sup>It is not made sure that each byte sequence generated actually corresponds to a valid Intel instruction.

### 14.2.3. Simulation Errors

Simulation errors occur if the simulation of the RReil code produced by the semantic translation fails. Simulation errors are caused by a faulty RReil program. Again, this is an indication of a programming error in the translator specification. Section 12.5 discusses simulation errors in detail.

### 14.2.4. Execution Errors

Execution errors occur if the execution of the instruction to be tested on the actual processor fails. This can be caused by a faulty decoding or semantic translation. However, it is also possible that the execution fails for other, unrelated reasons. Chapter 13 describes the execution of the instruction on the processor and also details on possible errors. Furthermore, in case *forking* is enabled (forking is described in Sect. 7.1.2), the instruction under test is executed using a separate process and, as a consequence, its execution can also cause a result value that signals a process crash.

### 14.2.5. Comparison Errors

A comparison error occurs if the final comparison of the effects on the system registers and memory cells of the simulation of the translated RReil code and the execution of the instruction under test unveils a mismatch regarding a memory cell or a register. Such a comparison error indicates a programming error in the semantic translation.

## 14.3. Statistical Analysis

The validation software facilitates two usage scenarios for test data collected. As mentioned above, the first one is the assistance offered for debugging the x86 decoder and semantic translator. However, besides that, it is also interesting to evaluate the dependability of the current implementation. To this end, the statistical analysis software runs a large number of test cases and outputs a summary of the data collected afterwards. The following sections describe the collected data and its evaluation.

### 14.3.1. Instruction Abstraction

For the statistical analysis, data about the results of the test cases is stored in a dictionary. The keys are derived from the test cases; a straight forward approach would use (a string representation of) the instructions generated for testing as keys. It is, however, necessary to abstract from the actual instructions. This is because it is neither efficient nor useful to collect data for every operand combination possible. For example, the data collected from the tests of the following two instructions can be merged:

|   |                           |
|---|---------------------------|
| 1 | <code>add rbx, rax</code> |
| 2 | <code>add rcx, rdx</code> |

The instructions both use register operands only. Therefore, the test results are stored in memory using the more generic string `add reg, reg` as key. Memory references

also are generalized; however, different addressing types are still distinguished. Memory operands are represented by the strings

- `[reg]`,
- `[reg + scale*reg]`,
- `[reg + scale*reg + imm]`, or
- `[imm]`.

### 14.3.2. Collected Data

For each abstract instruction, relevant data from the corresponding test cases is stored. This data contains the number of errors and the types of the respective errors. In case the tester component supplies additional information - the name of the signal which has been received during the execution of the testbed function, for example - this data is also aggregated. Finally, using a separate table, the number of failed instructions by CPU feature is logged - see the next section for more details.

The number of test cases to run can be configured by the user. In order to be able to continue testing after the occurrence of any kind of execution error and not lose statistical data, tests generally execute the respective instructions under test in child processes. All data collected is printed after the tests complete.

### 14.3.3. CPU Feature Dependence of Errors

As mentioned above, the number of failed test cases is additionally correlated to the CPU features required by the respective instructions. This is useful for two reasons. First, some CPU features are handled in a generic way by the decoder or the semantic translator - for example, *AVX* instructions share the decoder for the *VEX* prefix. By looking at the measurement data, programming errors in these shared components can be identified: In case a shared component used by a specific CPU feature is implemented incorrectly, the error rate presumably is significantly higher for instructions using that feature than in general. Second, the processor the validation software is run on might not offer every possible CPU feature. Because of that, some instructions might fail just because of an unmet feature requirement. This can be identified by looking at the error rate of the different CPU features - if all tests requiring a specific feature fail, the feature is probably not supported by the machine the tests are run on.

This concludes the description of the evaluation of the test result. The last chapter of this part presents a complete example and shows how the validation software can be used to fix bugs by describing a programming error detectable using a test case. Additionally, statistical data illustrating the current development state of the decoder and translator specifications is shown and discussed.

## 15. Examples and Statistics

Example inputs and outputs for every component of the validation infrastructure illustrating the different steps a test case takes from its generation to a successful validation are provided by the respective chapters. While this is useful to understand the components individually, this chapter provides complete examples. Furthermore, statistical data on the current development status of the decoder and semantic translator specification is discussed.

This chapter begins with a complete example for a test byte sequence that helps to understand the software as a whole in the following section. Next, Sect. 15.2 describes an error found in the translator specification using the validation software. Here, special emphasis is put on explaining how the output of the validation software helped to trace the bug. Finally, Sect. 15.3 contains statistical data collected using the current decoder and translator specification. The section also discusses the results with regard to the level of reliability the current specifications offer.

### 15.1. Example Test Case

The following example test case validates the byte sequence `0x48 0x29 0xc3`. The byte sequence has been generated by the Intel instruction generator. The decoder is able to decode the byte sequence; it outputs an instruction object which looks as follows (the illustration is simplified):

```
1 SUB {  
2   opnd1 = REG RBX,  
3   opnd2 = REG RAX,  
4   ...  
5 }
```

Thus, the instruction `sub rbx, rax` has been decoded. The instruction object is next handed to the translator which outputs a list of RReil statements representing the instruction semantics. The following listing shows a print representation of the AST using the RReil syntax.

```
1 t0 =:64 B - A  
2  
3 ...  
4 RREIL_ID_VIRTUAL_LTS =:1 B <_s:64 A  
5 FLAGS.7 =:1 t0 <_s:64 0  
6 FLAGS.11 =:1 RREIL_ID_VIRTUAL_LTS xor:1 FLAGS.7  
7 FLAGS.6 =:1 T0 =:64 0  
8 if(0) {
```

```
9  FLAGS =:1 B ≤u:64 A
10 } else {
11   FLAGS =:1 B <u:64 A
12 }
13 T4 =:8 T0
14 FLAGS.2 =:1 T4.7 =:1 T4.6
15 FLAGS.2 =:1 FLAGS.2 =:1 T4.5
16 FLAGS.2 =:1 FLAGS.2 =:1 T4.4
17 FLAGS.2 =:1 FLAGS.2 =:1 T4.3
18 FLAGS.2 =:1 FLAGS.2 =:1 T4.2
19 FLAGS.2 =:1 FLAGS.2 =:1 T4.1
20 FLAGS.2 =:1 FLAGS.2 =:1 T4
21 T5 =:64 T0 xor B
22 T5 =:64 T5 xor A
23 T5 =:64 T5 and 16
24 FLAGS.4 =:1 T5 =:64 0
25
26 B =:64 T0
```

It is important to note that the RReil code is not optimized in any way; this is noticeable by looking at the conditional statement in line 8 - the condition is always *false* and, thus, the conditional statement could be removed. The RReil code is quite extensive because of the status flag calculations. The actual subtraction is implemented by lines 1 and 26. The former subtracts the operands and saves the result in the temporary register `⌊0`. The latter writes the value of the temporary register to the destination operand. The detour using a temporary register is taken in order to allow the flag calculation to access both source operands and the result of the operation.

The translated semantics is next processed by the RReil inspection component. It identifies all accessed registers. For the above semantics, the following information is gathered:

```
1 Read registers:
2 Register B: ffffffffffffffff
3 Register A: ffffffffffffffff
4 Register FLAGS: 0000000000000084
5 Written registers:
6 Register FLAGS: 000000000000008d5
7 Register B: ffffffffffffffff
8 Dereferenced registers: None
```

In the listing, all registers with a least one accessed bit are listed. Within a register, the accessed bits are given as a bit mask. All 64 bits of both input operands are read; the destination operand additionally is written. Furthermore, some bits of the `flags` register are read and written. Since the instruction does not access the memory, no registers are dereferenced, i.e. used for address calculations.

Using the information gathered by the inspection, the testbed function is generated. For the example, the tester operates in the *minimalistic mode* of operation; thus, only the registers accessed according to the RReil code inspection are reserved to be used by the



instruction under test. The generation yields the following Intel assembly code:

```
1  push rbx
2  push rax
3  pushfq
4  push r8
5  push r9
6  push r10
7
8  mov r8 , 0x63c3c0
9  mov rbx , [ r8 ]
10 mov r8 , 0x63c380
11 mov rax , [ r8 ]
12 mov r8 , 0x63c300
13 mov r8 , [ r8 ]
14 push r8
15 popfq
16 mov r8 , 0x63c3c0
17 mov rbx , [ r8 ]
18
19 lea r10 , [ rip + 0x03 ]
20
21 sub rbx , rax
22 nop
23 nop
24 ...
25 nop
26
27 pushfq
28 pop r9
29 mov r8 , 0x63c300
30 mov [ r8 ] , r9
31 mov r8 , 0x63c3c0
32 mov [ r8 ] , rbx
33
34 pop r10
35 pop r9
36 pop r8
37 popfq
38 pop rax
39 pop rbx
40
41 retq
```

In the code, the different tasks of the testbed function can be identified. The function starts by backing up the registers touched during its execution (lines 1 through 6). Next, the registers accessed by the instruction under test are initialized (lines 8 through 17). Line 19

initializes the return address register. The instruction under test itself is executed in line 21. Afterwards, lines 27 through 32 write the modified registers back to the virtual registers inside the execution context of the processor. Finally, all used machine registers are reset to the value they have had before the call to the testbed function (lines 34 through 39) and the execution returns to the caller (line 41).

Before the simulation of the RReil code, the execution contexts (of both the simulator and the processor) need to be initialized. The instruction to be tested does not access the memory. Therefore, the constants generated do not need to be address-friendly. A valid initialization is shown in the following:

```
1 Register IP: 00007f14e590b05c [defined: ffffffffffffffff]
2 Register FLAGS: 0000000000000000 [defined: 0000000000244cd5]
3 Register A: 00b4006ba4e41891 [defined: ffffffffffffffff]
4 Register B: c35dbf2c68ea0033 [defined: ffffffffffffffff]
```

In the listing, the domain is also given using a bit mask - both input registers are fully defined; the `flags` register only has got a few defined bits. The value of the program counter (the `ip` register) is not arbitrary; instead, it contains the address of the instruction following the instruction under test inside the testbed function in memory (which is the correct value of the `ip` during the execution of the instruction under test).

Next, the simulation is run and, afterwards, the testbed function is executed. This results in new values for the different registers in both execution contexts:

```
1 CPU:
2 Register IP: 00007f14e590b05c [defined: ffffffffffffffff]
3 Register FLAGS: 00000000000000080 [defined: 0000000000244cd5]
4 Register A: 00b4006ba4e41891 [defined: ffffffffffffffff]
5 Register B: c2a9bec0c405e7a2 [defined: ffffffffffffffff]
6
7 RReil simulator:
8 Register IP: 00007f14e590b05c [defined: ffffffffffffffff]
9 Register FLAGS: 00000000000000080 [defined: 0000000000244cd5]
10 Register A: 00b4006ba4e41891 [defined: ffffffffffffffff]
11 Register B: c2a9bec0c405e7a2 [defined: ffffffffffffffff]
```

Again, the domain is given as a bit mask. The destination operand (register `b`) now contains the result of the subtraction; the source operand (register `a`) has not been changed.

As a final step, the results of the test are evaluated. For this, the two execution contexts are compared. As shown in the listing, all values match. Thus, the test of the instruction is successful.

## 15.2. Example Programming Error in the Translator

In the following, a programming error in the semantic translation of the Intel instruction `pblendvb` (Variable Blend Packed Bytes) is discussed. The signature of the instruction looks as follows:

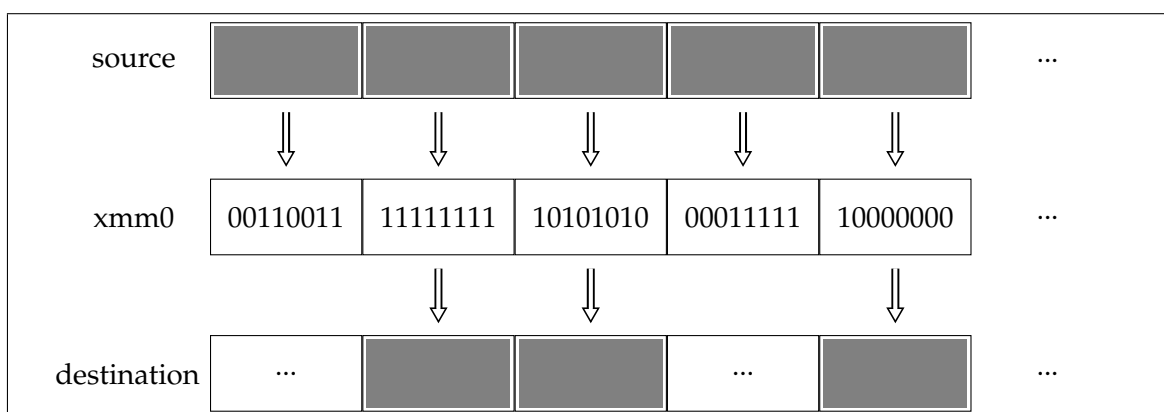


Figure 15.1.: Illustration of pblendvb

```
1 pblendvb xmm1, xmm2/m128, <xmm0>
```

Thus, the instruction has got three arguments, all of which are of size 128 bits. It conditionally copies bytes from its second operand to its first operand. The conditions are given by the third operand (which is fixed; it is always the register `xmm0`). For each byte of the source operand, the most significant bit of the corresponding byte in `xmm0` decides whether it is copied or not. If the bit has got the value *one*, the byte is copied, otherwise the byte in the destination operand is unchanged. An example execution of the instruction is illustrated in Fig. 15.1.

In the initial specification of the semantic translation, the following (erroneous) GDSDL monadic function handles the instruction:

```
1 val sem-pblend bit-selector opnd1 opnd2 opnd3 = do
2   size <- sizeof opnd1;
3   src1 <- rval size opnd1;
4   src2 <- rval size opnd2;
5   dst <- lval size opnd1;
6   mask <- rval size opnd3;
7
8   temp-src1 <- mktemp;
9   mov size temp-src1 src1;
10  temp-src2 <- mktemp;
11  mov size temp-src2 src2;
12  temp-mask <- mktemp;
13  mov size temp-mask mask;
14
15  temp-dst <- mktemp;
16  mov size temp-dst (imm 0);
17
18  element-size <- return 8;
19
20  let
```

```

21   val m i = do
22     offset <- return (element-size*i);
23
24     test-bit <- bit-selector element-size i temp-mask;
25     _if (/d (var test-bit)) _then
26       mov element-size (at-offset temp-dst offset)
27         (var (at-offset temp-src2 offset))
28   end
29   in
30     vector-apply size element-size m
31   end;
32
33   write size dst (var temp-dst)
34 end

```

The translation function first prepares the operands of the instruction (lines 2 through 6) for reading and writing. Next, lines 8 through 13 copy the two input operands into temporary registers since only these can be accessed using offsets<sup>1</sup>. Line 15 allocates a temporary register for the result, line 16 initializes the result to zero.

The main part of the semantics of the instruction can be found in lines 20 through 31. There, the `vector-apply` function is used - it calls a given monadic function (`m` in this case) for each element of a vector. Thus, `m` is called for each byte of the 16 byte operands. The monadic function `m` implements the recurring semantics for a single vector element. It reads the determining bit (line 24) and then copies one byte from the source register to the result register if the bit is set to one (lines 25ff). Finally, line 33 writes the value of the temporary result (vector) register to the destination operand.

Running the validation software for the instruction `pblendvb xmm2, xmm1` (with the implicit third operand `xmm0`) yields the following RReil code:

```

1  t0 =:128 xmm2
2  t1 =:128 xmm1
3  t2 =:128 xmm0
4  t3 =:128 0
5
6  if (t2.7) {
7    t3 =:8 t1
8  } else {
9  }
10 if (t2.15) {
11  t3.8 =:8 t1.8
12 } else {
13 }
14 ...

```

<sup>1</sup>That is because a generic *operand* does not need to be a register, it could instead be an immediate value or a memory location; in the concrete example, the instruction considered allows the second operand to be a memory location.

```

15
16 xmm2 = :128 t3

```

In the code, lines 1 through 4 correspond to the initialization of the temporary registers, lines 6 through 14 implement the conditional copying for each vector element (the code is shortened, there actually is a conditional expression for each of the 16 bytes), and line 16 finally implements the write back.

The test of the instruction fails with the following output:

```

1 Read registers:
2 Register XMM2: ffffffffffffffffffffffffffffffff
3 Register XMM1: ffffffffffffffffffffffffffffffff
4 Register XMM0: ffffffffffffffffffffffffffffffff
5 Written registers:
6 Register XMM2: ffffffffffffffffffffffffffffffff
7 Dereferenced registers: None
8
9 Register IP: 00007f2d4ed0e069 [defined: ffffffffffffffffffff]
10 Register XMM0: ac008cea00b42e8ce51b6c09e04ecf00 [defined: ffff ...]
11 Register XMM1: 388900704ddd16412a0011c83f27e7af [defined: ffff ...]
12 Register XMM2: 29f72200a290cc5e188e0011480081d9 [defined: ffff ...]
13
14 CPU:
15 Register IP: 00007f2d4ed0e069 [defined: ffffffffffffffffffff]
16 Register XMM0: ac008cea00b42e8ce51b6c09e04ecf00 [defined: ffff ...]
17 Register XMM1: 388900704ddd16412a0011c83f27e7af [defined: ffff ...]
18 Register XMM2: 38f70070a2ddcc412a8e00113f00e7d9 [defined: ffff ...]
19
20 RReil simulator:
21 Register IP: 00007f2d4ed0e069 [defined: ffffffffffffffffffff]
22 Register XMM0: ac008cea00b42e8ce51b6c09e04ecf00 [defined: ffff ...]
23 Register XMM1: 388900704ddd16412a0011c83f27e7af [defined: ffff ...]
24 Register XMM2: 3800007000dd00412a0000003f00e700 [defined: ffff ...]
25
26 Failing Registers:
27 XMM2
28 Failing memory addresses:
29 None
30 Result: TESTER_RESULT_COMPARISON_ERROR

```

Obviously, the values of the register `xmm2` differ (see lines 18 and 24). The comparison of the original value of the register to the output values of both the processor and the simulator gives insight into the problem: All vector elements not copied are zeroed in the output operand by the simulator but stay unchanged in the register associated with the processor. The reason can be found in the semantic translator function - line 16 of the listing initializes the output register to zero and the conditional statement in the lines 25ff fails to overwrite the result register with the respective bytes from the original value of the

destination operand since it lacks an `else` branch. The bug can be fixed by initializing the temporary result register to the value of the first operand (the destination operand) instead of zero:

```
1 val sem-pblend bit-selector opnd1 opnd2 opnd3 = do
2   ...
3
4   temp-dst <- mktemp;
5 #-mov size temp-dst (imm 0);
6   mov size temp-dst src1;
7
8   element-size <- return 8;
9
10  ...
11 end
```

In the listing, line 6 now uses the value of `src1` for the initialization of `temp-dst` instead of the immediate value zero.

### 15.3. Statistical Data on the Current Specifications

The following statistics on the current specifications have been collected on an Intel Core i5 processor. In order to obtain a representative amount of data, the software generated one million byte sequence for testing. As described by Sect. 14.3.1, the different instructions are mapped to more generic instruction types. The output of the software starts with information about each of the types that occurred:

| type                              | tot. | error type |      |       |       |        |
|-----------------------------------|------|------------|------|-------|-------|--------|
|                                   |      | trans.     | sim. | exec. | comp. | crash. |
| MOVAPD REG, [SCALE*REG+REG+IMM]   | 1    | 0          | 0    | 0     | 0     | 0      |
| VUCOMISS REG, [SCALE*REG+REG+IMM] | 1    | 0          | 0    | 1     | 0     | 0      |
| PHADDSW REG, [SCALE*REG+REG+IMM]  | 7    | 0          | 0    | 3     | 0     | 0      |
| VCMPSS REG, REG, [IMM], IMM       | 1    | 0          | 0    | 1     | 0     | 0      |
| PREFETCHW [SCALE*REG+IMM]         | 2    | 0          | 2    | 0     | 0     | 0      |
| VPSLLQ REG, REG, [REG+SCALE*REG]  | 1    | 0          | 0    | 1     | 0     | 0      |
| MOVZX REG, [SCALE*REG+IMM]        | 3    | 0          | 0    | 1     | 0     | 0      |
| MOVNTQ [SCALE*REG+IMM], REG       | 5    | 0          | 0    | 1     | 0     | 0      |
| ...                               | ...  | ...        | ...  | ...   | ...   | ...    |

The table does not list decoding errors, since it is not possible to associate an instruction type with a byte sequence if the byte sequence cannot be decoded. Crashes are listed separately since, strictly speaking, they are noticed outside the execution component and are therefore not counted as execution errors. Nevertheless, they only occur during the execution of the instruction under test. In total, the decoder was able to decode 526.252 (52,6%) of the byte sequences. The instructions were broken down into 4635 different instruction types.

| error type  | count  | relative | inner distribution  |        |         |        |
|-------------|--------|----------|---------------------|--------|---------|--------|
| none        | 306366 | 58,22%   |                     |        |         |        |
| translation | 4133   | 0,79%    |                     |        |         |        |
| simulation  | 52356  | 9,95%    | unaligned store     | 0,00%  |         |        |
|             |        |          | undef. address      | 0,00%  |         |        |
|             |        |          | undef. data (store) | 99,96% |         |        |
|             |        |          | undef. branch       | 0,04%  |         |        |
| execution   | 132757 | 25,26%   | mapping             | 6,88%  |         |        |
|             |        |          | signal              | 93,12% | SIGILL  | 55,38% |
|             |        |          |                     |        | SIGSEGV | 39,37% |
|             |        |          |                     |        | SIGBUS  | 4,26%  |
|             |        |          |                     |        | SIGFPE  | 0,94%  |
|             |        |          |                     |        | SIGTRAP | 0,05%  |
|             |        |          |                     |        | SIGALRM | 0,00%  |
|             |        |          |                     |        | SIGSYS  | 0,00%  |
| comparison  | 20025  | 3,81%    |                     |        |         |        |
| crashes     | 10615  | 2,02%    |                     |        |         |        |

Figure 15.2.: Test results

The tests resulted in the summary data shown in Fig. 15.2. The figure shows the different error types and their frequency. Most importantly, almost 60% of the tests succeeded. Only about 4% of the tests failed because of a deviation of the values of the registers or memory. The most notable error is the execution error - it occurred during more than 25% of the test cases. Among the execution errors, the two signals *SIGSEGV* (invalid memory access) and *SIGILL* (illegal instruction) are the most prominent causes. The former indicates an error in the semantic translation since it is caused by the lack of knowledge of an access to a memory address. The memory addresses are found during the simulation of the RReil code and, thus, it is probable for the RReil code to be faulty. The latter, in contrast, points to either an error in the decoder or the unavailability of an instruction. The decoder was able to decode the byte sequence, but the processor was not. Therefore, the decoder either accepts invalid instructions (which implies a programming error in the decoder) or the processor does not accept a valid instruction; that can be, for instance, due to a missing CPU feature.

About 10% of the tests resulted in a simulation error. Almost every of these simulation errors occurred because of an attempt to store undefined data. It is, however, important to note that the current translator handles unimplemented instructions by undefining their destination (which is assumed to be the first operand) operand. Thus, if the first operand of an unimplemented instruction is a memory operand, this results in such a simulation error. Therefore, the simulation errors are primarily caused by unimplemented instructions. A low number of test cases failed because of translation errors. Translation errors are caused by unhandled operand combinations in the instruction semantics. Finally, a few of the test cases caused the program to crash. Since the program crashed, it is hard to find out the exact reason for the error. A crash can happen if a signal handler cannot be called for some reason, for example due to a memory corruption or because the program did not register

| CPU feature           | number of tests | success rate |
|-----------------------|-----------------|--------------|
| NONE                  | 242105          | 66.883%      |
| AES                   | 1069            | 8.793%       |
| AVX                   | 32529           | 0.000%       |
| F16C                  | 22              | 0.000%       |
| INVPcid               | 67              | 0.000%       |
| MMX                   | 40032           | 67.613%      |
| CLMUL                 | 91              | 32.967%      |
| RDRAND                | 31              | 0.000%       |
| FSGSBASE              | 175             | 0.000%       |
| SSE                   | 39540           | 40.490%      |
| SSE2                  | 30020           | 39.616%      |
| SSE3                  | 2625            | 43.885%      |
| SSE4_1                | 4431            | 32.701%      |
| SSE4_2                | 455             | 21.978%      |
| SSSE3                 | 15126           | 60.934%      |
| XSAVEOPT              | 672             | 0.000%       |
| ILLEGAL_REP           | 91001           | 50.644%      |
| ILLEGAL_REPNE         | 94556           | 51.436%      |
| ILLEGAL_LOCK          | 745             | 8.590%       |
| ILLEGAL_LOCK_REGISTER | 25              | 0.000%       |

**Figure 15.3.:** Test results by CPU feature

a handler for that signal.

Figure 15.3 shows the number of tests and the success rate as a function of the CPU feature. The table shows that some CPU features cause all tests to fail; an important example is the *AVX* feature. Since the processor does not support it, all *AVX* instructions fail. This does not imply anything about the correctness of the decoding or of the semantic translation of *AVX* instructions. The same holds for other features that are not implemented by the current processor. Finally, there are some features in the table whose name begins with *ILLEGAL*. These features do not represent Intel CPU features, but instead indicate the ability for an instruction to accept a prefix which is not meant for it. For example, the `ret` (Return from Procedure) instruction allows to be used in combination with the *repeat* prefix, even though this prefix does not make sense for the instruction; it is ignored. Whenever an instruction is combined with such a needless prefix, it *requires* the respective pseudo CPU feature. The *ILLEGAL\_LOCK\_REGISTER* feature is used for a lockable instruction that is used in combination with the *lock* prefix and a register destination operand. The lock prefix is pointless in combination with register destination operands.



## **Part IV.**

# **Conclusion**



## 16. Conclusion

The decoding and semantic translation of binary code is an important step towards the static analysis of programs. This thesis presented the GDSL programming language that is specifically geared towards the implementation of instruction decoders. During the last two years, GDSL proved itself in the course of the implementation of an instruction decoder and semantic translator for the Intel x86 (including the 64 bit extension) architecture. By now, the GDSL toolkit features a complete decoder implementation and semantic translations into the RReil intermediate representation for at about half of the Intel instruction set. The remaining instructions are, for the most part, floating point instructions which are currently not handled by the corresponding analysis software which is also developed at our chair. Furthermore, some instructions perform highly complex operations or query the internal state of the processor; their translation needs to be implemented using primitives in the future.

The Intel x86 decoder and semantic translator both have got a size of at about 6000 lines of code. Writing such a great quantity of code inevitably leads to correctness issues. The lack of a debugger and the early development state of the compiler for the GDSL programming language made the problem worse. The thesis addressed this issue by presenting a validation approach that is able to detect programming errors within the decoder specification as well as within the semantic translation specification. The main idea of the test approach is to use the processor that runs the test suite itself to validate the correct decoding and semantic translation. To this end, byte sequences are generated randomly that are used as test cases. The generated byte sequences are later executed on the processor. The validation software records the changes to the system registers and memory that are caused by that execution and uses the data collected to confirm the correctness of the decoding and the semantic translation obtained from the GDSL toolkit. By executing the byte sequences on the actual system processor, a reference value for their effects is constructed which relies on an existing implementation of the underlying instruction set architecture rather than on a description of the architecture which may again contain errors and inaccuracies. Due to the automatic generation of test cases, the approach allows to run a multitude of test cases in order to achieve a good test coverage. Furthermore, the test approach allows for a detailed output in case of a test case failure which enables the programmer to precisely trace errors.

Finally, the thesis also discussed statistical data on the present implementation of the decoder and translator specifications. The data shows that currently at about half of the generated test cases succeed. The automatic generation of such statistical data enables the programmer to keep track of the development state and to confirm the reduction of programming errors as the implementation progresses.



# Bibliography

- [1] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA Framework for Binary Code Analysis. In *Computer Aided Verification*, LNCS, pages 165–170. Springer, 2011.
- [2] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Intel Corporation, Santa Clara, CA, USA, December 2011.
- [3] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In B. Robinet, editor, *International Symposium on Programming*, pages 106–130, Paris, France, April 1976.
- [4] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. CanSecWest, Canada, 2009.
- [5] A. Fox and M. O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Interactive Theorem Proving*, volume 6172 of LNCS, pages 243–258, Edinburgh, UK, 2010. Springer.
- [6] J. Kranz, A. Sepp, and A. Simon. GDSDL: A Universal Toolkit for Giving Semantics to Machine Language. 2013.
- [7] J. Lim and T. Reps. A System for Generating Static Analyzers for Machine Instructions. In L. Hendren, editor, *Compiler Construction*, volume 4959 of LNCS, pages 36–52. Springer, 2008.
- [8] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.
- [9] N. Ramsey and M. F. Fernández. Specifying Representations of Machine Instructions. *Trans. of Programming Languages and Systems*, 19(3):492–524, May 1997.
- [10] A. Sepp, J. Kranz, and A. Simon. GDSDL: A Generic Decoder Specification Language for Interpreting Machine Language. In *Tools for Automatic Program Analysis*, ENTCS, Deauville, France, September 2012. Springer.
- [11] A. Sepp, B. Mihaila, and A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In M. Pinzger and D. Poshyvanyk, editors, *Working Conference on Reverse Engineering*, Limerick, Ireland, October 2011. IEEE Computer Society.
- [12] Henry S. Jr. Warren. *Hacker’s Delight*. Addison-Wesley, Boston, Toronto, London, 2002.