Msc Thesis

Dynamic Hardware Binary Translator for ρ-VEX

A.E. Ntasios

CE-MS-2019-02

Delft University of Technology

Abstract

The last years, there has been a increasing trend in embedded system and FPGA implementations for greater flexibility and also, a rising adaptation of heterogeneous platforms. These platforms often include FPGAs and embedded cores side by side. ρ-VEX core, developed and maintained by the Computer Engineering group of TU Delft, is a VLIW processor mostly developed on FPGAs. On the other hand, popular embedded architectures include the established ARM architecture and the newly rising RISC-V architecture. In order for these architectures to communicate with the ρ-VEX core, a translation procedure has to be established. In this thesis, a hardware dynamic binary translator was designed, able to translate on-the-fly ARM and RISCV instructions to ρ-VEX instructions. The translator will enable heterogeneous platforms to be developed and also allow pre-compiled binaries for one (ARM/RISC-V) architecture to be directly ported in another one (ρ-VEX). This thesis provides a design process that focuses on two approaches: first is the minimization of the overhead resulting from the translation procedure, and the second is the minimization of the hardware alterations and/or hardware additions. These design choices were examined in the ρ-VEX core simulator. The simulations show that for overhead minimization, the resulting overhead can be as low as 1% for the RISC-V with focusing on the overhead minimization, and as high as 1024% with ARM and hardware minimization in mind.

TU Delft Delft University of Technology

Quantum & Computer Engineering

Dynamic Hardware Binary Translator for ρ-VEX

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Angelos E. Ntasios
born in Volos, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Dynamic Hardware Binary Translator for ρ-VEX

by Angelos E. Ntasios

Abstract

The last years, there has been a increasing trend in embedded system and FPGA implementations for greater flexibility and also, a rising adaptation of heterogeneous platforms. These platforms often include FPGAs and embedded cores side by side. ρ-VEX core, developed and maintained by the Computer Engineering group of TU Delft, is a VLIW processor mostly developed on FPGAs. On the other hand, popular embedded architectures include the established ARM architecture and the newly rising RISC-V architecture. In order for these architectures to communicate with the ρ-VEX core, a translation procedure has to be established. In this thesis, a hardware dynamic binary translator was designed, able to translate on-the-fly ARM and RISCV instructions to ρ-VEX instructions. The translator will enable heterogeneous platforms to be developed and also allow pre-compiled binaries for one (ARM/RISC-V) architecture to be directly ported in another one (ρ-VEX). This thesis provides a design process that focuses on two approaches: first is the minimization of the overhead resulting from the translation procedure, and the second is the minimization of the hardware alterations and/or hardware additions. These design choices were examined in the ρ-VEX core simulator. The simulations show that for overhead minimization, the resulting overhead can be as low as 1% for the RISC-V with focusing on the overhead minimization, and as high as 1024% with ARM and hardware minimization in mind.

Laboratory : Computer Engineering
Codenumber : CE-MS-2019-02

Committee Members :

Advisor: Prof. Dr. Ir. Stephan Wong, CE, TU Delft

Chairperson: Prof. Dr. Ir. Arjan Van Genderen, CE, TU Delft

Member: Prof. Dr. Ir. Przemyslaw Pawelczak, ES, TU Delft

Dedicated to my family and friends

Contents

List of Figures **xi**

List of Tables **xiv**

List of Acronyms **xv**

Acknowledgements **xvii**

1 Introduction **1**

2 Background **5**

2.1 ρ-VEX 5
2.2 Instruction Set Architectures and Binary Executable Files 6
2.3 Binary Translators 6
2.3.1 Static Binary Translators 7
2.3.2 Dynamic Binary translators 7
2.3.3 QEMU 8
2.3.4 Hardware-Accelerated Dynamic Binary Translation 9

3 RISC-V to ρ-VEX Binary Translation **11**

3.1 RISC-V Architecture 12
3.1.1 Architectural Highlights 12
3.2 Binary Translator High Level Architecture 14
3.3 Architectural Differences Between RISC-V and ρ-VEX 16
3.3.1 RISC-VLIW incompatibility 17
3.3.2 Endianess incompatibility 17
3.3.3 Link register incompatibility 18
3.3.4 Program Counter access incompatibility 19
3.3.5 Pipeline hazards 19
3.4 Instruction Set Translation 20
3.4.1 Register Operations 21
3.4.2 Immediate operations 21
3.4.3 Branch operations 23
3.4.4 Memory Load Operations 24
3.4.5 Memory Store Operations 25
3.4.6 Miscellaneous Instructions 25
3.5 Simulator Implementation and Benchmark Statistics 26
3.5.1 Simulator Modifications 26

4 ARM to ρ-VEX binary translator **29**
4.1 ARM architecture 30
4.1.1 Architectural Highlights 30
4.1.2 ARM registers 34
4.2 High-Level Description of the Translator 35
4.2.1 Decode unit 35
4.2.2 Translate unit 36
4.2.3 Subroutine ROM 36
4.3 Architectural Differences Between ARMv4 and ρ-VEX. 36
4.3.1 Pipeline hazards 37
4.3.2 Shifting mechanism 37
4.3.3 Direct access to PC 40
4.3.4 Conditional Execution 41
4.4 Overflow/Borrow detection 44
4.4.1 Overflow detection 44
4.4.2 Borrow detection 45
4.4.3 Carry detection 45
4.5 Second operand calculation 45
4.5.1 Immediate generation 46
4.5.2 Unmodified register 47
4.5.3 Logical left shift by immediate 48
4.5.4 Logical left shift by register 48
4.5.5 Logical right shift by immediate 50
4.5.6 Logical shift right by register 51
4.5.7 Arithmetic shift right by immediate 52
4.5.8 Arithmetic shift right by register 53
4.5.9 Rotate right by immediate 54
4.5.10 Rotate right by register 54
4.5.11 Rotate right by with extend 55
4.6 Memory Addressing Modes 56
4.6.1 Immediate offset 56
4.6.2 Register offset 57
4.6.3 Scaled register offset 58
4.6.4 Immediate pre-indexed 59
4.6.5 Register pre-indexed 60
4.6.6 Scaled register pre-indexed 60
4.6.7 Immediate post-indexed 62
4.6.8 Register post-indexed 62
4.6.9 Scaled register post-indexed 63
4.6.10 Multiple Load/Stores increment before/after 64
4.6.11 Multiple Load/Stores decrement before/after 65
4.7 Combining all the emulation techniques 66

5 Simulation results **71**
5.1 RISC-V execution metrics . . . 71
5.2 ARM execution metrics . . . 72
5.2.1 Shifting mechanism overhead . . . 72
5.2.2 PC write overhead . . . 74
5.2.3 Predication system overhead . . . 74
5.2.4 Flag comparison, translator overhead . . . 75
5.2.5 Flag update, emulation overhead. . . . 76
5.2.6 Flag update, translator implementation overhead . . . 76
5.2.7 Combination of all the translation techniques . . . 77

6 Future work and conclusions **79**
6.1 Conclusions . . . 79
6.2 Main contribution . . . 81
6.3 Future work . . . 81
6.3.1 Word Filling . . . 81
6.3.2 Instruction buffer . . . 82
6.3.3 Out-of-order execution . . . 83
6.3.4 Resource scheduler . . . 83

Bibliography **86**

A Appendix: RISC-V Instruction Translation **87**
A.1 add . . . 87
A.2 sub . . . 87
A.3 sll . . . 87
A.4 slt . . . 87
A.5 sltu . . . 88
A.6 xor . . . 88
A.7 srl . . . 88
A.8 sra . . . 88
A.9 or . . . 89
A.10 and . . . 89
A.11 addi . . . 89
A.12 slli . . . 89
A.13 slti . . . 89
A.14 sltiu . . . 90
A.15 xori . . . 90
A.16 srli . . . 90
A.17 srai . . . 90
A.18 ori . . . 91
A.19 andi . . . 91
A.20 beq . . . 91
A.21 bne . . . 91
A.22 blt . . . 92

A.23 bge . . . 92
A.24 bltu . . . 92
A.25 bgeu . . . 93
A.26 lb . . . 93
A.27 lh . . . 93
A.28 lw . . . 93
A.29 lbu . . . 93
A.30 lhu . . . 94
A.31 sb . . . 94
A.32 sh . . . 94
A.33 sw . . . 94
A.34 jal . . . 94
A.35 jalr . . . 95
A.36 lui . . . 95
A.37 auipc . . . 95

B Appendix: ARM Instruction Translation **97**
B.1 ALU instructions . . . 97
B.1.1 AND . . . 98
B.1.2 EOR . . . 98
B.1.3 SUB . . . 99
B.1.4 RSB . . . 99
B.1.5 ADD . . . 100
B.1.6 ADC . . . 101
B.1.7 SBC . . . 101
B.1.8 RSC . . . 102
B.1.9 TST . . . 103
B.1.10 TEQ . . . 103
B.1.11 CMP . . . 104
B.1.12 CMN . . . 105
B.1.13 ORR . . . 105
B.1.14 MOV . . . 106
B.1.15 BIC . . . 106
B.1.16 MVN . . . 107
B.2 Load/Store instruction . . . 107
B.2.1 STRB . . . 108
B.2.2 STRH . . . 108
B.2.3 STR . . . 109
B.2.4 LDRB . . . 109
B.2.5 LDRSB . . . 110
B.2.6 LDRH . . . 110
B.2.7 LDRSH . . . 111
B.2.8 LDR . . . 111
B.2.9 LDM . . . 112
B.2.10 STM . . . 112

B.3 Multiplication instructions . . . 113
B.3.1 MUL . . . 113
B.3.2 UMUL . . . 114
B.3.3 SMUL . . . 114
B.3.4 MLA . . . 115
B.3.5 UMLAL . . . 116
B.3.6 SMLAL . . . 117
B.4 Branch Instructions . . . 117
B.4.1 B . . . 118
B.4.2 BL . . . 118
B.5 Miscellaneous Instructions . . . 118
B.5.1 SWP . . . 118
B.5.2 SWPB . . . 119
B.5.3 SWI . . . 120

List of Figures

3.1 RISC-V base instruction formats. 14
3.2 High level design of the translator interface with ρ-VEX. 15
3.3 High level design of the translator. This is the least invasive design, meaning that it is the simplest design in terms of hardware complexity. . 16
3.4 Little to big and vice versa endianness converter. Left side shows the byte rearrangement of two half word bytes and right side shows the byte rearrangement of a full 4 byte word. 18
3.5 Logical interconnection of the modified segments of the simulator code. . 27

4.1 ARM architecture shifter. No individual shift instructions exist, instead the second operand can be shifted before the instruction execution. . . . 33
4.2 High level schematic of the ARM binary translator. The translator is within the second pipeline stage. Colored modules within the translator RTL are the optional additions and the light gray ones exist in all versions. 35
4.3 Flag register update emulation. Requires three cycles and eight instructions. 43
4.4 Multiple load store, increment vs decrement. 66

5.1 Comparison of the proposed shifting mechanics implementations. 74
5.2 Comparison of the proposed flag comparison techniques. 76
5.3 Comparison of the proposed flag update techniques. 78

6.1 Mixing instructions to fill NOPs generated by the translator. 82

List of Tables

3.1 RISC-V base instruction sets. All implementations must implement one of these ISAs. 12
3.2 RISC-V extension subsets. Some of the subsets might conflict with each other or with some base ISAs. 13
3.3 Calling conventions for integer registers. 14
3.4 Program counter access microcode instructions. Red PC values are invalid and the fetched instructions are not dispatched. Instead, the translator dispatches the PC access instructions. 20
3.5 Example of a Read After Write (RAW) hazard that can occur. Red instruction at stage 1 will receive a wrong value in Ra. 20
3.6 Inserting NOPs coupled with a backwards jump to avoid hazards. 20
3.7 RISC-V to ρ-VEX translation table. 22
3.8 Register operations and function fields decoding. 23
3.9 Integer operations and function fields decoding. 23
3.10 Branch operations, function fields decoding. 24
3.11 Emulation code to execute two instructions and jump back to normal program flow. Assuming a a RISC branch instruction has been fetched, this needs to be translated into two VEX instructions: compare and branch. 25
3.12 Memory load operations, function fields decoding. 25
3.13 Memory load operations, function fields decoding. 25
3.14 Memory store operations, function fields decoding. 26
3.15 Miscellaneous operations. 26

4.1 ARM predicate example. First column shows the time, second column shows the instruction, third column shows the instruction flags, fourth column shows whether the instruction will update the flag values or not and the final column are the processor flag values. 32
4.2 ARM processor states. 34
4.3 Shift instructions in powerstone benchmark 38
4.4 Second operand calculation program flow. The “ADD” instruction has a shifted second operand. First the shifted operand is calculated and then the main instruction is executed. 39
4.5 Complete instruction emulation example. The instruction to be executed is a simple Add with a shift incorporated and flag update. Red segment is the predication, blue segment is the second operand calculation, green part is the instruction execution and yellow shades show the calculation of the N, Z, C, V flags. The rest of the instructions update the flag register and resume normal program flow. 68

4.6 Total overhead resulting from emulating all the steps of executing the instructions. These include the predication calculation, the second operand calculation, the instruction calculation, the flag updating and the program flow resumption. 69

5.1 Execution metrics. First column shows the name of the testbench, second shows the original RISC-V instruction count, third column shows the ρ-VEX implementation instruction count and overhead with hardware minimization in mind. Fourth and final column shows the overhead of the provided implementation. 72
5.2 Shifting subroutine overhead 73
5.3 Register duplicate instruction overhead. 73
5.4 PC target instruction overhead. This table shows the overhead that results from instructions that target PC as destination register. 75
5.5 Flag comparison overhead. This table shows the overhead by emulating the flag comparison. 75
5.6 Flag updating overhead. This table shows the overhead by the emulation of updating the instruction flags. 77
5.7 Flag calculation overhead. This table shows the overhead resulting by calculating the flags locally in the translator. 77

List of Acronyms

RTOS Real Time Operating System

CPU Central Processing Unit

ALU Arithmetic Logic Unit

OS Operating System

RAW Read After Write

VLIW Very Large Instruction Word

ILP Instruction Level Parallelism

DLP Data Level Parallelism

RISC Reduced Instruction Set Architecture

RTL Register Transfer Level

Acknowledgements

Angelos E. Ntasios
Delft, The Netherlands
April 10, 2019

1 Introduction

The trend for larger parallelization in processors, has been increasing ever since the frequency speeds have reached their limits (mainly due to physical properties restraints) [1]. There are many ways that processors can exploit the parallelization of the executed programs and one of them is instruction level prallelism (ILP). Instruction level prallelization allows many independent instructions to be executed at the same time and on different data. This is in contrast to Data Level Parallelism (DLP) where the parallelization source is the data itself. There are many micro-architecture techniques that implement ILP, such as pipelining, superscalar execution, out-of-order execution, register renaming, speculative execution, VLIW, and others. The latter one is the micro-architecture technique deployed by ρ-VEX and a more thorough description is provided in the following chapter 2.

An important advantage of ρ-VEX VLIW architecure is the flexibility it offers in terms of resource utilization and division. More specifically, the ρ-VEX VLIW architecture can have adjustable instruction length, adjustable number of execution stages (which for example can include or not, a multiplier), and adjustable context execution, (i.e., programs can be executed in different resources at the same time). These programs can also be swapped with other ones on-the-fly, during gexecution.

An extra step to the flexibility and customizability of ρ-VEX would be to have the capability to execute binaries of other architectures. This could allow pre-compiled binaries in secondary architectures to execute directly to ρ-VEX without the need of re-compilation. In addition, another very effective way to make use of the above flexibility, is to run the ρ-VEX core alongside other popular processors as an accelerator. For example an ARM processor could directly communicate with ρ-VEX and drive compute- intensive or complex kernels to be executed by ρ-VEX. In order to allow direct communication a translator is required between the cores.

The main benefits of developing the binary translator are flexibility and efficiency. As an example, many FPGA boards nowadays incorporate embedded processors which work alongside FPGA cells. The ρ-VEX can co-exist next to another core and serve as an accelerator or a co-processor by executing code on-the-fly. Disregarding the aforementioned reasoning, the translation procedure can be performed (and in the vast majority of the cases is) by software translators. However, ρ-VEX has a feature that does not allow a software translation technique to match the performance produced by a hardware binary translator. This is due to the ρ-VEX variable instruction length which can change during run-time. This coupled with the VLIW nature of ρ-VEX, allows a hardware binary translator to generate variable length instructions based on the current configuration. Thus, the research question that arises is the following:

"How can we construct a dynamic binary translator that allows ρ-VEX to execute other ISAs on the fly?"

In order to answer the above question, a number of goals must be set and completed. These goals are the following:

- Provide a clear basis for a future hardware implementation.
- Provide a design space exploration for the future hardware implementation.
- Provide hardware design choices with different optimizations in mind.

In order to achieve the above goals an overview of the followed methodology is provided below:

- State of the art and literature review.
- Determine the most appropriate ISAs for translation.
- Analyze the architectures of the to-be-translated ISAs.
- Pinpoint the incompatibilities with ρ-VEX and provide micro-architectural solutions.
- Design an invasive and a non-invasive translation technique.
- Apply those techniques on the ρ-VEX simulator.
- Provide clear instructions and schematics for hardware implementation.
- Provide simulation results and measurements.

In the following chapters, the preemptive work of two hardware binary translators are presented. One is a RISC-V to ρ-VEX, and the other is ARMv4 to ρ-VEX. Both are implemented in the ρ-VEX simulator and provide a plan and description of the translator architecture, which will allow for a hardware implementation in the future.

A brief description of how the system works is as follows: the translators read the binary file generated by the secondary compiler (RISC-V or ARM) on one side, and generate ρ-VEX instructions on the other side, which emulate precisely the operation of the original binary file. The reason RISC-V was chosen was due to its simplicity, relatively small, instruction set, high compatibility with ρ-VEX and rising popularity. The RISC-V binary translator was implemented first. This would provide a relatively simple insight of how the more complex ARM translator should be implemented. RISC-V is a classic RISC architecture that has a large compatibility with the ρ-VEX instructions; large compatibility means that the instructions are mostly translated directly to a single ρ-VEX instruction. Furthermore RISC-V ISA translation will allow for further investigation of how well a simple RISC architecture can be executed and potentially accelerated by a VLIW processor.

The other architecture that was chosen was the ARM architecture mainly in embedded systems and platforms. This was chosen mainly due to its popularity since currently ARM is the most widely used architecture. However, due to its high complexity the

incompatibilities are significant. As a result two general approaches are examined: The first one, is complete instruction emulation, i.e., all instructions that do not translate to a specific ρ-VEX instruction are translated to multiple that emulate the behavior of the ARM instruction. The second approach was to incorporate some of the extra required functionality inside the translator, which has the effect of hiding many translation steps, reducing the translation overhead, but at the same time increasing hardware design complexity.

The thesis is organized as follows: Chapter 2 provides a brief overview of the ρ-VEX architecture as well as a brief introduction to binary translators. Chapter 3 and 4 present the exact methodology that was followed in order to develop the ARM and RISC-V binary translators. Next, Chapter 5 provides simulation metrics and results. Finally, Chapter 6 is devoted mainly to future work implementations and improvements as well as conclusions.

2 Background

This chapter contains background information for all the relative information of this thesis. It provides a brief explanation of the current status or the ρ-VEX project, as well as information about binary translators. In addition, basic concepts are explained such as binary executables and instruction set architectures. Furthermore, the current state-of-the-art in binary translators is presented.

2.1 ρ-VEX

ρ-VEX was developed within the TU Delft Computer Engineering group, and is written and maintained entirely by students. It is a VLIW, re-configurable and extensible processor that is implemented as a simulator, on FPGAs and even on an ASIC [2]. The main goals of the ρ-VEX core are to support dynamic workload execution as well as educational and research purposes.

The main focus of the ρ-VEX processor is flexibility. Flexibility in terms of performance, power consumption, parallelization, and code execution adaptation. ρ-VEX is a Very Large Instruction Word (VLIW) architecture that supports a variable length instruction and online context switching. More information on the ρ-VEX core and project can be found here [3]. Compared to a classic RISC architecture, a VLIW processor allows programs to explicitly specify instructions to execute in parallel, instead of sequentialy. Essentially, the compiler specifies multiple operations to be executed simultaneously instead of serially. These separate instructions are called syllables and all together an instruction.

Another important feature of ρ-VEX is re-configurability. The core, can adapt in real time to requirements and limitations on the contexts[1] and code it executes. Depending on the available Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP), the core can increase or decrease the size of the executed word and/or assign different number of resources to different threads. Furthermore, contexts can be swapped in and out of execution lanes depending on the priority and demands. Essentially, the core can at any point stop the execution of a context, store its current execution state, switch to a different one, and then later resume.

Lastly, another important feature of ρ-VEX is the fact that it is extensible. This means that new instructions can be designed and incorporated in the core as well as the accompanying toolchain. This can be particularly helpful in situations where an accelerator needs to be implemented side by side with ρ-VEX.

[1] A context can be thought as an instance of a thread that is executed. At any given time, at most four contexts can be executed.

2.2 Instruction Set Architectures and Binary Executable Files

The Instruction Set Architecture (ISA) is a list of mostly fixed instructions that define a computer architecture. Essentially, it is the list of operations that an implemented processor of this specific ISA can execute. An ISA defines everything from the list of instructions, to the required register list and the way the instructions are encoded. Furthermore, it serves as an interface between software and hardware. Any program written in a high level language is eventually translated into a list of these instructions, which are then translated into binary code that gets executed in the processor. A program written for a specific ISA can be executed by any processor that implements this ISA. For example any code written for the x86 ISA can run on either an Intel CPU or an AMD CPU since they both implement the x86 architecture. However, it cannot run on a processor that implements a different ISA.

Any program after compilation, results in a series of instructions that belong to a specific ISA. These instructions are next translated to a binary file that is executed by a processor. This file is usually called an executable-linkable format file (ELF) and contains all the binary code along with code for routine calls and memory initialization data. Essentially, this is the piece of binary information that goes into the processor's instruction memory and is read for execution. Since the work on this thesis is performed on the ρ-VEX simulator, this type of files are read as an input and interpreted by the simulator. The exact same input would also be in the hardware translation procedure.

2.3 Binary Translators

The x86 is an **backwards compatible** ISA developed by Intel in 1978. Most personal computers nowadays contain a processor that implements the x86 architecture. The fact that modern CPUs implement an ISA that was initially developed more than 4 decades ago, shows how firmly attached to legacy ISAs are modern software and hardware development. CPU developers do not opt for a new design due to development complexity, costs and mainly fear of losing market share due to software incompatibilities [4]. On the other hand, software developers do not want to develop complex tools nor spend time and resources to port legacy code to new architectures.

A good solution to the above problem is to utilize binary translators. Binary translators can operate with pre-compiled code targeted for a specific architecture and produce instructions for a different one. Generally, there are two types of binary translators: Static binary translators and dynamic binary translators [5].

As mentioned above, a program that is compiled for a specific ISA cannot run on a processor that implements a different ISA (at least without any modifications either on the code or the processor). This is where the translator comes and serves as an interface between the two different ISAs. Binary translators offer great possibilities in terms of flexibility and adaptation [4][6], since they allow for automatic binary code translation, which in turn, allows the code to be executed in a different architecture, without the need for recompilation. Furthermore, binary translators are a perfect fit for VLIW processors

[7] [8], [9] since the often resulting instruction overhead can be incorporated in a single VLIW within one cycle.

2.3.1 Static Binary Translators

Static binary translators operate offline and translate raw binary code from one ISA to another. Essentially, they receive a binary file as an input and after processing it, they produce another binary file which is translated to a different ISA. Static translators offer some key advantages compared to dynamic ones, namely: no run-time execution penalty, no memory and storage overhead. However, they suffer from two major drawbacks: code discovery and code location. When the translator parses data from the binary executable file, the translator cannot always tell if the parsed binary code is data or instructions; this is called code discovery problem. In some cases, the data might have the exact format as an instruction and the translator attempts to translate which results in an error. For example, x86 architecture often mixes data and instructions in the same address space.

Another problem arises with branches and PC-relative data. These kinds of data are known only at run-time and the translator cannot generate code that utilizes these values; this is called code location problem. Both of these cases are examined here, during the development of the ARM binary static translator [10]. Similar problems as the latter ones were also encountered in the current thesis work, however since this is a dynamic hardware translator implementation, the code location problem can be dealt with.

2.3.2 Dynamic Binary translators

Dynamic binary translators operate on-the-fly and translate instructions as the program is executed [11]. They can monitor the execution state and they can dispatch micro-codes that overcome many problems that are encountered in static translators. Generally, any translation of the binary code results in an overhead with a few exceptions [12]. As a result optimization in the code generation can play a significant role in the resulting overhead.

For example, suppose an instruction needs to generate a micro-code sequence in order to emulate an instruction. If the micro-code sequence generates a value during its execution, it could be the case that this value can be re-used in following instruction emulations. As a result a “smart” translator should be able to foresee this and store this value for future use. Especially in the case of ρ-VEX, optimization plays a significant role since the VLIW nature of the core offers great opportunities of parallel execution. Since translation often results in an overhead of instructions, if these instructions are not dependent on each other, they can be spread out in a single VLIW and executed within one cycle. Contrary to static binary translators, dynamic translators have performance restrictions since the translation operation is performed on-the-fly. As a result, the translation procedure should not hinder the overall performance of the system.

Most of the translator implementations are dynamic, since the major problems of static translation as discussed above can be overcome. Dynamic translators can be implemented with software, hardware or a combination of both. Some examples of software translators are QEMU [13], BOA [14], DAISY [15], CRUSOE [16], DYNAMO

[17], FX!32 [18] and UQDBT[19]. On the other hand, Simon Rokicki, et.al [20] present a hardware binary translator and Jiunn-Yeu Chen, et.al [10] a translator that combines hardware and software techniques.

2.3.3 QEMU

QEMU is a software binary translator (or emulator), which was used extensively during the development process of this thesis. QEMU can emulate several ISAs [21], namely:

- IA-32 (x86).
- x86-64.
- MIPS64 Release 6[5] and earlier variants.
- Sun's SPARC sun4m and sun4u.
- ARM.
- PowerPC.
- ETRAX CRIS.
- SMicroBlaze.
- RISC-V.

QEMU's main operational usage is to run one OS on another, such as Windows on Linux. It can also be used for debugging purposes since the execution can be paused at any time in order to examine the program flow. QEMU consists of the following subsystems:

- CPU emulator.
- Emulated devices (e.g., VGA display, mouse and keyboard).
- Generic devices used to connect the emulated devices to the corresponding host devices.
- Machine descriptions instantiating the emulated devices
- Debugger.
- User interface.

During this thesis QEMU was used for debugging purposes for the ARM translator. In order to debug the execution flow of the binary translator, QEMU was used to mirror the execution and compare the normal program flow to the equivalent program flow of the ARM-to-ρ-VEX binary translator. A major difficulty in this process was the fact that QEMU debugger does not display instructions of re-executed kernels. For example, if a loop runs 10 times, QEMU will display the code of only the first run. Furthermore, if the kernel is called multiple times by different segments of the code, again QEMU does not display the re-executed instructions.

2.3.4 Hardware-Accelerated Dynamic Binary Translation

In this paper [20] the authors present a hardware binary translator which translates MIPS [22] to a native VLIW core loosely based on VEX [23], as also ρ-VEX is. Their technique is based on a three stage parsing sequence. The to-be-translated code is parsed in three stages each of which extracts optimization information. The first stage performs basic instruction decoding, data extraction, register values and most importantly, translation to native instructions. No optimization is performed in this stage and the translated code is stored in a temporary memory location. The content of this memory, will be used in the next step to also identify block boundaries.

The second step is to re-order the instructions in the block generated in the previous step. This step takes into account instruction level dependencies by keeping track the register file reading/writing sequence. Finally, in the third step, the VLIW scheduler takes the re-ordered instructions generated by the second step, and by implementing a greedy algorithm tries to fill the VLIW.

3 RISC-V to ρ-VEX Binary Translation

This chapter describes the procedure of translating RISC-V targeted binary code to ρ-VEX instructions. The translation is performed in the ρ-VEX simulator, and the goal of this chapter is to establish the appropriate foundations required for a hardware implementation. Out of the two ISAs that were translated in this thesis, RISC V is simpler and easier to implement. This is because the initial design choice for the RISC-V ISA, was to be highly flexible, extensible, and architecturally non-complex. Naturally, the work on RISC-V is presented first. Furthermore, it has many architectural similarities and very few incompatibilities when compared to ρ-VEX, which renders the development procedure much smoother.

Initially, an overview of the RISC-V ISA is presented including all the extensions, the most compatible of which was chosen. The reason behind this choice is that it allows for a simpler and more efficient hardware translation and also an easier testing. Also a complete analysis of the translation procedure is presented, as well as RTL schematics.

The reader, after reading this chapter should expect the following:

- Section 3.1: brief overview of the RISC-V architecture.
- Section 3.2: high-level architecture of the translator.
- Section 4.3: architectural comparison between RISC-V and ρ-VEX.
- Section 3.4: ISA translation precedures.
- Section 3.5 simulator implementation and benchmark statistics.

3.1 RISC-V Architecture

RISC-V is an open-source instruction set architecture which was developed by Berkeley University of California. It is a Reduced Instruction Set Architecture (RISC) and its initial purpose was educational. The simplicity and relatively small number of instructions, makes it suitable for embedded systems applications and small devices. Being open-source, naturally, it can be freely used by anyone for any CPU design and software.

RISC-V is based on DLX which was developed in 1990 for educational purposes as well and was initially presented in the book "Computer Architecture: A Quantitative Approach". David Patterson who was a co-author helped with the DLX development and later on with RISC-V. According to the designers, RISC-V can be more efficient regarding area, power and speed, when compared to similar commercial CPUs. In addition, even though most academic ISAs are intended for educational purposes, RISC-V is also intended for commercial ones. Currently, there are companies that use RISC-V in their products like SiFive, Codasip and UltraSoC, while others are already working on the ISA or are on development [24]. For example Nvidia plans to replace the Falcon processor on the GeForce products with a RISC-V processor.

Usually, developing a CPU ISA is a difficult task which requires people with expertise in many fields, such us compilers, CPU micro-architecture, and electronics. As a result this kind of expertise is found in large groups of engineers, usually within companies. For this reason the RISC-V ISA is considered a community project, with people from the open-source community contributing.

3.1.1 Architectural Highlights

RISC-V was developed with simplicity in mind. One of the main goals is to avoid complex architectural techniques such as micro-coding and out-of-order execution. RISC-V is a modular Reduced Instruction Set Computing (RISC) instruction set. The modularity enables the choice of instruction subsets depending on the intended usage. The base small integer ISA is standard for all implementations and is used alongside every extension.

The ISA supports 32, 64, and 128-bit length words. There are slight variations in the subsets depending on the word length. The subsets are intended for various usage environments, including general purpose computers, embedded systems, vector processing and supercomputers. The ISA has reserved bits for all the 128-bit future implementations.

Table 3.1: RISC-V base instruction sets. All implementations must implement one of these ISAs.

Base	Description
RV32I	Base Integer Instruction Set 32-bit
RV32E	Base Integer Instruction Set Embedded 32-bit
RV64I	Base Integer Instruction Set 64-bit
RV128I	Base Integer Instruction Set 128-bit

All RISC-V implementations must implement one of the base integer instruction

subsets seen in Table 3.1. The base ISAs are a set of chosen instructions, that allow for a reasonable target for compilers, linkers, assemblers and operating systems. As a result it serves as the required frame, which can be used standalone or with other instruction set expansions. The base instruction sets cannot be altered and should not conflict with each other. What this means is that only one base integer should be used.

Table 3.2: RISC-V extension subsets. Some of the subsets might conflict with each other or with some base ISAs.

Extension	Description
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
Q	Standard Extension for Quad-Precision Floating-Point
L	Standard Extension for Decimal Floating-Point
C	Standard Extension for Compressed Instructions
B	Standard Extension for Bit Manipulation
J	Standard Extension for Dynamically Translated Languages
T	Standard Extension for Transactional Memory
P	Standard Extension for Packed-SIMD Instructions
V	Standard Extension for Vector Operations
N	Standard Extension for User-Level Interrupts

The base integer ISA can be extended with other non-standard subsets. These subsets can be seen in Table 3.2. The base ISA is named “I” for integer, so for example the 32 bit base ISA is the RV32I. In order to support more general purpose software development, integer multiplication/division was added (M extension), as well as floating point calculations (F, D, Q extensions). Furthermore the atomic instructions (A extension), add inter-processor synchronization capabilities, and individual memory instructions. During the time this thesis was written, (2017) only the above extensions were implemented.

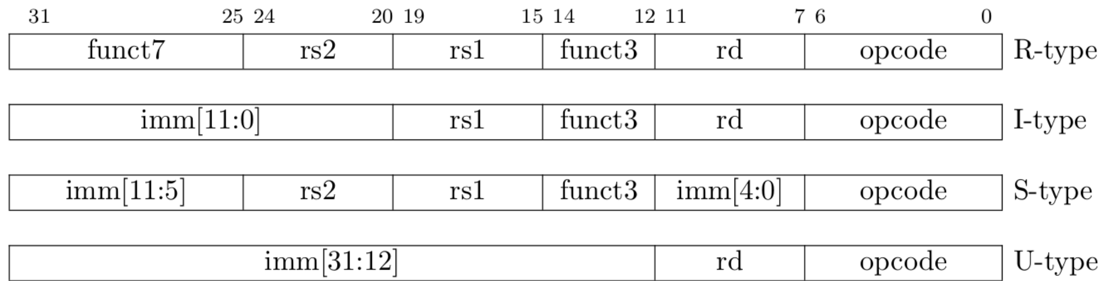
The base ISA has fixed 32-bit length instructions. These instructions must be aligned to 32-bit memory boundaries. However, larger instructions can be used in order to facilitate different needs. For example a 16-bit compressed version exists as well as larger than 32-bit sized instructions. In this work only the 32-bit sized instructions are used. Figure 3.1 shows the instruction format that is used for all the base 32-bit RISC-V instructions. The instruction formats can be categorized into four arrangements: R-type, I-type, S-type and U-type. All the instructions are formatted into one of these four categories. As can be noticed in all variants, the register addresses “rs1”, “rs2”, “rd” (when they are present) are encoded in the same positions. This is really helpful and efficient in the translation procedure, since it can be realized with a simple wire rerouting. Contrary to that, immediate values require multiplexers since the location of the immediate is not the same in all variants. Furthermore, in all variants with immediate values the sign bit is always at position 31. This can also be used to speed

Table 3.3: Calling conventions for integer registers.

Register	ABI name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporaries	Caller
x8	s0	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

up sign extension circuitry.

Figure 3.1: RISC-V base instruction formats.



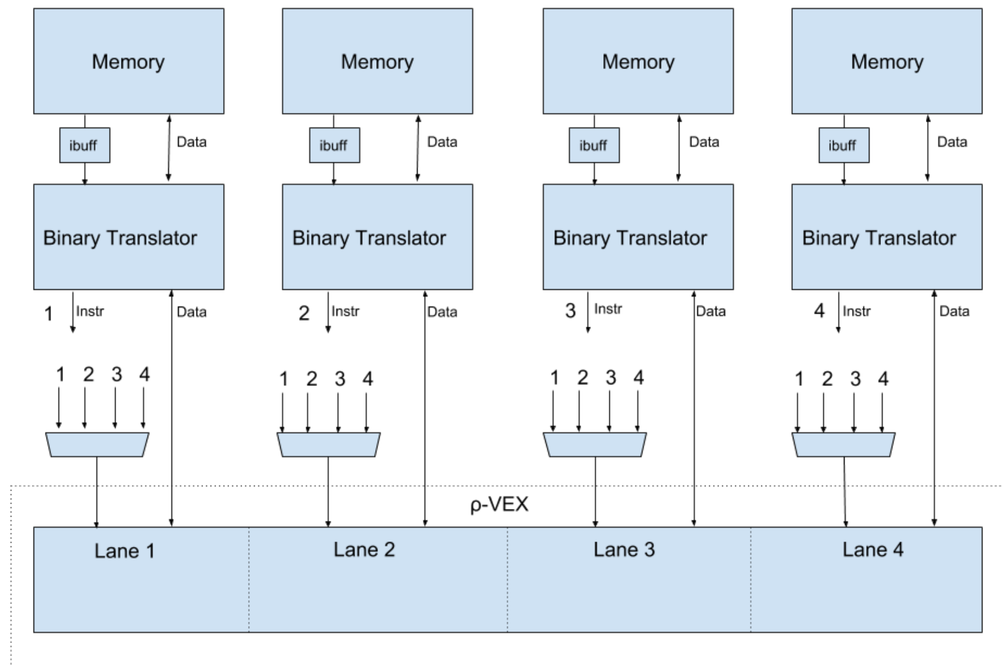
There are 33 user visible 32-bit registers in the RISC-V architecture, 31 general purpose (rs1-rs31), one that is hardwired to zero (rs0) and the PC which is the program counter. There is no dedicated link register, however the standard calling convention uses register rs1 as the register to hold the return address. Since ρ-VEX has 64 registers, the first 32 are utilized by the translator. In addition to the integer registers, there are also floating point registers. Table 3.3 shows the calling convention for all the registers. Since only the base integer instruction set is used for this work, the floating point calling convention is omitted. The reader is referred to the "RISC-V Instruction set manual" in the appendix for further information on floating point calling convention.

3.2 Binary Translator High Level Architecture

The main goal of this work is to provide a solid description of the architecture of the translator that will be implemented in actual HDL. Since the simulator operates only on one context the development and verification was done only with one context in mind.

However, the hardware implementation could be implemented with all 4 lane groups containing a binary translator module. This would allow different lanes to execute different architectures at the same time. The translator is located between the ρ-VEX core and the memory. Many design choices will be presented later which include invasive and non-invasive designs. The invasive designs alter the core design by adding modules or guiding signals such as the PC and register values to the translator. These designs might affect the core performance, area, and design complexity. The non-invasive designs utilize existing ρ-VEX instructions to emulate RISC-V instructions. This makes the communication of the core to the memory system almost completely transparent, however in many cases, it introduces a great instruction overhead. Figure 3.2 shows the high level interface of the binary translator with the core, assuming that all lanes contain a binary translator. This figure also assumes the least invasive designs where the only inputs and outputs of the translator are the instructions and the memory data. Each translator can potentially feed instructions to all lanes if scheduling techniques are utilized such us here [20]. Nevertheless, in this work only one lane is utilized with one translator.

Figure 3.2: High level design of the translator interface with ρ-VEX.

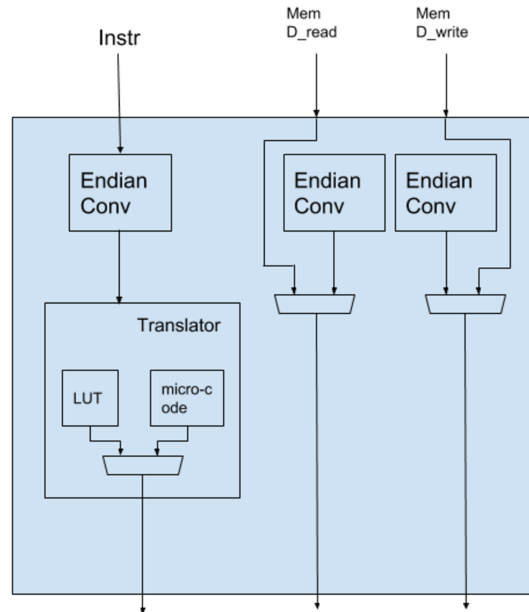


The translator design is depicted in Figure 3.3. This design is the least invasive out of a few that will be proposed in the following sections. This means that this is the simplest in terms of hardware complexity and requirements. The instruction input is the main input which is by default present[1]. The translator provides up to two instructions as

[1]It needs to be stated here that the only reason the memory data inputs are required is due to

output. The source of the output is either a lookup table in the case that the translation is 1 to 1 or a micro-code module that generates the appropriate instructions in order to emulate the RISC-V instructions with the appropriate ρ-VEX ones. All of these modules and multiplexers are controlled and monitored by a control unit.

Figure 3.3: High level design of the translator. This is the least invasive design, meaning that it is the simplest design in terms of hardware complexity.



3.3 Architectural Differences Between RISC-V and ρ-VEX

4.3 The first step to creating the binary translator is to pinpoint all the major architectural differences. Right from the start the most significant difference is the architecture type. While ρ-VEX is a VLIW architecture, RISC-V is a RISC architecture. Between them, ρ-VEX is clearly more complex since RISC-V was purposely designed with simplicity in mind. As a result RISC-V base ISA consists of nearly 4 times less instructions than ρ-VEX (RISC-V 38, ρ-VEX 169). Naturally, almost all of the RISC-V instructions can be translated directly to ρ-VEX with the exception of a few as will be presented

endianess incompatibility. These inputs can also be omitted by doing bit and reordering operations on all the words that are stored or loaded from memory.

with details in Section 3.4. Nevertheless, after translation, an overhead in instruction raw count will occur, which however, could potentially be diminished by exploiting the VLIW nature of ρ-VEX. A step even further in this direction is to actually reduce the cycles required to execute the same code. This can be achieved by filling the VLIW word with multiple independent RISC-V instructions. Further discussion on this matter is presented in the future work chapter 6. In addition several other incompatibilities exist such as endianness, instruction decoding, branching, program counter access and pipeline hazards all of which are analyzed below.

3.3.1 RISC-VLIW incompatibility

RISC-V is a classic RISC architecture, where the processor receives the instructions serially one at a time. As a result any optimization or parallelism in the execution pattern of the code has to be implemented by the hardware. Contrary to that, ρ-VEX, a VLIW processor, operates in a different manner. The core receives more than one instruction per cycle, in the form of a VLIW instruction; these VLIW instructions are issued by the compiler. This means that the compiler takes care of any paralelization and dependency checking. For the ρ-VEX compiler, the number of instructions embedded within a single VLIW instruction can be anywhere between 1 and 8. The ideal scenario would be to always dispatch 8 instructions per cycle, however this is not always possible. Contrary to that, the code generated by the RISC-V compiler is not optimized for a VLIW architecture. Even though the designers of the RISC-V architecture propose some VLIW functionality [24], as of 2018 none of these is yet implemented. Consequently, in order to fill the VLIW bundle with RISC-V instructions, extra hardware is required for dependency checking. The approach chosen in the current implementation is to place only one instruction per VLIW word with a few exceptions which will be discussed later. Even though this is not the optimum in terms of speed and performance, it is the first step towards a more optimized VLIW utilization; as a result, examination on this simple implementation can be performed in order to determine more optimized procedures. However, due to time restrictions this was not performed in this thesis.

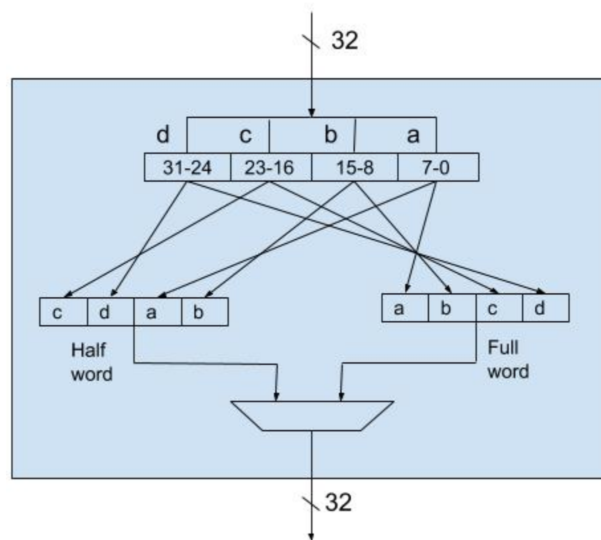
3.3.2 Endianess incompatibility

RISC-V has a little endian memory system while ρ-VEX has a big endian. The binary file generated by the RISC-V compiler contains both instructions and data in a little endian format. Since ρ-VEX is big endian the data transfered from the memory by the core are incomprehensible since the byte order is reversed. As a result any time an instruction is fetched, or data is read from the memory an endianess conversion must always occur. Furthermore, when data are written back to the memory, they should also be converted. This seems redundant at first (converting both when reading and writing), however data already generated by the compiler are in little endian which are later converted to big by the ρ-VEX memory system. The same procedure is followed when data are written back to memory (the ρ-VEX memory system converts them), as a result they need to be reversed again before they are written.

The solution to this problem is to reorder the bytes whenever memory is read or written. Endianess reordering occurs in two places in the translator; firstly, for all

instructions read from the instruction memory, and secondly when data is communicated with the data memory. Instruction word reordering is rather simple since the instructions are always 32 bits and are fed directly to the translator. As a result all instructions pass through the endianess module. The actual reordering hardware implementation is only a simple wire reordering as can be seen on schematic 3.4. However when data is communicated with the memory, the endianess reordering is a bit more complicated. Loaded data pass through the converter and depending on the size (halfword, word) the appropriate conversion is performed.

Figure 3.4: Little to big and vice versa endianness converter. Left side shows the byte rearrangement of two half word bytes and right side shows the byte rearrangement of a full 4 byte word.



3.3.3 Link register incompatibility

RISC-V architecture does not have a dedicated link register, however software calling convention as seen in Figure 3.3 sets register rs1(x1) as the link register. Furthermore register rs5(x5) can be used as an alternative link register when compressed RISC-V instructions are used ([2]. ρ-VEX on the other hand has a dedicated link register, which however can be mapped to a normal general-purpose register (rs63) on design time. Since the RISC-V link register is a general-purpose register, this means that it can be accessed for any operation. As a result the general-purpose link register mapping should also be activated in the ρ-VEX core.

[2] Refer to the RISC-V instruction manual [25], page 16 for more information

3.3.4 Program Counter access incompatibility

Another register that is also visible to the user in RISC-V architecture is the program counter. There are 3 RISC-V instructions that needs to read the value of PC: *auipc* as well as *jal* and *jalr* depending on the implementation, however currently there is no instruction to access the PC in ρ-VEX. The only way to access the PC, is to use the branch instructions which store the value of the next PC in the link register. Two solutions are provided for this problem: One is to use the existing ρ-VEX instructions in order to access the PC, and the other one is to make small hardware modifications.

The first solution is to generate a number of instructions that provide access to the PC, while at the same time maintaining normal program flow and data integrity. Table 3.4 shows these instructions. Suppose that in step 0 a RISC-V instruction requires access to the current PC. The first step is to copy the current link address value to a random register (rs32-rs62). Note that step 1 executes at the same time that the instruction is read, this means that the PC has not incremented yet. The link register copy can be performed with an “or” instruction. As mentioned earlier RISC-V utilizes only 32 registers, as a result, any of the remaining registers between 32 and 62 can be used for temporary storage.

After that in step 2 a random branch with “call” instruction is performed. This will allow us to store PC+1 into the link register. During this procedure, all of the fetched instructions are ignored by the translator until the normal program flow is resumed. As a result even though the PC increments normally and instructions are fetched, no new instructions are executed. In the next step a ‘1’ is subtracted by PC+1 in order to acquire the PC. Furthermore, the instruction that requires PC is executed since the value of the PC is now available in a general purpose register.

In the next step an unconditional jump is performed to PC and finally the link address is copied back to the link register. Note that this final step is needed because the program counter keeps on incrementing during the above steps. If for example a pause signal is available for the PC this final step can be omitted. This solution is a simple micro-code bundle that can be stored and get called for every instruction that requires access to PC. The problem with this implementation is the overhead in cycles as well as instructions. All of these instructions are dependent, as a result, they cannot be parallelized in a single VLIW. This means that for every instruction that requires access to PC there will be an overhead of 7 cycles and instructions.

The other solution is to slightly modify the ρ-VEX core datapath and drive the current PC value directly to the translator. In the provided implementation this approach was chosen. Both solutions have advantages and disadvantages: On one hand, the first solution introduces an overhead of 7 instructions but has no core modifications, and on the other hand, the second one requires modifications on the core but it does not introduce any overhead.

3.3.5 Pipeline hazards

The ρ-VEX core utilizes a pipeline in its architecture. The RISC-V compiler however is not aware of this, as a result the code generated is not optimized in any way to avoid pipeline hazards that can occur. The translator should handle the potential hazards

Table 3.4: Program counter access microcode instructions. Red PC values are invalid and the fetched instructions are not dispatched. Instead, the translator dispatches the PC access instructions.

Step #	Current PC	ρ-VEX Instruction	Link Register	RS50
0	PC	auipc	Original link address	X
1	PC	or	Original link address	X
2	PC+1	call	Original link address	Original link address
3	PC+2	sub	PC+1	Original link address
4	PC+3	Execute RISC-V instruction	PC	Original link address
5	PC+4	igoto	PC	Original link address
6	PC	or	PC	Original link address
7	PC+1	Resume normal execution	Original link address	X

Table 3.5: Example of a Read After Write (RAW) hazard that can occur. Red instruction at stage 1 will receive a wrong value in Ra.

stage	T1	T2	T3
1	add Ra, Ra, Rb(PC)	X(PC+1)	add Ra, Ra, Rb(PC+2)
2	X	add Ra, Ra, Rb	X
3	X	X	add Ra, Ra, Rb
4	X	X	X
5	X	X	X
6	X	X	X

by issuing NOPs coupled with backward jumps to the PC with the instruction that generated the hazard. Table 3.5 shows the potential hazard and Table 3.6 shows the correct program flow that results from NOP insertion.

3.4 Instruction Set Translation

As mentioned above the base integer instruction set for RISC-V consists of 38 instructions (excluding the FENCE instructions and assuming the handling of all system calls by a single instruction). Some of those instructions, have a direct equivalent in the ρ-VEX architecture and some do not. The ones that do not have an exact translation, either require extra hardware or need more than one ρ-VEX instructions in order to be implemented. The extra instructions required in the translation process is referred to as instruction overhead. In other words, a RISC-V instruction that is directly translated to

Table 3.6: Inserting NOPs coupled with a backwards jump to avoid hazards.

stage	T1	T2	T3	T4	T5	T6	T7
1	add Ra, Ra, Rb(PC)	X(PC+1)	add Ra, Ra, Rb(PC+2) dispatches a NOP instead	NOP(PC+3)	X(PC+1)	goto -2	add Ra, Ra, Rb(PC+2)
2	X	add Ra, Ra, Rb	X	NOP	NOP	X	goto -2
3	X	X	add Ra, Ra, Rb	X	NOP	NOP	X
4	X	X	X	add Ra, Ra, Rb	X	NOP	NOP
5	X	X	X	X	add Ra, Ra, Rb	X	NOP
6	X	X	X	X	X	add Ra, Ra, Rb	X

a ρ-VEX one, has zero overhead while one that requires 2 instructions has an overhead of 1. Out of the 38 RISC-V instructions, 29 can be directly translated, 6 can be either directly translated or with an overhead of 6 or 1 (depending on the implementation). One can be either directly translated or with an overhead of 5; and two are translated into two instructions which however are not dependent, thus allowing them to be executed simultaneously into two slots in the VLIW. Table 3.7 shows the translation table between RISC-V and ρ-VEX. The table shows all the RISC-V instructions of the integer base set, that are implemented, as well as their translation to ρ-VEX. Furthermore the third column shows a brief description of the instruction and the fourth shows the overhead that results from the translation.

The decoding process starts with the RISC-V instruction fetch from the memory. After the instruction passes through the endianess unit, it is ready for decoding. The first part that is extracted for all operations is the opcode. The opcode field always resides in bits 6-0 and it determines the generic type of the operation which can be one of the following:

- OP: Register operations
- OP-IMM: Immediate operations
- BRANCH: Branch operations
- LOAD: Memory load operations
- STORE: Memory store operations
- MISC operations

A clear description of all the instructions can be found in Appendix A.

3.4.1 Register Operations

These instructions use the R-type instruction format as seen in Figure 3.1. The fields of the encoding are six. "rs1", "rs2" and "rd" are the two source and destination registers respectively. "opcode" determines the generic type of the operation which in this case is operations between registers. Furthermore, there are two more fields, funct3 (bits 14-12) and funct7 (bits 31-25) which determine the specific instruction within the generic operation group according to Table 3.8. For example, assuming the opcode is the following: "0000000 00011 00100 000 00101 0110011". The first field examined is the opcode (bits 6-0) = 0110011, this means that the instruction contains a register-register operation. Next we examine funct7(31-25) and funct3 (14-12) which are "0000000" and "000" respectively. This means that the operation is add with rd = rs1 + rs2 (rd = 00101, rs1 = 00100, rs2 = 00011).

3.4.2 Immediate operations

These instructions perform operations between a source register, rs1, and an immediate value and store the result in rd. These instructions use the I-type format as seen in Figure

Table 3.7: RISC-V to ρ-VEX translation table.

RISC-V	**ρ-VEX**	**Description**	**Instruction Overhead**
add	add	RS1 + RS2	0
sub	sub	RS1 - RS2	0
sll	shl	RS1 << RS2 logical shift	0

slt	cmplt	if(signed(RS1) < signed(RS2))
		{RD = 1}
		else
		{RD=0}
			0

sltu	cmpltu	if(unsigned(RS1) < (unsigned(RS2))
		{RD = 1}
		else
		{RD=0}
			0

xor	xor	RS1 xor RS2	0
srl	shru	RS1 >> RS2 logical shift	0
sra	shr	RS1 >> RS2 arithmetic shift	0
or	or	RS1 or RS2	0
and	and	RS1 and RS2	0
addi	add	RS1 + sign_ext(IMM)	0
slli	shl	RS1 << IMM	0

slti	cmplt	if(signed(RS1) < sign_ext(IMM))
		{RD = 1}
		else
		{RD=0}
			0

sltiu	cmpltu	if(signed(RS1) < IMM)
		{RD = 1}
		else
		{RD=0}
			0

xori	xor	RS1 xor sign_ext(IMM)	0
srli	shru	RS1 >> IMM logical shift	0
srai	shr	RS1 >> IMM arithmetic shift	0
ori	or	RS1 or sign_ext(IMM)	0
andi	and	RS1 and sign_ext(IMM)	0
beq	cmpeq, br	if (RS1 == RS2) {branch}	0/1/7
bne	cmpne, br	if (RS1 != RS2) {branch}	0/1/7
blt	cmplt, br	if (signed(RS1) < signed(RS2)) {branch}	0/1/7
bge	cmpgt, br	if (signed(RS1) < signed(RS2)) {branch}	0/1/7
bltu	cmpltu, br	if (unsigned(RS1) < unsigned(RS2)) {branch}	0/1/7
bgeu	cmpgtu, br	if (unsigned(RS1 > unsigned(RS2)) {branch}	0/1/7
jal	or, goto	'0' or (PC + 4), jump (PC + sign_ext(IMM))	0
jalr	or, goto	'0' or (PC + 4), jump (sign_ext(IMM) + RS1)	0
lui	or	RD <== IMM	0
auipc	or	RD <== PC + IMM	0/5
lb	ldb	Load sign_ext(byte)	0
lh	ldh	Load sign_ext(halfword)	0
lw	ldw	Load word	0
lbu	ldbu	Load byte	0
lhu	ldhu	Load halfword	0
sb	stb	Store byte	0
sh	sth	Store halfword	0
sw	stw	Store word	0
ecall	trap	trap	0

Table 3.8: Register operations and function fields decoding.

operation	funct7 31-25	funct3 14-12
add	0000000	000
sub	0100000	000
sll	0000000	001
slt	0000000	010
sltu	0000000	011
xor	0000000	100
srl	0000000	101
sra	0100000	101
or	0000000	110
and	0000000	111

Table 3.9: Integer operations and function fields decoding.

operation	funct3 14-12
add	000
sub	000
sll	001
slt	010
sltu	011
xor	100
srl	101
sra	101
or	110
and	111

3.1. All the integer operations have the opcode field = "0010011" and the function fields can be seen on Table 3.9.

3.4.3 Branch operations

These instructions perform a comparison between rs1 and rs2 and depending on the result a branch is executed. They are of the SB-Type format as seen in Figure 3.1. The opcode field is equal to "1100011". The branch target is derived by adding the signed immediate value to the current PC and can generate jumps in the range of ±4KB. The offset value is formulated by concatenating the values located in the following segments: **Target = Instr(31) & Instr(7) & Instr(30 downto 25) & Instr(11 downto 8)**. Since the target is always a multiple of two the last bit is always zero and it is not stored, however the above value needs to be shifted left by 1. This produces the final 12-bit signed jump value which when added to the current PC produces the branch target. The branch instruction encoding can be seen in Table 3.10.

These instructions do not have a direct equivalent to ρ-VEX. The branch instructions in ρ-VEX perform the operation in two stages, first by storing the comparison result to

Table 3.10: Branch operations, function fields decoding.

operation	funct3 14-12
beq	000
bne	001
blt	100
bge	101
bltu	110
bgeu	111

branch registers, and then with another instruction, based on the value of the branch register, a branch is performed or not. This means that RISC-V branch instructions combine two ρ-VEX instructions in one. As a result three solutions are proposed for this problem one of which was used in this thesis. The three solutions can be seen below:

- The first solution is to utilize only the existing ρ-VEX instructions to translate the RISC-V instructions. Assuming that the PC cannot be stalled the core needs to execute two instruction separately and in different cycles, first the comparison and then the branch. While these two instructions are executed the PC increments which means that one fetched instruction will be invalid and will not be executed. As a result after the comparison and before the branch a jump is performed back to the initial PC value. This procedure is explained in Section 3.3.4. Essentially after the compare instruction is issued, a jump back to PC is performed and then the branch is executed normally.

- The second solution is to simply incorporate a comparison circuit inside the translator. The advantages of this approach is the zero instruction overhead and the fact that no modifications are required to the core. However, the major disadvantage of this approach is the area overhead and gate delay overhead which can potentially result in a longer critical path.

- The third solution, which is the one chosen for implementation in the simulator, is to perform the branch in two separate cpu cycles while at the same time stalling the PC. In the first cycle the comparison is performed and the result is stored in the branch register, while in the second cycle the branch is performed. This requires for the PC to be paused for one cycle so that normal program flow is maintained in case the branch is not taken. This approach is somewhat in between the two previous approaches as it has an instruction overhead of only 1, little to no area overhead and requires a simple pause signal for the PC. Below follows the branch instructions with the assumption that the third option is implemented.

3.4.4 Memory Load Operations

These instructions perform memory loads. The opcode field is equal to “0000011” and the function fields can be seen at Table 3.12. When data is communicated with the memory the endianess must change as mentioned in Section 3.3.2.

Table 3.11: Emulation code to execute two instructions and jump back to normal program flow. Assuming a a RISC branch instruction has been fetched, this needs to be translated into two VEX instructions: compare and branch.

Step #	Current PC	ρ-VEX Instruction
1	PC	cmplt
2	PC+1	NOP
3	PC+2	NOP
4	PC+3	goto
5	PC	br
6	X	resume normal execution

Table 3.12: Memory load operations, function fields decoding.

operation	funct3 14-12
lb	000
lh	001
lw	010
lbu	100
lhu	101

3.4.5 Memory Store Operations

These instructions perform memory stores. The opcode field is "0100011" and the function fields can be seen in Table 3.14. When the stored data is smaller than a full word (halfword/ byte) then the data are extracted from the lower bits of the registers. Furthermore, when data is communicated with the memory the endianess must change as mentioned in Subsection 3.3.2.

3.4.6 Miscellaneous Instructions

Besides the groups of instructions presented above, there are also some instructions that are distinct in their functionality. They are distinguished by the "opcode" field and they can be seen in Table 3.15. They are control transfer instructions and system calls. "jal" uses the J-type encoding, "jalr" uses the I-type encoding, "lui" and "auipc" uses the U-type and "ecall" which is distinguished by the "opcode" field value only.

Table 3.13: Memory load operations, function fields decoding.

operation	funct3 14-12
lb	000
lh	001
lw	010
lbu	100
lhu	101

Table 3.14: Memory store operations, function fields decoding.

operation	funct3 14-12
sb	000
sh	001
sw	010

Table 3.15: Miscellaneous operations.

operation	opcode
jal	110111
jalr	1100111
lui	0110111
auipc	101
exall	110

3.5 Simulator Implementation and Benchmark Statistics

As mentioned earlier this work is meant to provide the basis for further development and implementation in hardware. Everything mentioned above was implemented in the simulator which emulates the ρ-VEX operation with one context in scope. Proper functionality was verified with the "Powerstone" testbench programs that provide a proper verification output. Nevertheless, this does not mean that the provided version is 100% bug free. Below the following are presented: an overview of the simulator modifications, execution metric results, development methodology and code metrics.

3.5.1 Simulator Modifications

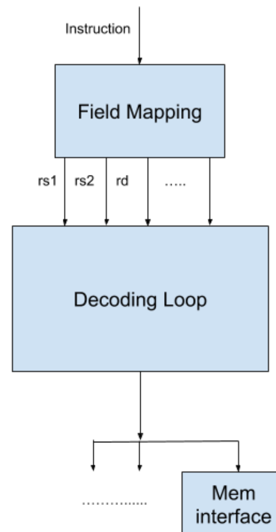
In the simulator only three areas are modified:

- Memory interface.
- Instruction decoding loop.
- Instruction field remapping.

The above simulator fields can be mapped to areas in the hardware implementation which are the data memory interface and the instruction buffer. However, it is hard to say with accuracy what and how should be altered precisely since this would require further work upon the actual VHDL hardware implementation. Figure 3.5 provides an overview of the three entities that were altered in the simulator and how they are logically connected.

Regarding the three simulator modifications, the simpler one is the memory interface where the only thing that was changed is the addition of an endianess function to the data store and load instructions. Most of the modifications were performed on the decoding loop where all the logic of the new instruction routing is performed. Finally, the mapping of the instruction fields is also completely changed.

Figure 3.5: Logical interconnection of the modified segments of the simulator code.



4 ARM to ρ-VEX binary translator

This chapter presents the methodology of developing the ARM to ρ-VEX binary translator. The ultimate goal is to create a ρ-VEX core that can seamlessly operate and execute different architectures at the same time, as a result a second ISA is chosen for binary translation also. Ideally different contexts with potentially different architectures will be executed in the ρ-VEX core.

The second ISA chosen is the ARM architecture. ARM was chosen mainly for its popularity; currently, it is one of the most popular processor architectures in the market. Compared to the previous implementation presented in Chapter 3, ARM is much more complex and even less compatible with ρ-VEX. This renders any conventional translator (e.g., a simple software translator) inefficient as the instruction overhead that is generated by using only ρ-VEX instructions is quite substantial as will be presented in the following sections. On the other hand, the VLIW nature of ρ-VEX, allows for more efficient designs since parallelization can be issued by the translator directly in hardware. All of the aforementioned topics will be presented in the following sections in detail along with other development procedures; More specifically this chapter is formed in the following manner:

- Section 4.1: brief overview of the ARM architecture.
- Section 4.2: translator high level architecture.
- Section 4.3: architectural differences between ARM and ρ-VEX.
- Sections 4.4, 4.5, 4.6: low level translation techniques
- Section 4.7: combining all the emulation techniques.

4.1 ARM architecture

The ARM architecture is a family of RISC CPU architectures developed by British ARM (Advanced RISC Machine). ARM develops CPU architectures which are then licensed to other companies that design their own products based on those architectures. ARM architecture currently, is the most broadly used architecture with more than 100 billion devices using CPUs with ARM architecture. ARM was initially developed by Acorn Computers in the 80s with the intent to incorporate it in its personal computers. In the early 90s ARM Holdings was formed in order to focus primarily on the CPU and IP core development. Many companies have developed cores that include the ARM architecture including Apple, NXP Semiconductors, Qualcomm, Samsung, Nvidia, and many more.

There are numerous ARM architectures created by either ARM itself or other third party companies. When it comes to naming convention about the ARM architecture there are three distinct ways to do it: the architecture family, the architecture version, and the processor that implements the architecture. A clear listing of all the above can be found here [26]. As a convention the architecture family will be referred to as ARM# e.g., ARM5 and the architecture with ARMv# e.g., ARMv3.

4.1.1 Architectural Highlights

ARM is a RISC architecture inherently, this means that simplicity is a key part of the architecture. A few attributes of the architecture that conform with its RISC nature are:

- It is a load/store architecture with support for 32-bit and 64-bit words.
- Uniform register file.
- Simple addressing modes.
- Fixed uniform instruction lengths.
- No microwords

There is not support for unaligned memory accesses in the old versions (prior to ARMv6), however later versions support unaligned access for halfwords and single words. The most important attributes that result from the architecture choices are the following:

- Reduced implementation size.
- Performance.
- Low power consumption
- High code density.

Another key attribute is the wide variety of architecture choices. ARM architecture evolves through each version. With every new version, new capabilities and instructions are added. However this does not make older versions obsolete (after ARMv3) since older versions are simpler and the processors that implement them are consequently also

simpler. This makes the ARM architecture almost modular with each version fitting different needs. As a result there are three main ARM architectural groups and each group can have various extensions. These groups are the following:

- A (Architecture): High performance computing such as mobile devices.
- R (Real-time): For embedded systems such as automotive.
- M (Micro-controller): For the micro-controller industry.

The extensions that add functionality to certain architectures are the following:

- Security extensions.
- SIMD instructions.
- Virtual machine extensions.
- Cryptographic extensions.

Furthermore, "THUMB" is a subset of the main architectures that was introduced in the ARM7TDMI architecture. It sacrifices functionality but it can have a more compact instruction size, i.e., a 16 bit instruction length instead of 32.

The ARM architecture also incorporates some features that allow for better performance and control. Perhaps the three most important attributes that allow the above, are the conditional execution, the fact that shifts are incorporated into the datapath and the CPU operation modes. These features are explained roughly below.

4.1.1.1 ARM Predication System

Almost all ARM instructions are conditionally executed, this feature is called predication. In other words even though all instructions are fetched into the core, not all of them are executed. Each instruction has a a 4 bit field that indicates whether the instruction will be executed or not. Those bits are indications for flags, namely:

- N, for negative.
- Z, for zero.
- C, for carry.
- V, for overflow.

Those condition bits are contained in the 4 MSBs of the instruction in the above ordering, i.e. N in bit 31, Z in bit 30, C in bit 29 and V in bit 28. The processor also contains four 1 bit registers with the aforementioned flag values which are updated by specified instructions. When an instruction is decoded the flag bits are compared to the flag values in the processor registers and if the values are equal then the instruction is executed, if not then the instruction is not executed. If for example an instruction caused an overflow

the V flag will be '1'. Next instruction that comes and has a flag value of '0', it will not be executed. Whether an instruction's result will update the processor flag values is determined by a field in the instruction. As a result not all instructions update the flag values. Table 4.1 shows an execution example. Initially the processor flag values are NZCV=1010 e.g. negative flag is '1', zero flag is '0', carry flag is '1' and overflow flag is '0'. The first instruction that comes(add) has flag values of "1010" which is the same as the ones stored in the processor registers. As a result the instruction is executed. Furthermore the instruction does not update the flag values. The next instruction that arrives(sub) also has flag values of "1010" which are equal to the processor flag values. As a result it is also executed normally, however this instruction will also update the flag values of the processor. If the subtraction causes an underflow the V flag value will change from '0' to '1'. This is actually the case in the example which results in the next instruction(cmp) not being executed.

Table 4.1: ARM predicate example. First column shows the time, second column shows the instruction, third column shows the instruction flags, fourth column shows whether the instruction will update the flag values or not and the final column are the processor flag values.

Time	Instruction	NZCV	Update?	N	Z	C	V
T_0	add	1010	NO	1	0	1	0
T_1	sub	1010	YES	1	0	1	0
T_2	cmp	1010	YES	1	0	1	1
T_3	X	X	X	1	0	1	1

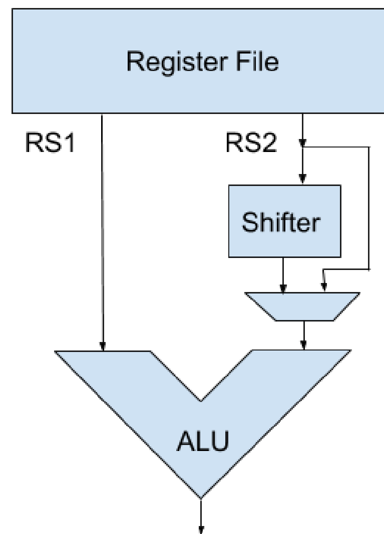
More specifically the flag bits represent the following conditions:

- N: If the result is regarded as a signed two's complement, then if the result is negative, N = 1 and if the result is positive, N = 0.
- Z: If the result is zero then Z = 1 and Z = 0 otherwise.
- C: Depending on the operation C indicates one of the following:
  - If the instruction was an addition, then C = 1 if a carry was produced and C = 0 otherwise.
  - If the instruction was a subtraction, then C = 0 if a borrow was produced and C = 1 otherwise.
  - If a shift instruction was incorporated in the previous instruction, then C is set to the last bit shifted out of the register.
  - For all other cases C is left unchanged with the exception of a few special cases.
- V: If there was an two's complement addition or subtraction, V = 1 when an overflow occurs and V = 0 otherwise. If the instruction was not addition or subtraction, then V usually remains unchanged.

4.1.1.2 ARM Shift Mechanism

ARM architecture does not contain individual shift instructions. All shift instructions are incorporated into the processor datapath. The way the second operand is encoded in the instruction fields determines whether and how it will be shifted or not. As a result the shifting is performed on only the second operand before the execution stage. An example schematic is depicted in Figure 4.1.

Figure 4.1: ARM architecture shifter. No individual shift instructions exist, instead the second operand can be shifted before the instruction execution.



The second operand can be shifted or not. In all cases, the second operand is calculated within one cycle. This achieves a higher code density (since shift instructions are completely omitted) and higher throughput since the shift instructions are combined with other more basic ones.

4.1.1.3 ARM architecture CPU modes

The ARM architecture supports several modes of CPU execution. The CPU can be in only one state at a given time. The modes allow for graduated system resource control and prevent illegal access to data and resources. These modes are presented in Table 4.2.

The main program execution mode is the User mode; all usual programs are executed in this non-privileged state. Furthermore, there are two modes to handle interrupts, one for normal interrupt requests and one that require immediate handling. Abort mode protects the virtual or physical memory integrity, if for example a memory fetch is aborted prematurely. Undefined is the state when the CPU does not recognize the instruction, in this mode the core waits for a response by co-processors if the instruction is actually executed by them. The system mode is entered when an explicit instructions modifies

Table 4.2: ARM processor states.

Mode	Privileged?	Description	Versions
User	no	Normal program execution	All
FIQ	yes	Fast interrupt handling	All
IRQ	yes	Normal interrupt handling	All
Supervisor	yes	OS	All
Abort	yes	Memory protection	ARMv3+
Undefined	yes	Co-processor	ARMv3+
System	yes	Modifies the CPU state register	ARMv4+
Monitor	yes	Monitoring mode	TrustZone extension
Hyp	yes	Hypervisor mode	Virtualization extension
Thread	yes/no	User tasks for RTOSs	ARMv(6-7-8)M
Handler	yes	Dedicated for exception handling	ARMv(6-7-8)M

the execution mode register. Monitor mode is used by the security extension TrustZone. The Hyp mode is used in order to support the Popek and Goldberg virtualization requirements [27]. This mode is used for RTOS environments and bare-metal applications. Finally, the Handler mode is a mode specifically dedicated for exception handling.

4.1.2 ARM registers

The ARM processor contains 37 registers in total, 32-bit wide. Of those, 31 are general purpose registers; however, not all are mapped to separate physical registers. In addition, 6 of them are status registers which also are not all mapped to physical registers depending on the ARM architecture. What this means, is that a specific register address might refer to multiple physical registers, which depends on the processor execution state. Out of the 31 general purpose registers R15 is the program counter. R0 to R7 are 8 completely general purpose registers and refer to the same physical register in all processor modes. R8 to R14 refer to different physical registers depending on the processor mode. R8 to R12 have can refer to two physical registers and R13 to R14 to six for a total of 22 physical registers. Combined with the PC and the general purpose registers mentioned above, they make for a total of 31 physical registers (plus 6 execution status registers).

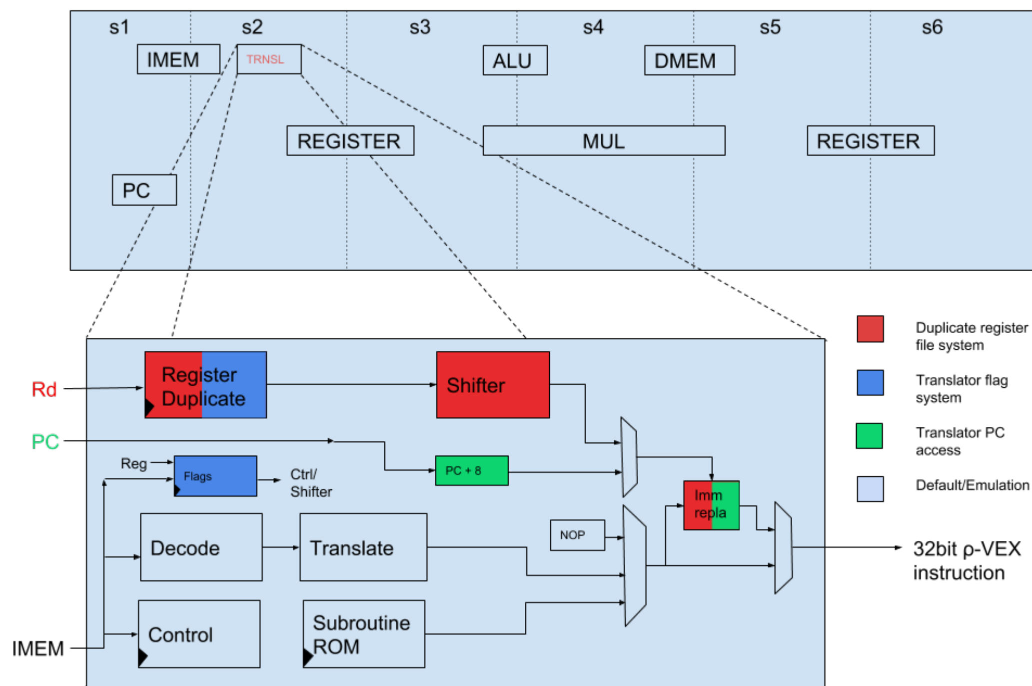
However, the implementation from ARM to ρ-VEX presented in this thesis, does not support any kind of different processor execution mods except the default User mode. This means that only 16 physical registers are utilized in the translator implementation, 15 general purpose registers and 1 for the program counter. Besides the R15 that is used as the program counter, R13 is used as the stack pointer(SP), and R14 is used as the link register(LR). Attention needs to be given to register R0; ARM does not have a zero hardwired register, as a result when R0 is referenced it should be mapped to another register address since R0 in ρ-VEX is hardwired to zero. In the provided simulator implementation, all register addresses are incremented by 1 to avoid references to R0.

4.2 High-Level Description of the Translator

The high level RTL of the translator is depicted in figure 4.2. The translator resides in the second pipeline stage. By default the output of the translator should operate in a combinational way, i.e., the output should be dependent on the current fetched instruction from the instruction memory. However, when special cases are detected, after the initial instruction dispatch, the subroutine ROM serves as the instruction memory, dispatching further instructions. These special case are described in detail in the following subsections.

The colored modules and inputs are parts of different optional implementations which will be explained in the following subsections. The gray colored modules are the **minimum required hardware** for translating the instructions and exist in all versions regardless.

Figure 4.2: High level schematic of the ARM binary translator. The translator is within the second pipeline stage. Colored modules within the translator RTL are the optional additions and the light gray ones exist in all versions.



4.2.1 Decode unit

The decode unit is responsible for splitting the incoming ARM instruction into the respective fields. This is a combinational circuit and it simply splits, reorders and copies

incoming values. The decoding process is presented later on in details along with the explanation of the individual instructions.

4.2.2 Translate unit

This unit is responsible for generating a ρ-VEX instruction based on the inputs from the Decode unit. This module operates in a combinational way and is not aware of the execution state of the processor or the translator. The control module is responsible to utilize, or not, the output of this unit. Essentially this unit can be implemented with two LUT tables, one that generates the ρ-VEX instructions and one that generates the controls needed to multiplex the decoded values from the Decode unit.

4.2.3 Subroutine ROM

This unit contains the subroutines required to emulate certain instructions. These instructions are generated sequentially one on every cycle. Once a subroutine has been activated the unit generates the required instructions starting from the next cycle. This unit is controlled by the Control unit.

4.3 Architectural Differences Between ARMv4 and ρ-VEX.

The ARM architecture version that was chosen was ARMv4. The choice was made because this version is amongst the most compatible. Furthermore, it is still compatible with the toolchain provided by ARM since ARM versions from ARMv3 and backwards are now obsolete. This version does not include any instruction expansion sets such us SIMD and DSP instructions since this would introduce even greater incompatibility and complexity.

Nevertheless, there are numerous incompatibilities that introduce either a significant instruction overhead, or increased hardware complexity. Namely these incompatibilities can be grouped in the following categories:

- Pipeline hazards.
- Shifting mechanism.
- Direct access to PC.
- Conditional execution.
- Overflow detection.
- Memory addressing modes.

4.3.1 Pipeline hazards

The pipeline hazard problems and solutions are exactly like the ones in the RISC-V implementation (see Section 3.6) the solution is also the same and it is to simply insert NOPs coupled with backwards jumps whenever a hazard is detected[1].

4.3.2 Shifting mechanism

One of the most important characteristics of the ARM architecture is the incorporation of the shifting mechanism inside instructions. As a result there are no individual shift instructions. The shifting mechanism is embedded in the second operand datapath and all data shifting is done within one cycle before the data is processed. This is in contrast to how the ρ-VEX architecture operates where the shifts are treated as individual instructions which are executed separately. As an example consider the following C line of code:

```
a = a + (b << 2);
```

In ρ-VEX this would be translated into two instructions:

```
shl $r0.b = $r0.b, 2
add $r0.a = $r0.a, $r0.b
```

First 'b' is shifted left by 2 and then the result is added to 'a'. However in ARM architecture this C line of code would be translated to one instruction like this:

```
ADD Ra, Ra, Rb, LSL #2
```

As a result it is clear that if an instruction has an incorporated shift, ρ-VEX cannot execute it directly. For the above problem two solutions are proposed one of which was implemented in the simulator: First solution, is to emulate all the shifts with ρ-VEX instructions and the second, is to perform the shifting part of the instruction, within the translator. The second option will reduce the instruction overhead and execute the shifts seamlessly inside the translator, however, it requires core modifications and extra hardware.

As can be seen on Table 4.3 shifts are incorporated in a big proportion of the instructions. This is logical because the compiler tries to utilize the shifter as much as possible while at the same time reducing code size. On first sight, a strange output can be seen for the programs *crc.c* and *des.c* where they seem to be more than the actual instruction dispatched. This can actually be the case because of two special load/store instructions where multiple consecutive locations of memory are accessed. However all of these accesses are initialized by one instruction. As a result even though one instruction is dispatched, this can be translated to several load/store instructions.

The choice between the two suggested implementations, should be based on appropriate profiling of the main application usage case of the to-be-designed system. If many shifts are called (especially rotations as will be described below), then the second option should be chosen; if not, then the first option should be chosen. More details on the instruction overhead difference between the two options will be presented in their according sections below.

[1] The pipeline hazards and the according solutions were not tested since the simulator does not include pipeline configuration

Table 4.3: Shift instructions in powerstone benchmark

program	Total Cycles	Total Shifts(Rotations)
bcnt.c	7171	5648 78% (5038 89% of shifts)
blit.c	56451	34621 61% (26375 76% of shifts)
compress.c	198448	295863 74% (116994 79% of shifts)
crc.c	53697	67155 125% (38915 57%116994 of shifts)
convolution.c	1330596	1018202 76% (701781 69% of shifts)
engine.c	1120660	949296 84% (372956 39% of shifts)
des.c	131401	149325 113% (86119 57% of shifts)
g3fax.c	2552428	1390576 54% (1339656 96% of shifts)
jpeg.c	8929392	6508136 72% (2279936 35% of shifts)
ucbqsort.c	466403	187050 40% (186045 99% of shifts)
v42.c	4597173	3613391 78% (1962181 54% of shifts)
pocsag.c	90407	64767 71% (57594 88% of shifts)

4.3.2.1 Solution 1. Instruction emulation.

The simplest solution and easiest in terms of implementation is to simply emulate all the shifting sub-instructions with ρ-VEX instructions. The emulation procedure consists of three steps. The first step is to check whether or not a shifting is actually required. In some cases the second operand is not shifted at all, therefore, the rest of the steps are not required. If shifting is not required, then the instruction is dispatched directly.

If shifting is required, then the shifted second operand will be calculated before the actual instruction is executed, and it will be stored in a temporary register. The ARM architecture supports five shift mechanisms:

- ASR Arithmetic shift right
- LSL Logical shift left
- LSR Logical shift right
- ROR Rotate right
- RRX Rotate right with extend.

All of the above are directly translated to ρ-VEX instructions except the two rotate instructions. The two rotate instructions can be emulated by the ρ-VEX "ADDCG" instruction. It needs to be noted that "ADDCG" is not intended specifically for rotations (it is used for addition between numbers larger than 32 bits). As a result this instruction is highly inefficient in performing rotations as it performs only one bit left rotation per cycle. For example, for a rotate left by 20, "ADDCG" needs to be called 20 times. A simple modification that is suggested for consideration is to modify the barrel shifter so that it can perform rotations.

The third step is to resume correct program flow and execute the instruction that requires the shifted result. Since there is no mechanism to pause the ρ-VEX PC, the shifting instructions must be followed by a direct backwards jump instruction ("goto"),

that jumps to the original PC-shift_cycles.Finally, the instruction is dispatched with the second operand now being shifted and available in a general purpose register. Since the translator that issues the instructions and the write back stage of the processor, are 3 stages apart, the translator needs to wait for 4 cycles before dispatching the instruction that utilizes the shift result. During this time the translator should dispatch "NOPs". Furthermore, it must ignore the next 4 fetched instructions. As a result on normal shifts the shift_cycles = 4 and for rotations it is at least 4 unless the rotate is larger than 4, on which case shift_cycles = rotations.

As can be understood from the above, the overhead for a shift is at least 4 instructions depending whether it is a rotation or not. An example "ADD" with a shifted second operand program flow can be seen on table 4.4. Initially the translator receives an ARM "ADD" instruction with a shifted second operand (shifted left by 2). The translator begins the shifting subroutine and dispatches a "shl" instruction to calculate the second operand. The result is stored in a random unutilised register, in this case r40. While the second operand is calculated the translator dispatches NOPs. After three cycles the result is in r40 and the translator issues a jump back to the initial PC and this time the add instruction is dispatched. However instead of using register Rb as the second operand it uses r40 which now contains the shifted result.

Table 4.4: Second operand calculation program flow. The "ADD" instruction has a shifted second operand. First the shifted operand is calculated and then the main instruction is executed.

PC Value	Fetched instr(ARM)	Dispatched instr(VEX) NZCV=1000
PC	ADD Ra. Ra, Rb LSL#2	shl $r0.r40, $r0.b, 2
PC+1	X	NOP
PC+2	X	NOP
PC+3	X	NOP
PC+4	X	goto -4
PC	ADD Ra. Ra, Rb LSL#2	add $r0.Ra, $r0.Ra, $r0.R40

4.3.2.2 Solution 2. Register file live copy.

Another solution to the problem of shifting the second operand is to maintain a live copy of the register file within the translator. This will allow the translator to calculate the shifted operand internally. After the shifted value is calculated it can be driven as an immediate value to the second operand datapath. These modifications can be seen in red in figure 4.2. Since in the current implementation only the *User Mode* registers are used, the register file copy size needs to be only 16 registers wide. As seen above in the emulation implementation, the overhead that results in most cases is quite significant. Due to this, the live register copy technique was chosen in the provided implementation.

The only modification that is required on the ρ-VEX is a direct connection of the target register value Rd. i.e. before the register file is written in stage 5 of the pipeline, the value should also be driven to the translator. However, for instructions that contain a shift, the translator can no longer dispatch instructions within one cycle, since it needs

to read the internal register file before calculating the shifted operand. This means that whenever an instruction with shift is fetched, the translator initially dispatches a backwards jump to the same instruction i.e. a “goto 0” while it calculates the shifted operand and on the next cycle it dispatches the instruction.

Besides the above problem the translator control should also handle further problems that can occur due to pipelining; and more specifically the RAW(Read After Write) hazard. This can occur if the translator register file copy is out of date and the translator tries to read the old values. As mentioned above the translator and the register write back are 3 stages apart see figure 4.2, as a result if an instruction containing a shift, tries to read a register whose value is out of date, the translator should dispatch “NOPs” along with a jump backward instruction to provide time to the register file to be updated. Essentially, the duplicate register file gets updated at the same time as the normal register file. As a result the RAW hazards occur for both of the register files, at the same time and for the same instructions. As a result, the same logic and procedure that is used for normal pipeline hazards can be used for this problem. The duplicate register file still introduces an overhead mainly due to pipeline hazards.

Overall, both solutions have their advantages and disadvantages. The instruction emulation on one hand, has the advantage of implementation simplicity, since no core modifications are required. Also the only extra required hardware for the emulation implementation, is the control logic within the translator that keeps track of the subroutine execution. Nevertheless, the first solution introduces a significant instruction overhead. The second solution is the same as the first with the exception of a modification to the ρ-VEX shifter which will allow rotations. This will result again in a significant overhead, though slightly reduced compared to the first solution.

The third and final solution on the other hand, introduces a much smaller overhead compared to the other two solutions, but requires some core modifications, and a larger and more complex translator(16x32 register file, multiplexers, shifter) which can potentially result in performance reduction.

4.3.3 Direct access to PC

In ARM architecture the PC is located in R15 which is a register that can be accessed like any other register. In ρ-VEX the PC is not accessible by normal instructions. This problem can be split into two categories: One, when an instruction tries to write the PC and another when an instruction tries to read the PC.

4.3.3.1 Reading the Program Counter

When an instruction reads the PC, the value that should be provided is the PC plus 2 (8 bytes). The 2 LSB are always 0 since the PC values are always word aligned. A special case to the above rule is the “stm” and “str” which can read either PC+2 or PC+3. The choice between the two is implementation defined; for our case PC+2 is chosen for simplification reasons. As a result, whenever PC is read the value that is provided should always be PC+2.

Similar to previous cases, two options are provided: one is to simply drive the current PC value to the translator as seen in figure 4.2 with the green color, and the other is

to emulate the instructions that require the PC. The procedure is the same as the one shown in table 3.4 in chapter 3 with the exception of the subtraction step. Instead of subtracting a '1' it is added in order to acquire PC+2.

4.3.3.2 Writing the Program Counter

Any instruction that targets the PC register must be treated as a jump. As a result for every instruction that is fetched, a check must be performed to determine whether or not the target is PC (R15 in the ARM architecture). If the target is indeed PC then two steps are required. The first step is to execute the instruction and save the result to a temporary register. The second step is to jump to the address of that register. The execution of the first step is straight forward, however the second step requires emulation since no ρ-VEX instruction can perform register relative jumps. These emulation instructions can be seen below:

```
1 ARM and R15, R14, R13; //Fetched instruction
2 VEX and R40, R14, R13; //Dispatched instruction , store in r40 instead of PC
3     and R62, R63, 0xFFFFFFFF; // copy link register to a temporary register
4     sub R40, R40, 0xFFFFFFFF; // subtract 1 from R40
5     igoto;//jump to R40
6     and R63, R62, 0xFFFFFFFF; // copy R62 back to LR
7 Resume proper flow
```

When an instruction that targets r15 arrives, the instruction is dispatched in the same cycle, but instead of storing the result to r15, it is stored in a random temporary unused register such as r40. After that the contents of the Link Register are also copied to a random temporary location such us r62. Next a 1 is subtracted from the jump target inside the temporary register R40. This is done because the jump, which is performed in the next step, is performed 1 cycle before normal execution is resumed. Finally the original Link Register value is copied back to the LR from the temporary location. Essentially each instruction that targets the PC introduces an overhead of 4.

4.3.4 Conditional Execution

One of the most important characteristics of the ARM architecture is the conditional execution. This means that instruction execution is conditional, i.e. even though all instruction are fetched and parsed, not all are executed. Each instruction has 4 bits out of 32 that indicate whether the instruction will be executed or not. Even though this reduces the available bits for encoding in the instructions, it compensates for the lack of a branch prediction unit in the ARM architecture. The flag bits can be updated depending on the result of the execution of previous instructions. Any fetched instruction has the bits in the instruction field compared to the flag bits, if they are equal the instruction is executed, if not the instruction is not executed. This system is called predication.

ρ-VEX on the other hand, does not have a predication system, instead it utilizes branch registers. This means that a system is needed to support the predication of ARM architecture. In addition, not all instructions update the flag values. whether or not an instruction updates the flags is set by the 'S' bit which is bit 20 of the 32bit instruction. This is the same for all instructions capable of updating the flags.

There are two parts in the predication system. First, is the flag updating system and another one is to make the comparison between instruction and processor flags before execution in order to determine whether or not the instruction will be executed. Generally for both parts there are two options, one is emulation and the other is to execute them locally on the translator.

4.3.4.1 Predication system emulation

The first option is to implement both flag updating and comparison, with ρ-VEX instructions emulation. The flags should be stored in the 4 LSBs or MSBs1 of a dedicated unutilized register. An instruction should be executed only if the flags of the instruction are equal to the flags of this register. This can be implemented with a simple 'cmpeq' instruction that dispatches the instruction flag values as an immediate value and compares them to the register that contains the processor flags. The result should be stored in a branch register. Next, a branch instruction checks the result of the comparison and either jumps back to PC, hence executing the instruction, or proceeds to PC+1, thus bypassing it. This procedure can be seen below:

```
1 PC and r2, r3, r4\\triggers flag comparison
2 PC cmpeq b1, fr, imm\\compare the flag reg. to the instruction flags
3 PC+1 NOP
4 PC+2 NOP
5 PC+3 NOP
6 PC+4 goto −5\\jump to PC
7 PC    br −1\\branch to PC
```

This procedure introduces an instruction overhead of 5 for every instruction that is predicated. Next, the flag update procedure is examined. An instruction that is set to update the flags, does not always update all the flags, here an example is presented where one flag is updated. The following instructions show how the flag update can be emulated:

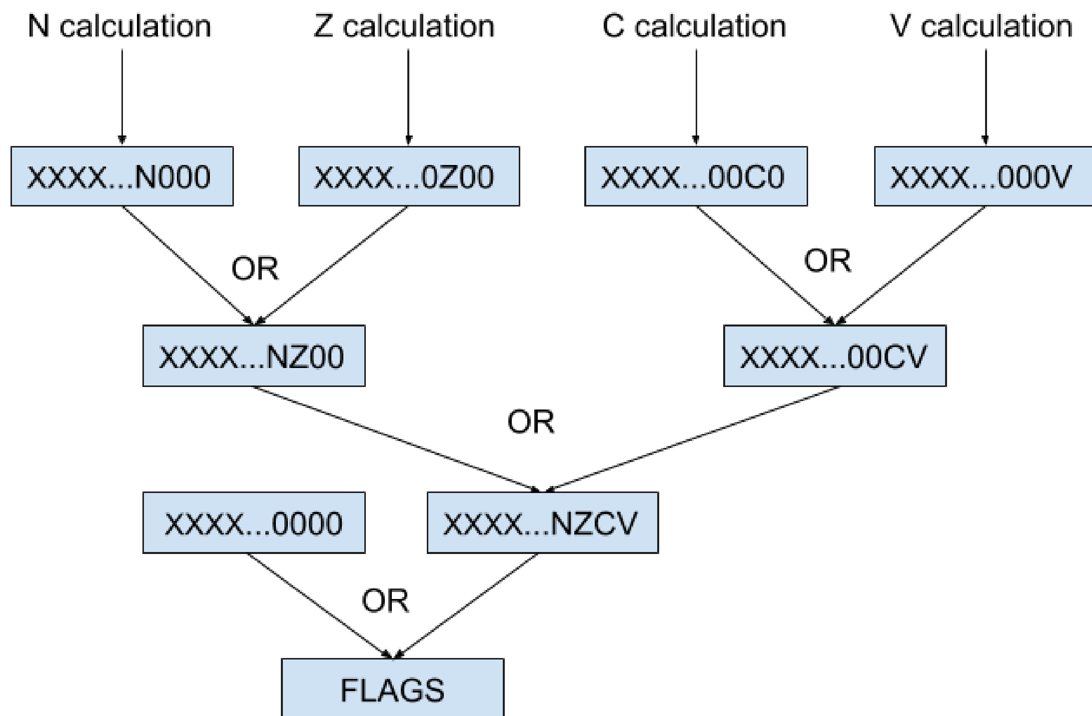
```
1 Step 1 PC:    Flag_updating_instruction\\dispatch instruction
2 Step 2 PC+1: NOP\\wait for the instruction to execute
3 Step 3 PC+2: NOP\\wait for the instruction to execute
4 Step 4 PC+3: NOP\\wait for the instruction to execute
5 Step 5 PC+4: flag value calculation Rd=a1\\Store the N flag in a1
6 Step 6 P+n+5: NOP\\wait for flag calculation
7 Step 7 PC+n+6: NOP\\wait for flag calculation
8 Step 8 PC+n+7: NOP\\wait for flag calculation
9 Step 9 PC+n+8: shl a1, a1, 3\\place the N flag in a4(3)
10 Step 10 PC+n+9: NOP\\wait for flag calculation
11 Step 11 PC+n+10: NOP\\wait for flag calculation
12 Step 12 PC+n+11: NOP\\wait for flag calculation
13 Step 13 PC+n+12:OR Fr, a1, Fr\\replace the old flag values with the new
    ones
14 Step 14 PC+n+13:goto −14\\jump back to normal program flow
```

Listing 4.1: Flag update emulation instructions.

Initially at step 1 an instruction that updates the N flag is dispatched. The instruction is executed normally (supposing that the predication is 1) and then three NOPs are dispatched to allow for the result to become available in the register file. After that at

step 5 the flag is calculated. Note that the flag calculation duration is not constant and varies based on the instruction and the data itself. It can vary anywhere between 4 cycles up to 10. After the calculation is finished the flag is placed in register a1. The next step is to replace the old flag in the flag register Fr. After the bits are shifted in their appropriate positions, all values are "ORed" and placed in the flag register. Figure 4.3 shows how all the flags are updated in the flag register from the individual flag values.

Figure 4.3: Flag register update emulation. Requires three cycles and eight instructions.



From the above procedure the precise way of flag calculation was omitted. This is intentional because the exact procedure of calculating the flag value depends on the individual instructions. These can be seen in the Appendix B or the second operand calculation chapter 4.5.

4.3.4.2 Translator predication system

The other choice for implementation regarding the predication system is to embed the required functionality inside the translator. This should be implemented in combination with the register duplicate solution presented here 4.3.2.2 because the flag calculation depends on at least one of the three operands of an instruction. Contrary to the emulation solution presented above, the flags are stored locally within the translator. As a result the required logic for generating and updating the flag values reside within the translator.

What follows is with the assumption that the register file duplicate system 4.3.2.2 is used.

Initially we examine the flag comparison procedure. Since everything is maintained and calculated within the translator, the flags are now stored locally within four 1-bit registers. The translator needs to compare the flags stored locally with the ones in the instruction fields. If they are equal then the instruction is dispatched normally, if they are not, then a NOP is dispatched. Comparing the flags locally in the translator introduces zero overhead.

Next we examine the flag updating system. When an instruction is set to update the flags, the translator dispatches the instruction normally and afterwards sends NOPs while waiting for the result to be available to the register file duplicate. Once it is forwarded to the duplicate register file, the translator dispatches a backwards jump instruction while at the same time it calculates the new flags. Finally the next instruction is fetched and executed. While this subroutine executes, any fetched instructions are ignored. A sample program flow of this procedure can be seen below:

```
1 PC   and $r0.a = $r0.b, $r0.c//issues a flag update
2 PC+1 NOP//wait for $r0.a result to be available
3 PC+2 NOP//wait for $r0.a result to be available
4 PC+3 NOP//wait for $r0.a result to be available
5 PC+4 goto −4//jump backwards while calculating new flags
6 PC+1 next instruction
```

4.4 Overflow/Borrow detection

The ARM architecture utilizes overflow/underflow and borrow detection for the subtraction calculations. More specifically the ‘V’ flag is based on overflow detection and the ’C’ flag is sometimes based on the borrowed bit in subtractions. ρ-VEX does not support any form of overflow or borrow detection. Similarly to the previous cases the overflow detection mechanism needs to be either emulated or embedded within the translator.

4.4.1 Overflow detection

First we examine the overflow detection. Overflow can only be generated from the adder when an addition or a subtraction is performed. When the bits required to represent a number are not enough then an overflow occurs. As an example consider the following two numbers $a = 0110$, $b = 0101$ which in twos complement are equal to 6 and 5 respectively. When they are added we receive the number $a + b = 1011$ which in twos complement is equal to -5 which is obviously wrong. However, if we had one extra bit in our disposal this number would be represented as 01011 which in twos complement is 11. The other case from overflow comes from the subtraction. Consider the following example where $a = 0111 = 7$ and $b = 1010 = -6$ and we need to calculate a - b. The result should obviously be $a-b = 13$ however the twos complement subtraction provides the following result $0111 - 1010 = 0111 + 0110 = 1101 = -3$ (The negative of a twos complement number is $-a = \bar{a} + 1$) which is not correct. The general rule to detect overflow from addition and subtraction is the following:

- Addition

  - $(+A) + (+B) = -C$
  - $(-A) + (-B) = +C$

- Subtraction
  - $(+A) - (-B) = -C$
  - $(-A) - (+B) = +C$

In the case that the overflow is implemented within the translator then the flag unit as seen in figure 4.2 can implement the above logic and update the flags seamlessly. In the case of emulation, then the ‘C’ flag value calculation step 4.1 is replaced by a set of instructions that calculate the above logic depending if it is addition or subtraction.

4.4.2 Borrow detection

The borrow detection occurs only in subtractions and can sometimes update the ‘C’ flag. Borrow occurs when the true subtraction result is less than 0 when the operands are treated as unsigned integers, so the subtraction a - b will produce a borrow in the following case:

```
1 if (unsigned(a) < unsigned(b))
2 {
3     return 1;
4 }
5 else
6 {
7     return 0;
8 }
```

Similarly to the above case this can either be implemented within the translator or by emulation.

4.4.3 Carry detection

This can occur only in addition and returns a 1 when there was a carry out generation from the 32 bit addition. Essentially occurs when the result is bigger than $2^(32) - 1$ in unsigned arithmetic. Again this can be implemented in the translator or emulated. In the case of emulation, the “addcg” instruction can be used to detect the carry out of the addition.

4.5 Second operand calculation

This section describes the way the second operand is decoded and calculated. In section 4.3.2, a disription was given as to how to implement the shifting mechanism that is required to calculate the second operand. Regardless of the chosen implementation, the same decoding procedure should be followed and the same results should be yielded. As mentioned earlier, ARM architecture does not have separate shift instructions, instead all the shifting functionality is embedded in the instruction fields which determine how

and if the second operand will be shifted. We will examine how to decode the fields that specify the second operand. Generally there are 11 formats that determine how and if the second operand will be shifted and those formats are the following:

- Immediate.
- Register.
- Logical left shift by immediate.
- Logical left shift by register.
- Logical right shift by immediate.
- Logical right shift by register.
- Arithmetic right shift by immediate.
- Arithmetic right shift by register.
- Rotate right by immediate.
- Rotate right by register.
- Rotate right with extend.

Arm architecture specifies that besides the shifted operand, the shifter should also produce a carry out which sometimes updates the C flag. This however in the ρ-VEX cannot be the case since the shifter cannot produce these values. The carry out calculation will either be emulated or generated within the translator depending on the implementation that is chosen (see section 4.3.2). In this section, besides presenting the shifted value calculation, the carry out generation from the shifter is also described. This value is used sometimes to update the 'C' flag values as mentioned in the previous section 4.3.4.1.

4.5.1 Immediate generation

31 28	27 26 25	24 21	20	19 16	15 12	11 8	7 0
cond	0 0 1	opcode	S	Rn	Rd	rotate_imm	immed_8

This subsection describes the decoding procedure of the second operand for immediate values. Essentially it describes how immediate values are produced as a second operand. The immediate value is acquired by rotating an 8-bit immediate to an even position in a 32-bit word. If the rotation immediate is zero, then the carry-out from the shifter is the C flag, otherwise it is the 31st bit of the shifter operand. The pseudo code for these operations can be seen below:

```
shifter_operand = immed_8 Rotate_Right (rotate_imm * 2)
if rotate_imm == 0 then
    shifter_carry_out = C flag
else /* rotate_imm != 0 */
    shifter_carry_out = shifter_operand[31]
```

The shifter_carry_out logic should be either embedded in the translator logic or emulated depending on the implementation chosen here 4.3.4.1. If both the predication system and shifting mechanism are implemented inside the translator, then the shifter_carry_out calculation is trivial and the above procedure should be followed. If the shifting mechanism is not implemented inside the translator then the procedure explained here should be followed 4.3.2.2. Essentially the translator should dispatch NOPs to wait until the second_operand result becomes available. At this point the 31st bit should be extracted within the translator in order to update the C flag. If neither the predication, nor the shifting mechanism is implemented within the translator, i.e. everything is emulated, then the procedure shown here 4.3.4.1 should be followed. The flag value calculation can be provided by a 'tbit' instruction that extracts the 31st bit of the result or the C flag from the flag register. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: determines the destination register.
- rotate_imm: determines the rotation amount.
- immed_8: determines the initial immediate value to be rotated.

4.5.2 Unmodified register

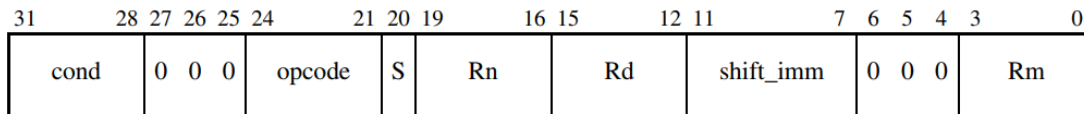
31–28	27 26 25	24–21	20	19–16	15–12	11 10 9 8 7	6 5 4	3–0
cond	0 0 0	opcode	S	Rn	Rd	0 0 0 0 0	0 0 0	Rm

This format dictates that the value of the register Rm is provided completely unmodified. Furthermore the carry_out flag is equal to the C flag, which in the case of emulation it can be provided by a "tbit" instruction which copies the C flag from the flag register to a destination register. Having said that there is nothing else non-trivial about this decoding procedure. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.

- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: determines the destination register.
- Rm: Second operand register.

4.5.3 Logical left shift by immediate



This format dictates that the second operand will be provided by a register value logically shifted to the left by an immediate. The data inside the register Rm is shifted left and zeros are pushed in from the right. The shift amount is provided by an immediate value. The carry out of this operation is the last bit that was popped out of the left. The pseudo-code for the above functionality can be seen below:

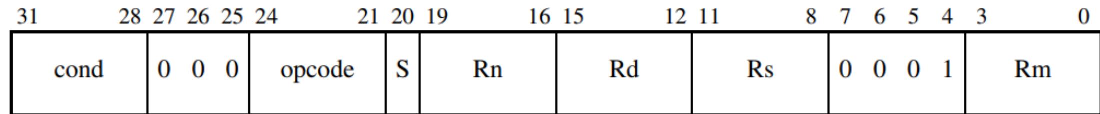
```
1 if shift_imm == 0 then /* Register Operand */
2     shifter_operand = Rm
3     shifter_carry_out = C Flag
4 else /* shift_imm > 0 */
5     shifter_operand = Rm Logical_Shift_Left shift_imm
6     shifter_carry_out = Rm[32 - shift_imm]
```

The shfiter_carry_out should be calculated with the procedure described above 4.5.1. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: determines the destination register.
- shift_imm: Denotes the shift amount.
- Rm: Second operand register.

4.5.4 Logical left shift by register

This format dictates that the second operand will be provided by a register value logically shifted to the left by a register value. The data inside the register Rm is shifted left and zeros are pushed in from the right. The shift amount is provided by the 8 LSBs of



register Rs. The carry out of this operation is the last bit that was popped out of the left. If the shift amount is larger than 32 then the carry out is zero. The pseudo-code for the above functionality can be seen below:

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Logical_Shift_Left Rs[7:0]
    shifter_carry_out = Rm[32 - Rs[7:0]]
else if Rs[7:0] == 32 then
    shifter_operand = 0
    shifter_carry_out = Rm[0]
else /* Rs[7:0] > 32 */
    shifter_operand = 0
    shifter_carry_out = 0
```

The shfiter_carry_out should be calculated with the procedure described above 4.5.1. The above can be emulated by the translator in 4 stages. Since there are four conditions the translator needs to determine which one is true without having a picture of the values or the result. This can be done by monitoring the incoming instruction after a compare and branch. i.e. for every condition above a compare and branch should be performed sequentially and conditionally. The branch should point to the original instruction if the comparison is true and to the next one if the comparison is false. This way the translator can monitor the program flow and dispatch another set of compare and branch instructions or proceed to the next instruction. As an example suppose that Rs[7:0] ¡ 32 which corresponds to the second condition. Initially the translator dispatches the instruction required to execute the first condition as follows:

```
PC   and RX, Rs, imm(0x000000FF)
PC+1 NOP
PC+2 NOP
PC+3 NOP
PC+4 cmpeq BX, RX, imm(0)//compare RX with zero
PC+5 goto -5//jump back to PC
PC   br BX, -1//branch to PC if the above comparison is true
```

At this point the next instruction that will arrive is going to be either the same that is currently being executed or the one located in PC+1[2]. Since Rs[7:0] ¡ 32 the latter is the case and the instruction that arrives is the one contained in PC+1. As a result

[2] There is an extreme case where the two arriving instructions are exactly the same which can result in a deadlock. This has not been tested since the simulator implementation does not include the emulation procedures. However a possible failsafe solution is to calculate all result cases and all "if else" conditions and store them in registers. As a result, the condition registers will contain either a 1 or a 0 but only a single register will be 1. These registers need to be left shifted by 31 and then signed right shifted by 31 again in order to copy the LSB to all 32 positions. This way when the condition registers are ANDed

the translator knows that the comparison was not true and ignores the next instruction. Next it dispatches the instructions required for the second condition:

```
PC+1 cmplt BX, RX, imm(32)
PC+2 NOP
PC+3 NOP
PC+4 NOP
PC+5 cmpeq BX, RX, imm(0)//compare RX with zero and store in branch
    register
PC+6 goto −6//jump back to PC
PC   br BX, −1//branch to PC if the above comparison is true
```

Now that the same instruction will arrive the translator knows that the comparison was true and continues with the corresponding flag calculation procedure:

```
PC+1 sub RX, imm(32), RX
PC+2 NOP
PC+3 NOP
PC+4 NOP
PC+5 tbit RX, Rm, RX
```

Similarly when the comparison proceeds to the other conditions the emulations would be the same with different comparison instructions. The above emulation procedure should be used whenever the shifter_carry_out is requested for updating the C flag. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: determines the destination register.
- shift_imm: Denotes the shift amount.
- Rm: Second operand register.

4.5.5 Logical right shift by immediate

31–28	27	26	25	24–21	20	19–16	15–12	11–7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	0	1	0	Rm

This format dictates that the second operand value is provided by a register value logically shifted to the right by an immediate value. The data inside register Rm is

with the result registers only on will remain with its contents unchanged, the rest will be reduced to 0. Then is a simple matter of ORing all the result registers to acquire the final value.

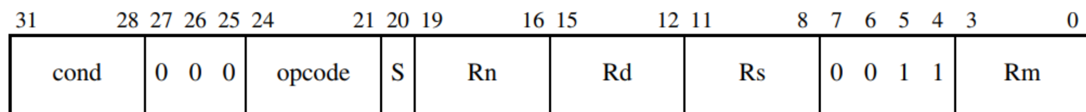
shifted to the right by an immediate value with zeroes pushed in from the left. The carry out of this operation is the last bit that was popped out from the right. The pseudo-code for the above functionality can be seen below:

```
if shift_imm == 0 then
    shifter_operand = 0
    shifter_carry_out = Rm[31]
else /* shift_imm > 0 */
    shifter_operand = Rm Logical_Shift_Right shift_imm
    shifter_carry_out = Rm[shift_imm - 1]
```

The shfiter_carry_out should be calculated with the procedure described above 4.5.1. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: determines the destination register.
- shift_imm: Denotes the shift amount.
- Rm: Second operand register.

4.5.6 Logical shift right by register



This format dictates that the second operand will be provided by a register value logically shifted to the right by a another register value. The data inside register Rm is shifted to the right by the value located in the 8 LSBs of Rs, with zeroes pushed in from the left. The carry out of this operation is the last bit that was popped out from the right. If the shift amount is larger than 32, then the carry out is zero. The pseudo-code for the above functionality can be seen below:

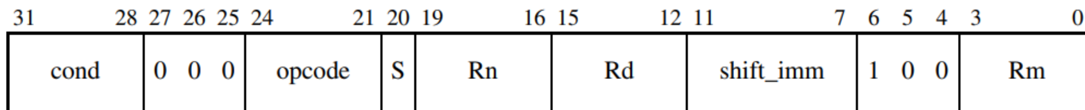
```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Logical_Shift_Right Rs[7:0]
    shifter_carry_out = Rm[Rs[7:0] - 1]
else if Rs[7:0] == 32 then
    shifter_operand = 0
    shifter_carry_out = Rm[31]
```

```
10 else /* Rs[7:0] > 32 */D
11     shifter_operand = 0
12     shifter_carry_out = 0
```

The shfiter_carry_out should be calculated with the procedure described above 4.5.1. The emulation procedure is similar as in 4.5.4. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: Determines the destination register.
- Rs: Determines the shift amount.
- Rm: Second operand register.

4.5.7 Arithmetic shift right by immediate



This format dictates that the second operand will be provided by a register value arithmetically shifted to the right by an immediate value. The data inside register Rm are shifted to the right by an immediate value in the range of 1-32 with the sign bit pushed in from the left. The carry out of this operation is the last bit that was popped out from the right. If the shift amount is larger than 32, then the carry out is zero. The pseudo-code for the above functionality can be seen below:

```
1  if shift_imm == 0 then
2      if Rm[31] == 0 then
3          shifter_operand = 0
4          shifter_carry_out = Rm[31]
5      else /* Rm[31] == 1 */
6          shifter_operand = 0xFFFFFFFF
7          shifter_carry_out = Rm[31]
8  else /* shift_imm > 0 */
9      shifter_operand = Rm Arithmetic_Shift_Right <shift_imm>
10     shifter_carry_out = Rm[shift_imm - 1]
```

The shfiter_carry_out should be calculated with the procedure described above 4.5.1. The emulation procedure is similar as in 4.5.4. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: Determines the destination register.
- shift_imm: Determines the shift amount.
- Rm: Second operand register.

4.5.8 Arithmetic shift right by register

31–28	27 26 25	24–21	20	19–16	15–12	11–8	7 6 5 4	3–0
cond	0 0 0	opcode	S	Rn	Rd	Rs	0 1 0 1	Rm

This format dictates that the second operand will be provided by a register value arithmetically shifted to the right by a register value. The data inside register Rm are shifted to the right by the value inside the 8 LSBs of register Rs, with the sign bit pushed in from the left. The carry out of this operation is the last bit that was popped out from the right. If the shift amount is larger than 32, then the carry out is zero. The pseudo-code for the above functionality can be seen below:

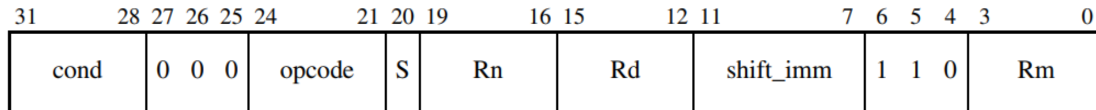
```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Arithmetic_Shift_Right Rs[7:0]
    shifter_carry_out = Rm[Rs[7:0] - 1]
else /* Rs[7:0] >= 32 */
    if Rm[31] == 0 then
        shifter_operand = 0
        shifter_carry_out = Rm[31]
    else /* Rm[31] == 1 */
        shifter_operand = 0xFFFFFFFF
        shifter_carry_out = Rm[31]
```

The shfiter_carry_out should be calculated with the procedure described above 4.5.1. The emulation procedure is similar as in 4.5.4. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.

- Rn: Determines the first source operand register.
- Rd: Determines the destination register.
- Rs: Determines the shift amount.
- Rm: Second operand register.

4.5.9 Rotate right by immediate



This format dictates that the second operand value will be provided by the value of a register rotated right by an immediate. The data inside register Rm are rotated to the right by an immediate value. Bits popped out of the right are inserted back into the value from the left. The carry out of this operation is the last bit that was popped out from the right. The pseudo-code for the above functionality can be seen below:

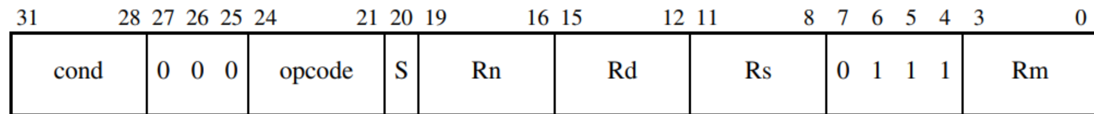
```
1 if shift_imm == 0 then
2     See \ref{}
3 else /* shift_imm > 0 */
4     shifter_operand = Rm Rotate_Right shift_imm
5     shifter_carry_out = Rm[shift_imm - 1]
```

The shfiter_carry_out should be calculated with the procedure described above 4.5.1. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: Determines the destination register.
- shift_imm: Determines the shift amount.
- Rm: Second operand register.

4.5.10 Rotate right by register

This format dictates that the second operand value will be provided by the value of a register rotated right by a register value. The data inside register Rm are rotated to the right by value contained in the 5 LSB of register Rs. Bits popped out of the right are inserted back into the value from the left. The carry out of this operation is the last bit that was popped out from the right. The pseudo-code for the above functionality can be seen below:

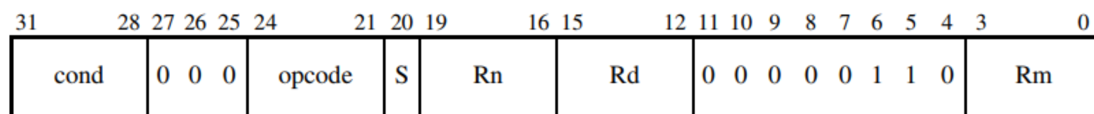



```
1 if Rs[7:0] == 0 then
2     shifter_operand = Rm
3     shifter_carry_out = C Flag
4 else if Rs[4:0] == 0 then
5     shifter_operand = Rm
6     shifter_carry_out = Rm[31]
7 else /* Rs[4:0] > 0 */
8     shifter_operand = Rm Rotate_Right Rs[4:0]
9     shifter_carry_out = Rm[Rs[4:0] - 1]
```

The shfiter_carry_out should be calculated with the procedure described above 4.5.1. The emulation procedure is similar as in 4.5.4. The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: Determines the destination register.
- Rs: Determines the shift amount.
- Rm: Second operand register.

4.5.11 Rotate right by with extend



This format dictates a right rotation on a 33-bit value using the C flag as the 33rd bit. Essentially, the value is obtained by right shifting Rm by one and replacing the vacated MSB with the carry flag. The bit that was popped out of the right is the carry out.

```
1 shifter_operand = (C Flag Logical_Shift_Left 31) OR (Rm Logical_Shift_Right
    1)
2 shifter_carry_out = Rm[0]
```

The field encoding annotation can be seen below:

- cond: The instruction flag values NZCV in bits 31,30,29 and 28 respectively.
- opcode: Main instruction opcode.
- S: Determines whether the instruction will update the flags.
- Rn: Determines the first source operand register.
- Rd: Determines the destination register.
- Rm: Second operand register.

4.6 Memory Addressing Modes

When it comes to memory access, ρ-VEX has one addressing mode, which is the following:

```
address = Rx + imm
```

Basically a signed immediate value is added to a signed register value to acquire the address. This however is not the case in ARM architecture since it has fifteen addressing modes. Furthermore, some addressing modes utilize post and pre-indexing. This means that sometimes the contents of the base address register are replaced with a new value before or after the memory access. The address calculation can be handled inside the translator in a relatively simple way, or emulated.The index replacement however, needs to be emulated regardless of the implementation chosen. All fifteen addressing modes are presented below.

4.6.1 Immediate offset

This mode calculates the address by adding or subtracting an immediate offset to the base register. This is the only case that the addressing mode has similarities to the ρ-VEX. The offset is an unsigned number which can be either added or subtracted to the base register. On the other hand, ρ-VEX architecture always adds a signed offset to the base register. For this reason, the translator should perform a sign extension on the offset before dispatching it. The operation logic can be seen below:

```
//Depending on the instruction
//the offset can be either offset_8
//or offset_12(See below)

if U == 1 then
    address = Rn + offset
else /* U == 0 */
    address = Rn - offset
```

- Rn: Base address register
- offset_12: Encoded in bits 11 down to 0.

- immedH: Encoded in bits 11 down to 8 in the instruction.
- immedL: Encoded in bits 3 down to 0 in the instruction.
- offset_8: immedH && immedL
- U: Encoded in bit 23 in the instruction.

If the translator handles the offset, then the register live copy should also be implemented (see 4.3.2.2). If the emulation approach is chosen then the following instructions will result in the complete address located in a register, which can then be used as is, i.e. offset of zero. The emulation procedure can be seen below:

```
if U == 1 then translator dispatches the following:
    add Rx1, Rn, offset
    goto -2
if U == 0 then translator dispatches the following:
    sub Rx1, offset, Rn
    goto -2
```

4.6.2 Register offset

This mode calculates the address by adding or subtracting a register value offset to the base register. The operation logic can be seen below:

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
```

- Rn: Base address register
- Rm: Offset register
- U: Encoded in bit 23 in the instruction.

If the translator handles the offset, then the register live copy should also be implemented (see 4.3.2.2). If the emulation approach is chosen then the following instructions will result in the final memory address located in a register, which can then be used as is, i.e. offset of zero. The emulation procedure can be seen below:

```
if U == 1 then translator dispatches the following:
    add Rx1, Rn, Rm
    goto -2
else if U == 0 then translator dispatches the following:
    sub Rx1, Rm, Rn
    goto -2
```

4.6.3 Scaled register offset

This mode generates the address by adding a shifted register value to the base address. The operation logic can be seen below:

```
case shift of
0b00 /* LSL */
    index = Rm Logical_Shift_Left shift_imm
0b01 /* LSR */
    if shift_imm == 0 then /* LSR #32 */
        index = 0
    else
        index = Rm Logical_Shift_Right shift_imm
0b10 /* ASR */
    if shift_imm == 0 then /* ASR #32 */
        if Rm[31] == 1 then
            index = 0xFFFFFFFF
        else
            index = 0
    else
        index = Rm Arithmetic_Shift_Right shift_imm
0b11 /* ROR or RRX */
    if shift_imm == 0 then /* RRX */
        index = (C Flag Logical_Shift_Left 31) OR
        (Rm Logical_Shift_Right 1)
    else /* ROR */
        index = Rm Rotate_Right shift_imm
endcase

if U == 1 then
    address = Rn + index
else /* U == 0 */
    address = Rn - index
```

- Rn: Base address register
- Rm: Offset register
- LSL: Logical shift left
- LSR: Logical shift right
- ASR: Arithmetic shift right
- ROR: Rotate right
- RRX: Rotate right with extend
- shift_imm: Shift amount contained in bits 11 down to 7
- U: Encoded in bit 23 in the instruction.

Only one of the above cases will be executed, i.e. only one of the shifts will be dispatched since all the values in the conditions are available immediately to the translator. As a result the logic can be implemented inside the translator. The way the shift is implemented is dependent on the implementation choice(See 4.3.2).

```
index = shifted_Rm
if U == 1 then translator dispatches the following:
    add Rx1, Rn, index
    goto -2
else if U == 0 then translator dispatches the following:
    sub Rx1, Rm, index
    goto -2
```

4.6.4 Immediate pre-indexed

This case is almost the same as the one above 4.6.1 with the exception that the new address can replace the old base address register. The replacement occurs if the predication is true. The operation logic can be seen below:

```
//Depending on the instruction
//the offset can be either offset_8
//or offset_12(See below)

if U == 1 then
    address = Rn + offset
else /* if U == 0 */
    address = Rn - offset
if predication == true then
    Rn = address
```

- Rn: Base address register
- offset_12: Encoded in bits 11 down to 0.
- immedH: Encoded in bits 11 down to 8 in the instruction.
- immedL: Encoded in bits 3 down to 0 in the instruction.
- offset_8: immedH && immedL
- U: Encoded in bit 23 in the instruction.

The predication check and potential address replacement should occur after the memory request has been dispatched. For the predication implementation See 4.3.4.1. The emulation procedure can be seen below:

```

if U == 1 then translator dispatches the following:
    add Rx1, Rn, offset
    goto -2
else if U == 0 then translator dispatches the following:
    sub Rx1, Rm, offset
    goto -2
if predication == true then
    add Rn, Rx1, R0
```

4.6.5 Register pre-indexed

This case is almost the same as the one above 4.6.2 with the exception that the new address can replace the old base address register. The replacement occurs if the predication is true. The operation logic can be seen below:

```
if U == 1 then
    address = Rn + Rm
else /* if U == 0 */
    address = Rn - Rm
if predication == true then
    Rn = address
```

- Rn: Base address register
- Rm: Offset register
- U: Encoded in bit 23 in the instruction.

The predication check and potential address replacement should occur after the memory request has been dispatched. For the predication implementation See 4.3.4.1. The emulation procedure can be seen below:

```

if U == 1 then translator dispatches the following:
    add Rx1, Rn, Rm
    goto -2
else U == 0 then translator dispatches the following:
    sub Rx1, Rm, Rn
    goto -2

if predication == true then
    add Rn, Rx1, R0
```

4.6.6 Scaled register pre-indexed

This mode generates the address by adding a shifted register value to the base address. Furthermore, if the predication is true the old base address is replaced by the new one. The operation logic can be seen below:

```
case shift of
0b00 /* LSL */
    index = Rm Logical_Shift_Left shift_imm
0b01 /* LSR */
    if shift_imm == 0 then /* LSR #32 */
        index = 0
    else
        index = Rm Logical_Shift_Right shift_imm
0b10 /* ASR */
    if shift_imm == 0 then /* ASR #32 */
        if Rm[31] == 1 then
            index = 0xFFFFFFFF
        else
```

```
14             index = 0
15     else
16         index = Rm Arithmetic_Shift_Right shift_imm
17 0b11 /* ROR or RRX */
18     if shift_imm == 0 then /* RRX */
19         index = (C Flag Logical_Shift_Left 31) OR
20         (Rm Logical_Shift_Right 1)
21     else /* ROR */
22         index = Rm Rotate_Right shift_imm
23 endcase
24
25 if U == 1 then
26     address = Rn + index
27 else /* U == 0 */
28     address = Rn - index
29 if predication == true then
30     Rn = address
```

- Rn: Base address register
- Rm: Offset register
- LSL: Logical shift left
- LSR: Logical shift right
- ASR: Arithmetic shift right
- ROR: Rotate right
- RRX: Rotate right with extend
- shift_imm: Shift amount contained in bits 11 down to 7
- U: Encoded in bit 23 of the instruction

Only one of the above cases will be executed, i.e. only one of the shifts will be dispatched since all the values in the conditions are available immediately to the translator. As a result the logic can be implemented inside the translator. The way the shift is implemented is dependent on the implementation choice(See 4.3.2). For the predication implementation See 4.3.4.1. The emulation procedure can be seen below:

```
1 index = shifted_Rm
2 if U == 1 then translator dispatches the following:
3     add Rx1, Rn, index
4     goto -2
5 else U == 0 then translator dispatches the following:
6     sub Rx1, Rm, index
7     goto -2
8
9 if predication == true then
10     add Rn, Rx1, R0
```

4.6.7 Immediate post-indexed

This case uses only the base address to access the memory. Next the base address can be updated or not with the new calculated address. This mode differs from the pre-indexed modes in two ways: one only the base address is used without taking into account the offset, and two the potential replacement of the old base register value takes place after the memory address is dispatched.

```
//Depending on the instruction
//the offset can be either offset_8
//or offset_12(See below)

address = Rn
if predication == true then
    if U == 1 then
        Rn = Rn + offset
    else /* U == 0 */
        Rn = Rn - offset
```

- Rn: Base address register
- offset_12: Encoded in bits 11 down to 0.
- immedH: Encoded in bits 11 down to 8 in the instruction.
- immedL: Encoded in bits 3 down to 0 in the instruction.
- offset_8: immedH && immedL
- U: Encoded in bit 23 in the instruction.

The predication check and potential address replacement should occur after the memory request has been dispatched. For the predication implementation See 4.3.4.1. The emulation procedure can be seen below:

```

if U == 1 then translator dispatches the following:
    add Rx1, Rn, offset
    goto -2
else if U == 0 then translator dispatches the following:
    sub Rx1, Rm, offset
    goto -2
if predication == true then
    add Rn, Rx1, R0
```

4.6.8 Register post-indexed

This is the same as the previous case, except instead of immediate offset a register value is used.

```
address = Rn
if predication == true then
    if U == 1 then
```

```
4         Rn = Rn + Rm
5     else /* U == 0 */
6         Rn = Rn - Rm
```

- Rn: Base address register
- Rm: Offset register
- U: Encoded in bit 23 in the instruction.

The predication check and potential address replacement should occur after the memory request has been dispatched. For the predication implementation See 4.3.4.1. The emulation procedure can be seen below:

```
1
2 if U == 1 then translator dispatches the following:
3     add Rx1, Rn, Rm
4     goto -2
5 else if U == 0 then translator dispatches the following:
6     sub Rx1, Rm, Rn
7     goto -2
8 if predication == true then
9     add Rn, Rx1, R0
```

4.6.9 Scaled register post-indexed

This mode uses the base register address as memory address. Next depending on the predication the old base register might be replaced with the new value which is the shifted register value added or subtracted to the old base address.

```
1 address = Rn
2 case shift of
3     0b00 /* LSL */
4         index = Rm Logical_Shift_Left shift_imm
5     0b01 /* LSR */
6         if shift_imm == 0 then /* LSR #32 */
7             index = 0
8         else
9             index = Rm Logical_Shift_Right shift_imm
10    0b10 /* ASR */
11        if shift_imm == 0 then /* ASR #32 */
12            if Rm[31] == 1 then
13            index = 0xFFFFFFFF
14        else
15            index = 0
16        else
17            index = Rm Arithmetic_Shift_Right shift_imm
18    0b11 /* ROR or RRX */
19        if shift_imm == 0 then /* RRX */
20            index = (C Flag Logical_Shift_Left 31) OR
21            (Rm Logical_Shift_Right 1)
22        else /* ROR */
23            index = Rm Rotate_Right shift_imm
```

```
24 endcase
25
26 if predication == true then
27     if U == 1 then
28     Rn = Rn + index
29 else /* U == 0 */
30     Rn = Rn - index
```

- Rn: Base address register
- Rm: Offset register
- LSL: Logical shift left
- LSR: Logical shift right
- ASR: Arithmetic shift right
- ROR: Rotate right
- RRX: Rotate right with extend
- shift_imm: Shift amount contained in bits 11 down to 7
- U: Encoded in bit 23 in the instruction

Only one of the above cases will be executed, i.e. only one of the shifts will be dispatched since all the values in the conditions are available immediately to the translator. As a result the logic can be implemented inside the translator. The way the shift is implemented is dependent on the implementation choice(See 4.3.2). For the predication implementation See 4.3.4.1. The emulation procedure can be seen below:

```
1 index = shifted_Rm
2 if U == 1 then translator dispatches the following:
3     add Rn, Rn, index
4     goto -2
5 else U == 0 then translator dispatches the following:
6     sub Rn, index, Rn
7     goto -2
```

4.6.10 Multiple Load/Stores increment before/after

This addressing mode sets the start_address for the multiple load/store instructions. Load/store instructions access the memory numerous times to get sequential data as is more thoroughly explained in the instruction translation Appendix B. The start_address is calculated and stored in the base register. Subsequent addresses are acquired by incrementing the previous value by 4, starting with the start_address. Furthermore an end_address is also calculated to determine the number of memory accesses. Both the start_address and end_address are calculated initially before the memory access is performed. Depending on the value of bit 24 one of the two procedures is followed:

```
if (bit24 == 0) then//this check can be performed in the translator
    start_address = Rn
    end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4) - 4
    if predication == true and bit21 == 1 then
           Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
else
    start_address = Rn + 4
    end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
    if predication == true and bit21 == 1 then
        Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

- start_address: Determines the address from which memory accesses initialize
- Rn: Contains the value of the base address
- register_list: bits 15 down to 0 of the instruction
- (Number_Of_Set_Bits_In: The number of '1's in the value

The number_of_set_bits_in(register_list) parameter can be always calculated inside the translator since the register_list is available in the instruction field. Aside from that there is also the value of the end_address that needs to be calculated as well as the new Rn value. If the register copy is implemented(See 4.3.2.2) then these values can be calculated within the translator, in the case of emulation the following instructions needs to be dispatched in order to calculate the end_address and the new Rn value:

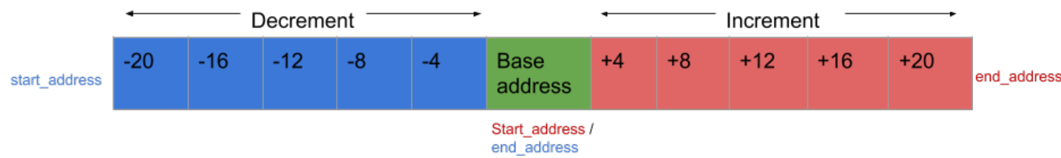
```
add Rx1, Rn imm(number_of_set_bits*4)
sub Rx1, 4, Rx1//Rx1 now contains the end_address
if bit21 == 1//check within the translator
    if predication == true
        add, Rn, Rn, imm(number_of_set_bits*4)
```

Furthermore, in the second case where bit24 == 1, the start_address also needs to be calculated since it is equal to Rn+4 this can be done with a simple "add" instruction.

4.6.11 Multiple Load/Stores decrement before/after

This addressing mode sets the start_address for the multiple load/store instructions. Load/store instructions access the memory numerous times to get sequential data as is more thoroughly explained in the instruction translation Appendix B later on. The start_address is calculated and stored in the base register. Subsequent addresses are acquired by decrementing the previous value by 4, starting with the start_address. Furthermore an end_address is also calculated to determine the number of memory accesses. Both the start_address and end_address are calculated initially before the memory access is performed. The difference with the previous case(increment before/after) is that the start address now is the value of the base address minus all the words that we want to access. Essentially in the first case the base register serves as the start_address and in the second case as the end_address. Figure 4.4 shows this visually. The blue case on the left is the decrement before/after and the red case is the increment before after. In the first case start_address is -20 and end address is the base address, while in the second case the start address is the base address and the end address is +20.

Figure 4.4: Multiple load store, increment vs decrement.



Depending on the value of bit 24 one of the two procedures is followed:

```
if (bit24 == 0) then//this check can be performed in the translator
    start_address =  Rn - (Number_Of_Set_Bits_In(register_list) * 4) + 4
    end_address = Rn
    if predication == true and bit21 == 1 then
           Rn = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
else
    start_address =  Rn - (Number_Of_Set_Bits_In(register_list) * 4)
    end_address =  Rn - 4
    if predication == true and bit21 == 1 then
        Rn = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
```

- start_address: Determines the address from which memory accesses initialize
- Rn: Contains the value of the base address
- register_list: bits 15 down to 0 of the instruction
- (Number_Of_Set_Bits_In: The number of '1's in the value

The number_of_set_bits_in(register_list) parameter can be always calculated inside the translator since the register_list is available in the instruction field. Aside from that there is also the value of the end_address that needs to be calculated as well as the new Rn value. If the register copy is implemented(See 4.3.2.2) then these values can be calculated within the translator, in the case of emulation the following instructions needs to be dispatched in order to calculate the start_address and the new Rn value:

```
add Rx1, Rn, imm(4)
sub Rx1, imm(number_of_set_bits*4), Rx1
if (bit21 == 1) then
    if predication == true then
        sub Rn, imm(number_of_set_bits*4), Rn
```

Furthermore, in the second case where bit24 == 1, the end_address also needs to be calculated since it is equal to Rn-4 this can be done with a simple "sub" instruction.

4.7 Combining all the emulation techniques

So far only seperate procedures of emulation have been presented. To clarify the procedures explained in the previous sections consider an example where the following instruction needs to be completely emulated:

```
ADD R9, R5, R5, LSL #3 ; R9 = R5 + R5 x 8//R5 = 2
```

Furthermore the instruction is set to update the flags. The flags are updated according to the following:

```
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rn + shifter_operand)
V Flag = OverflowFrom(Rn + shifter_operand)
```

As can be derived, the calculation is $2 + 2 * 2^3 = 18$. As a result the N flag will be 0, the Z flag will be 0, the C flag will be 0 and the overflow will also be 0. The emulation stage consists of the following four stages listed in temporal order:

- Predication
- Second operand calculation
- Execution
- Flag update

In order to translate a single instruction that performs all actions(predication and flag updating included), the translator should dispatch instructions that emulate all the above stages in that order. Table 4.5 shows all the ρ-VEX emulation instructions required to execute the above ARM instruction. The table is split into 4 colors: Red indicates the predication emulation (recall 4.3.4.2), blue indicates the second operand calculation (recall 4.4), green is the instruction execution and the yellow shades are the N, Z, C, V flags calculation (recall 4.4). The rest uncolored instructions are used to combine the flag results and update the old flag register as well as resume proper program flow.

As can be seen an overhead of 35 instructions is required to emulate a single ARM instruction. The extra cycles however can be less since many steps can be combined in a single bundle. This however was not examined properly due to time restrictions, however it is something that should definitely be considered during hardware implementation, since dynamic bundle length and instruction allocation is the main advantage that the hardware translator has over a software one. As can be seen on table 4.6 the total resulting overhead for all stages of execution being emulated is roughly 2800% for all the testbench programs.

Table 4.5: Complete instruction emulation example. The instruction to be executed is a simple Add with a shift incorporated and flag update. Red segment is the predication, blue segment is the second operand calculation, green part is the instruction execution and yellow shades show the calculation of the N, Z, C, V flags. The rest of the instructions update the flag register and resume normal program flow.

T	PC value	Instruction	Comments
T0	PC	cmpeq Bd, Rf, imm	Bd: branch register, Rf: flag register, imm: instruction flags
T1	PC+1	NOP	wait for calculation to finish
T2	PC+2	NOP	wait for calculation to finish
T2	PC+3	NOP	wait for calculation to finish
T3	PC+4	goto -3	jump to PC
T4	PC	branch 0	branch to PC or ignore and continue to PC+
T5	PC(branch taken)	shl RX0, R5, 3	shift left R5 by 3 and store in an unused register(RX0)
T6	PC+1	NOP	wait for calculation to finish
T7	PC+2	NOP	wait for calculation to finish
T8	PC+3	NOP	wait for calculation to finish
T9	PC+4	goto -4	jump back to original PC
T10	PC	add R9, R5, RX0	execute instruction
T11	PC+1	NOP	wait for calculation to finish
T12	PC+2	NOP	wait for calculation to finish
T13	PC+3	NOP	wait for calculation to finish
T14	PC+4	tbit Ra4, R9, imm(31)	copy R9(31) to Ra4
T14	PC+5	cmpeq Ra3, R9, imm(0)	compare R9 to 0
T14	PC+6	tbit Rb1, R0, 0	set branch register Rb1 to 0
T15	PC+7	addcg RX, R5, R5	perform addcg in order to acquire the overflow in the branch reg
T16	PC+8	goto -8	jump backwards to inform the translator of an incoming branch
T17	PC	br Rb1, -1	Will not branch back to PC since addcg did not generate a carry, thus Rb1 = 0
T18	PC+1(branch not taken)	or Ra2, R0, imm(0)	put a 0 in Ra2 to indicate that there was not a carry out
T19	PC+2	cmpgt Ra4, R9, imm(0)	check if first operand positive
T19	PC+3	cmpgt Ra3, R9, imm(0)	check if second operand positive
T20	PC+4	cmpeq Rb2, Ra3, Ra4	If both positive set a branch register to one
T21	PC+5	goto -5	jump backwards to inform the translator of an incoming branch
T22	PC	br Rb2, -1	Will branch back to PC to inform the translator know that both are positive
T23	PC(branch taken)	cmplt Ra1, R9, imm(0)	Check if result is negative. This provides the overflow result
T23	PC+1	NOP	wait for result
T24	PC+2	NOP	wait for result
T25	PC+3	NOP	wait for result
T26	PC+4	shl a2, a2, 1	Place the C flag in position 1
T27	PC+5	shl a3, a3, 2	Place the Z flag in position 2
T28	PC+6	shl a4, a4, 3	Place the N flag in position 3
T29	PC+7	NOP	wait for result
T30	PC+8	NOP	wait for result
T31	PC+9	NOP	wait for result
T32	PC+10	or b1, a1, a2	OR the two flag registers in positions 0,1
T33	PC+11	or b2, a3, a4	OR the two flag registers in positions 2,3
T34	PC+12	or b3, b1, b2	OR the two registers that contain the flags NZ and CV
T35	PC+13	OR Fr, b3, R0	Replace the old flag values in Fr with the new ones
T36	PC+14	goto -14	Jump back to normal program flow

Table 4.6: Total overhead resulting from emulating all the steps of executing the instructions. These include the predication calculation, the second operand calculation, the instruction calculation, the flag updating and the program flow resumption.

Program	ARM total instructions	Overhead
bcnt	7171	208697 +2910%
blit	56451	1584400 +2806%
compress	198448	5755041 +2900%
convolution	1330596	38351665 +2882%
crc	53697	1576556 +2936%
des	131401	3854285 +2933%
engine	1120660	32051430 +2860%
g3fax	2552428	72117457 +2825%
jpeg	8929392	258426134 +2894%
pocsag	90407	2568755 +2841%
ucbqsort	466403	13495884 +2893%
v42	4597173	131529395 +2861%

5 Simulation results

This chapter provides the results of the simulations run on the “Powerstone” set of benchmark programs for both RISC-V implementation, as well as ARM. These programs were executed on the un-modified ρ-VEX simulator along with the modified version with the binary translator. This was done in order to compare the results and also verify the program execution flow.

5.1 RISC-V execution metrics

As mentioned earlier, the Powerstone testbench programs were used to develop, debug and verify the proper functionality of the simulator. These programs perform various operations and in the end compare the execution result to the already known expected result. For monitoring and debugging purposes two functionalities have been added to the simulator, one that prints the executed instructions and one that prints metrics about the execution (number of each instruction executed, total cycles, total instructions, overhead).

In order to be able to extract performance and metrics for design choices, all instructions are counted individually. This allows for a very early and rough design exploration which can potentially hint for a more beneficiary approach when proceeding into a hardware implementation. More specifically two scenarios are presented for comparison with the original RISC-V execution pattern. One implementation with minimization of extra hardware and ρ-VEX alteration, and one that tries to approach the optimum solution between extra hardware and instruction overhead. There are solutions that provide a trade off between extra hardware and instruction overhead and more specifically the various options mentioned in Section 3.4.3. Furthermore the original instruction count is also monitored.

After executing all the testbench programs with the three different aforementioned designs, the following results were gathered: Table 5.1 shows the instruction count results for the Powerstone programs. Two scenarios are shown along with the original instruction count. The second column is the original instruction count generated by the RISC-V compiler, the third column is the implementation with minimum hardware overhead in mind, and the fourth one shows the overhead generated by the provided implementation.

As can be seen in the Table 5.1, the resulting instruction overhead can be quite significant, up to 177% with an average of 72% if we consider the minimum hardware approach. On the other side of the spectrum, if we would require no instruction overhead, core modifications and extra hardware would be required and more specifically: driving the current PC to the translator, an extra full adder and an extra comparison unit, both embedded in the translator. The provided implementation is a midway solution. Most of the overhead comes from PC access request (overhead of 7 from roughly 15%

Table 5.1: Execution metrics. First column shows the name of the testbench, second shows the original RISC-V instruction count, third column shows the ρ-VEX implementation instruction count and overhead with hardware minimization in mind. Fourth and final column shows the overhead of the provided implementation.

operation	Original	Minimum Hardware	Provided Implementation
bcnt.c	6832	10118 +48%	7509 +10%
blit.c	55631	70864 +27%	58305 +4%
compress.c	189325	295863 +56%	210689 +11%
crc.c	53198	85608 +60%	60229 +13%
convolution.c	3121461	6951324 +122%	3871525 +24%
engine.c	2002237	5546841 +177%	2641202 +31%
des.c	177961	193039 +8%	181425 +1%
g3fax.c	2296503	3344012 +45%	2500954 +2%
jpeg.c	25438372	63040439 +147%	32097961 +2%
ucbqsort.c	418380	712274 +70%	496245 +7%
v42.c	4364875	6740390 +54%	4895597 +4%
pocsag.c	82379	125474 +52%	91386 +3%

of all instructions), while at the same time the workaround is a simple routing of the current PC to the translator or a PC pause signal. As a result, this design choice was implemented in the simulator which results in an average of 9% overhead.

5.2 ARM execution metrics

The ARM-ρ-VEX translator is provided with two implementation choices in mind: 1) All instructions are entirely emulated, 2) Some translation procedures are incorporated within the translator. Both of these techniques have advantages and disadvantages. On one hand, the first solution is the simplest and does not require any modifications to the ρ-VEX, however, it generates a significant overhead as will be shown here. On the other hand, the second solution generates a relatively small overhead but is more complex to develop and requires modifications on ρ-VEX.

The main metric that is used to calculate the performance of each technique is the generated instruction overhead. Overhead is the extra instructions required to translate one ρ-VEX instruction to ARM. For each of the solutions to the ARM-ρ-VEX incompatibilities presented in Chapter 4, a measurement of the overhead is calculated for all the testbench programs. Similarly to the RISC-V case the “Powerstone” set of testbench programs is utilized.

5.2.1 Shifting mechanism overhead

The first cases that are examined are the two proposed solutions in Chapters 4.3.2.1 and 4.3.2.2 for the shifting mechanism. The first case is the instruction emulation, and the overhead that results for each of the *powerstone* programs can be seen on table 5.2. The first column of the table has the name of the *powerstone* program; the second column

is the base program instruction count; the third column is the overhead resulting from emulating every instruction; the fourth column is the resulting overhead with a modified shifter that allows rotations. The values shown, is the extra instruction count that is generated by the translation procedure. For example, the program “bcnt.o” as seen on Table 5.2, has a base instruction count of 7171. When we chose to pass this program through the translator the generated program has 30746 more instructions than the base one. By adding them up, we can acquire the transalted program instruction count.

Table 5.2: Shifting subroutine overhead

program	Original total cycles	Shift overhead(emulated rotations)	Shift overhead(hardware rotations)
bcnt.c	7171	30746 +428%	28240 +393%
blit.c	56451	175707 +311%	173105 +306%
compress.c	198448	796322 +401%	737700 +371%
crc.c	53697	379327 +706%	335775 +625%
convolution.c	1330596	5093472 +382%	5091010 +382%
engine.c	1120660	5386158 +480%	4746480 +423%
des.c	131401	1132963 +862%	746625 +568%
g3fax.c	2552428	7895246 +309%	6952880 +272%
jpeg.c	8929392	36165434 +405%	32540680 +364%
ucbqsort.c	466403	1143988 +245%	935250 +200%
v42.c	4597173	20901133 +454%	18066955 +393%
pocsag.c	90407	328177 +362%	323835 +358%

The second case is the register live copy solution described in Chapter 4.3.2.2. This solution produces significantly smaller overhead compared to the emulation technique shown above, however, it requires a significant amount of functionality to be embedded within the translator. The overhead can be seen in Table 5.3.

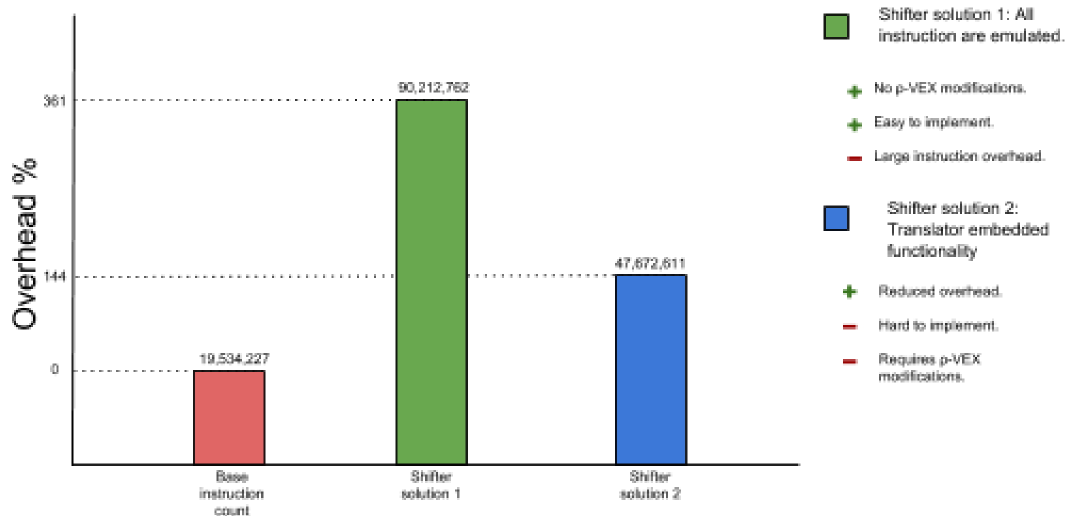
Table 5.3: Register duplicate instruction overhead.

Program	Original total cycles	Instruction overhead
bcnt	7171	5236 +73%
blit	56451	42788 +75%
compress	198448	140844 +70%
convolution	1330596	1407868 +105%
crc	53697	133712 +249%
des	131401	70744 +53%
engine	1120660	1186320 +105%
g3fax	2552428	3269316 +128%
jpeg	8929392	15078528 +168%
pocsag	90407	114488 +126%
ucbqsort	466403	300104 +64%
v42	4597173	6388436 +138%

Overall, the two solutions provide two design choices that allow for a tailor-made development. If simplicity is required, then the first solution should be followed since the translation procedure does not require any -VEX core alterations nor complex translator development techniques. If on the other hand, performance is required, then the second

solution offers significantly reduced overhead, with the price payed being development complexity and -VEX alterations. Figure 5.1 shows a summary of the proposed solutions.

Figure 5.1: Comparison of the proposed shifting mechanics implementations.



5.2.2 PC write overhead

Next, we examine the overhead that results when an instruction tries to write the PC as described in 4.3.3.2. A table with all the overheads of the Powerstone testbench programs that result from the PC access translation procedure, is shown in Table 5.4. As can be seen on the table the overheads are negligible compared to the ones introduced by other emulation procedures such as the shifting emulation.

5.2.3 Predication system overhead

In order to incorporate the predication system of ARM in ρ-VEX, extra steps in the translation procedure are required. The predication system consists of two parts 4.3.4, the flag comparison and the flag update. Similarly to other cases, two solutions are provided for both the flag comparison and the flag update procedures: 1) Instruction emulation, 2) translation implementation.

5.2.3.1 Flag comparison, emulation overhead.

First, the instruction emulation overhead is calculated. Table 5.5 shows the resulting overhead for the powerstone testbench, that results from the emulation procedure.

As can be seen the emulation procedure for the flag comparison, is not efficient with overhead being over 500%. This is because for every instruction translated, 4 flag bits need to be compared while at the same time maintaining the correct program flow.

Table 5.4: PC target instruction overhead. This table shows the overhead that results from instructions that target PC as destination register.

Program	Original total cycles	Instruction overhead
bcnt	7171	488 +6%
blit	56451	496 +1%
compress	198448	4228 +2%
convolution	1330596	45128 +3%
crc	53697	1520 +2%
des	131401	1052 +1%
engine	1120660	49268 +4%
g3fax	2552428	58108 +2%
jpeg	8929392	125400 +1%
pocsag	90407	2672 +3%
ucbqsort	466403	93716 +20%
v42	4597173	174224 +3%

Table 5.5: Flag comparison overhead. This table shows the overhead by emulating the flag comparison.

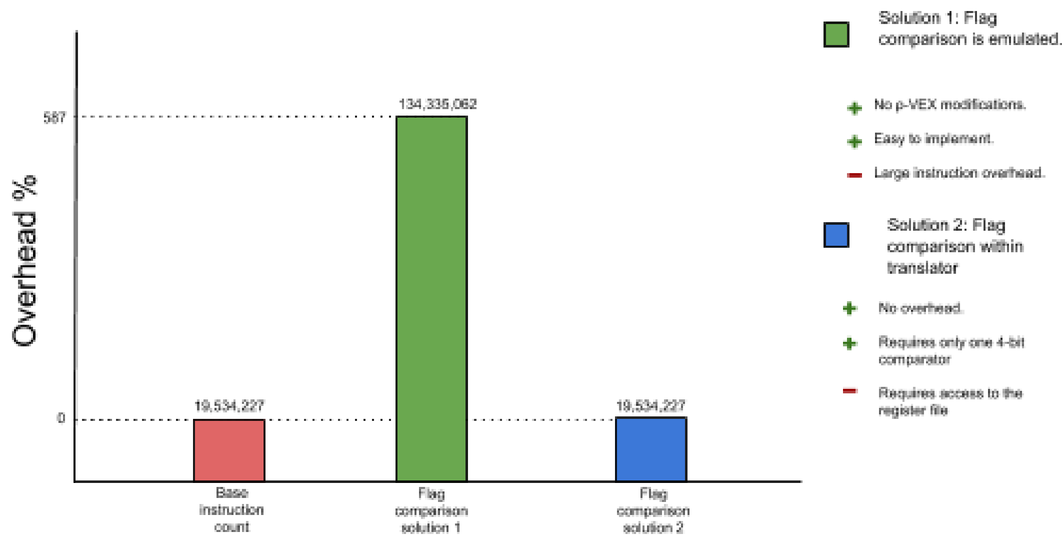
Program	ARM total instructions	Flag comparison overhead
bcnt	7171	35525 +495%
blit	56451	281920 +499%
compress	198448	1171716 +492%
convolution	1330596	7614774 +476%
crc	53697	314118 +487%
des	131401	785466 +498%
engine	1120660	5959332 +443%
g3fax	2552428	15197064 +496%
jpeg	8929392	53435178 +498%
pocsag	90407	531648 +490%
ucbqsort	466403	2737596 +489%
v42	4597173	26736498 +484%

As was shown in Chapter 4.3.4.1, this can be particularly complicated. However, if this functionality is embedded within the translator, the overhead is essentially zero. However, the translator needs to have access to the register file via the solution. A summary of the design choices can be seen here 5.2.

5.2.4 Flag comparison, translator overhead

As mentioned above, the overhead that results from implementing the flag comparison system within the translator is zero. This is because the comparison operates on-the-fly and produces a valid/not valid instruction signal immediately.

Figure 5.2: Comparison of the proposed flag comparison techniques.



5.2.5 Flag update, emulation overhead.

Next, the flag update system is examined. When an instruction is executed and at the same time it is set to update the flags, then the system must update the old values with the new ones and at the same time make sure to maintain correct program flow. First, we examine the emulation technique.

For every instruction that updates the flags there is an average of 22 instructions overhead. This is because the flag calculation is not straight forward and most of the required functionality does not exist or is not supported by ρ-VEX (see Chapter 4.3.4.1). Table 5.6 shows the resulting overhead for the Powerstone testbench. As can be seen the overhead is an average of 2400%, which is significant. As a matter of fact it is the major overhead source of all the translation procedure, assuming that everything is emulated. For this reason this was not chosen in the provided implementation and also is not advised for hardware implementation. Instead, the flag predication system should be incorporated inside the translator.

5.2.6 Flag update, translator implementation overhead

Having the translator store locally the flag values reduces the overhead significantly. However, it still is not equal to zero. This is because the translator needs to wait for the results of the flags. This procedure introduces an overhead of four (see Section 4.1.1.1) whenever an instruction is set to update the flags. The resulting overhead can be seen in Table 5.7. As can be seen the overhead compared to the emulation technique is significantly smaller especially when it is combined with the zero overhead from the flag comparison step. The required modifications in the translator can be seen in figure 4.2 with blue color.

Table 5.6: Flag updating overhead. This table shows the overhead by the emulation of updating the instruction flags.

Program	ARM total instructions	Flag updating overhead
bcnt	7171	173172 +2414%
blit	56451	1302480 +2307%
compress	198448	4778611 +2407%
convolution	1330596	32006020 +2405%
crc	53697	1314791 +2448%
des	131401	3199730 +2435%
engine	1120660	27085320 +2416%
g3fax	2552428	59453237 +2329%
jpeg	8929392	213896819 +2395%
pocsag	90407	2125715 +2351%
ucbqsort	466403	11214554 +2404%
v42	4597173	109248980 +2376%

Table 5.7: Flag calculation overhead. This table shows the overhead resulting by calculating the flags locally in the translator.

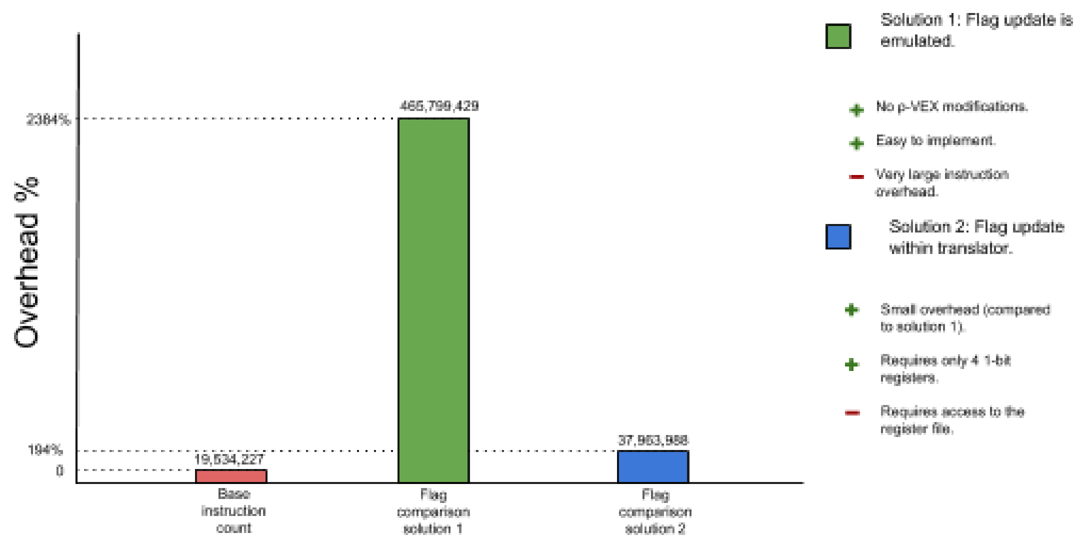
Program	ARM total instructions	Flag calculation overhead
bcnt	7171	14472 +201%
blit	56451	120012 +212%
compress	198448	559492 +281%
convolution	1330596	1844064 +138%
crc	53697	102576 +191%
des	131401	196660 +149%
engine	1120660	3139240 +280%
g3fax	2552428	6028672 +236%
jpeg	8929392	13710184 +153%
pocsag	90407	238292 +263%
ucbqsort	466403	1054048 +225%
v42	4597173	10956276 +238%

A summary of the two flag update techniques are depicted in Figure 5.3.

5.2.7 Combination of all the translation techniques

In order to provide a high-level picture of the translation techniques all of the translation procedure overheads are combined. This is done with the two extreme approaches: minimum hardware and minimum overhead. Overall, the minimum hardware approach results in an overhead of 1026% and the minimum overhead approach results in an overhead of 12%

Figure 5.3: Comparison of the proposed flag update techniques.



6 Future work and conclusions

The translation procedure of two ISAs has been presented, RISC-V to ρ-VEX and ARMv4 to ρ-VEX. The translation technique implemented was basic and straightforward, since whenever it was possible, a simple direct one-to-one instruction per cycle translation was performed. Both of the ISAs that were translated have compilers that generate code for serial execution and are not meant for a VLIW processor such as ρ-VEX. As a result the translator generates in most cases one syllable per word per cycle. As can be understood, this is highly inefficient since, assuming that no other contexts operate on the core, most of the resources are not utilized. This was a conscious decision during development since trying to implement a more efficient system would be off the time limits of this thesis. However, a series of potential improvements are proposed and briefly explained, which can be implemented in order to improve the efficiency and execution of the translated code.

6.1 Conclusions

Overall, many binary translation techniques exist and depending on the application, available development time and platform, different types can be used. Software binary translators are easy to implement and easy to port in different machines running on the same OS. However, they do not offer the optimum performance compared to hardware.

Another way to distinguish translators, is the type: dynamic or static. Static translators operate offline by processing a compiled binary code. However, due to mainly low visibility of the execution state of the program, they are hard to develop and often require human input. On the other hand, dynamic translators operate during execution time. As a result, they have a complete picture of the execution flow. The problem with dynamic translators is the fact that there are time limitations since they need to keep up to the program execution time, otherwise performance will become a major issue. In this thesis, the translator is designed as a hardware dynamic one in order to take advantage of the hardware performance, and the development ease of the dynamic translators.

Chapter 3, presents the design of a hardware binary translator from RISC-V ISA to ρ-VEX ISA has been presented in the third chapter. The implementation is provided in a software simulator that mirrors the actual hardware ρ-VEX core. A brief overview of the RISC-V architecture was presented as well as the major incompatibilities between the two architectures. Furthermore, it is shown that all instructions can be either directly translated, emulated or partially executed by extra hardware; each of the aforementioned approaches results in different advantages and disadvantages. More solutions to these incompatibilities are proposed with different optimizations in mind. Proper functionality was verified with the Powerstone testbench and execution metrics were also provided to assist with future development. Further work can be put in improving the transla-

tion procedure and especially the extent to which the full capabilities of ρ-VEX can be utilized.

The development process for the RISC-V-ρ-VEX translator lasted for approximately 2 months and is composed of roughly 1000 lines of C code. “Spike” [28] simulator was used to run RISC-V binaries for comparison and debugging purposes.

The fourth chapter 4, presented the implementation of a binary translator from ARMv4 architecture to ρ-VEX. The work was implemented in the software simulator that emulates the ρ-VEX functionality. Since almost no ARM instruction can be directly translated to ρ-VEX ones, mainly due to predication, the required functionality can be implemented with either extra hardware and core modifications, or purely by instruction emulation. The instruction emulation, even though it does not require any core modifications, it introduces a very large instruction overhead (up to 35 instructions overhead for a single ARM instruction). Furthermore the control logic required for the emulation approach is significantly more complex, since it requires logic to maintain program flow for every instruction. On the other hand, the second proposed, implementation incorporates extra hardware inside the translator to execute the required additional functionalities, e.g. shifter and predication system. The main disadvantage of the second implementation technique is that in many cases it requires extra hardware and some core modifications, mainly signals and values that need to be driven to the translator. Furthermore, some of the signals fed into the translator can reside inside the critical path which might impact the core performance.

As mentioned before, the main implementation technique used in the simulator was to incorporate any possible functionality to the translator in order to avoid large overheads. Furthermore, it allowed for a much simpler implementation. Regardless, the development of the simulator for the ARM translation was particularly complex, due to debugging difficulties since there is no accurate way to monitor the executed code program flow. The development process for the ARM-ρ-VEX translator lasted 4 months and it consists of roughly 4500 lines of C code.

Chapter 5, presents the simulation results for both of the translators. Generally for both implementations, two approaches were presented. The two general approaches, emulation and translator implementation, can vary significantly; both of the solutions have advantages and disadvantages. On one hand, the emulation technique provides a simple to implement methodology and does not require and modifications to the existing ρ-VEX core which might result in performance degradation. On the other hand however, it also introduces a significant overhead which can result in orders of magnitude more instructions. Contrary to that, by implementing a part of the translation functionality within the translator, the resulting overhead is significantly smaller and in some cases zero. However, this procedure requires modifications of ρ-VEX core. This can prove to be hard in terms of implementation and also increases the risk of reducing the performance of the core. The design choice should be made after taking into account all of the factors presented in this thesis.

6.2 Main contribution

The intent of this thesis is to establish the necessary base for a future hardware development. Two hardware binary translators, RISC-V and ARMv4 to ρ-VEX, have been presented. The implementation was performed on the ρ-VEX simulator. As a general approach two implementations choices are shown for each of the RISC-V and ARM translators, one that tries to hide any translation differences by implementing them within the translator, and one that utilizes only existing ρ-VEX instructions in order to emulate the functionality of the instructions that cannot be directly translated. Both have advantages and disadvantages; on one hand the approach that tries to hide the differences, requires more hardware and in a few cases requires modifications to the core, however it introduces little to no instruction overhead. On the other hand, the emulation approach does not require any core modifications, however the instruction overhead introduced is large often up to 2000%.

Recalling the research question:
"How can we construct a dynamic binary translator that allows ρ-VEX to execute other ISAs on the fly?"
The solution provided are two dynamic binary translators, one for RISC-V and one for ARM both translating into ρ-VEX ISA. Both are provided with two design choices in mind allowing for for execution on-the-fly, with, either performance in mind, or hardware minimization. The main contributions of this thesis are listed below:

- RISC-V to ρ-VEX ISA dynamic binary translator design.
- ARM to ρ-VEX dynamic binary translator design.
- Design choices aimed at maximizing performance or minimizing hardware.
- ρ-VEX simulator able to execute ARM/RISC-V binaries and simulate the translation procedure.
- Testbench programs execution metrics for many design choices.
- Brief overview of binary translation techniques.

6.3 Future work

Below follows some proposals for future work based on this thesis. These projects will enhance the performance of the translator as well as the ρ-VEX core itself.

6.3.1 Word Filling

The first relatively simple improvement, is to fill the word with syllables coming from the translation of a single instruction. For example a fully emulated ARM instruction can generate up to 35 ρ-VEX instructions. In the provided ARM translator implementation there is no system that handles the spreading of these instructions in different ρ-VEX syllables. Note that this system is not required in the RISC-V because in the few cases

that there is translation to more than one instructions, these instructions are dependent to each other.

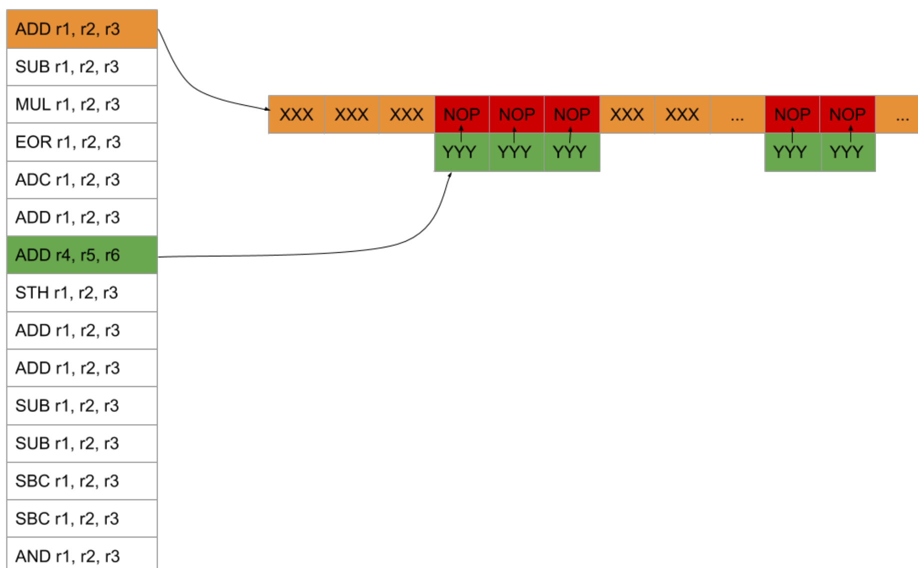
The first requirement for such a system is to be able to recognize the current available ρ-VEX pipelanes of the translator. This information should be driven by ρ-VEX itself to the translator. As an example consider that the current configuration and context spreading provide to the translator two pipelanes. As a result the maximum parallelization of the generated instructions would be two per cycle.

In many cases though the translator generates NOPs while waiting for results to become available in the pipeline. This is highly inefficient since as can be seen in Table 5.3 the NOPs can be up to 35% of the total translation instructions.

6.3.2 Instruction buffer

A solution to the above problem is to replace the NOP instructions with other independent instructions. The tricky part in this is to maintain an appropriately sized monitoring buffer, i.e., providing a window of instructions to the translator to pick for translation and dispatching, while at the same time making sure that the buffer does not affect performance. Figure 6.1 a rather naive example is shown. Consider a 15 instruction window buffer from which the translator can pick instructions for translation. As can be seen only one instruction (green) is not dependent on registers r1, r2 and r3. Assuming that the first instruction is translated (orange) when NOPs are encountered the translator can chose another independent instruction from the window buffer to start translating and replace the NOPs with these instructions. Any instruction that is completely translated should be replaced with the next instruction outside the window.

Figure 6.1: Mixing instructions to fill NOPs generated by the translator.



However, if the new instruction that replaces the old one is a branch instruction, then the updating should halt until all instructions in the window buffer are executed. Another issue that will arise with this technique is the synchronization of the internal translator PC with the actual program PC. In the provided emulation procedure, the PC synchronizes every time after every translation step (predication, second operand calculation etc). This allows for a simple translator logic implementation. However, the translator now needs to keep track of the parallel PC values of different instructions that are executed at the same time. A simple solution to this is to attach labels to the instructions along with their original PCs. This way the translation has a complete overview of the translation state.

6.3.3 Out-of-order execution

Another possible improvement is to implement an out-of-order-execution system in order to extract as much ILP as possible from the generated code. Essentially the basic ideas that can be implemented can be seen in Tomasulo algorithm [29]. When it comes to utilizing a translator, ρ-VEX is an ideal platform to apply out-of-order-execution mainly for two reasons: One, ρ-VEX contains 63 general-purpose registers, most of which remain unutilized for a translation like ARM where only 16 registers are required. As a result these registers can be used for temporary placeholders when register renaming is required. Also, the VLIW nature of ρ-VEX and the multiple available execution resources, can be used in this case as a superscalar architecture where the instruction scheduling task falls upon the translator.

6.3.4 Resource scheduler

A good improvement which can be regarded as an addition to ρ-VEX, more than to the translator, is to create a system that monitors the utilization of the provided pipelanes to each context. For example if 4 pipelanes are provided to the translator context and the translator dispatches on average 2 syllables per cycle, half of the available pipelanes are not used. A solution to this problem could be a real time feedback from the translator, of how many syllables are utilized on average. This information should be handled by a higher level scheduler that keeps track of all contexts and the distribution of pipelanes among them. For example there could be segments of code where parallelization can be much higher than other segments. In this case the scheduler should assign more pipelanes to the translator context. If on the other hand there are segments that cannot be parallelized the distribution should again change according to the requirements of each context. Essentially the provided feedback should be the average parallelization extraction on a given window of time, which can of course be adjustable.

Bibliography

[1] G. David, "Chip makers turn to multicore processors."

[2] L. van Bremen, "ρ-VEX on chip the design of an ASIC for a dynamically reconfigurable vliw processor with 24-port register file."

[3] "ρ-VEX publications." [Online]. Available: http://rvex.ewi.tudelft.nl/?page_id=104

[4] A. Erik, K. David, and S. Yaron, "Welcome to the opportunities of binary translation."

[5] C. Cristina and M. Vishv, "Binary translation: Static, dynamic, retargetable?*."

[6] S. Richard, C. Anton, K. Matthew, M. Maurice, and R. Scott, "Binary translation."

[7] E. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritts, P. Ledak, D. Appenzeller, C. Agricola, and Z. Filan, "BOA: The architecture of a binary translation processor."

[8] E. Kemal, F. Jason, K. Stephen, G. Michael, A. Erik, K. Krishnan, and B. Terry, "An eight-issue tree-VLIW processor for dynamic binary translation."

[9] A. Erik, E. Kemal, G. Michael, and S. Sumedh, "Advances and future challenges in binary translation and optimization."

[10] C. Jiunn-Yeu, S. Bor-Yeh, O. Quan-Huei, Y. Wuu, and H. Wei-Chung, "Effective code discovery for ARM/thumb mixed ISA binaries in a static binary translator."

[11] P. Mark, "Dynamic binary translation."

[12] H. Urs, "Adaptive optimization for self: Reconciling high performance with exploratory programming."

[13] B. Fabrice, "QEMU, a fast and portable dynamic translator."

[14] G. Michael, A. Erik R., S. Sumedh, L. Paul, and D. Appenzeller, "Dynamic and transparent binary translation."

[15] H. Weiwu, L. Qi, W. Jian, C. Songsong, S. Menghao, and L. Xiaoyu, "Efficient binary translation system with low hardware cost."

[16] A. Klaiber, "The technology behind CRUSOE processors."

[17] V. Bala, E. Duesterwald, and S. Banerjia, "Transparent dynamic optimization: The design and implementation of DYNAMO."

[18] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "FX!32A profile-directed binary translator."

[19] E. Kemal, A. Erik, G. Michael, and S. Sumedh, "Dynamic binary translation and optimization."

[20] R. Simon, R. Erven, and D. Steven, "Hardware-accelerated dynamic binary translation."

[21] "QEMU emulator ISAs." [Online]. Available: https://qemu.weilnetz.de/doc/qemu-doc.html

[22] D. Patterson and J. Hennessy, "Computer organization and design MIPS edition: The hardware/software interface."

[23] J. Fisher, P. Faraboschi, and C. Young, "Embedded computing: a VLIW approach to architecture, compilers and tools."

[24] "RISC-V members." [Online]. Available: https://riscv.org/members-at-a-glance/

[25] "RISC-V instruction manual." [Online]. Available: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

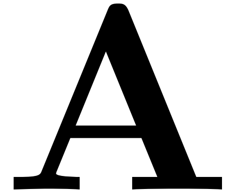
[26] "ARM microarchitectures list." [Online]. Available: https://en.wikipedia.org/wiki/List_of_ARM_microarchitectures

[27] P. Gerald J and G. Robert P, "Formal requirements for virtualizable third generation architectures."

[28] "RISC-V simulator." [Online]. Available: https://riscv.org/software-tools/risc-v-isa-simulator/

[29] R. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development.*

Appendix: RISC-V Instruction Translation



A.1 add

This instruction adds the contents of two registers and stores the result in the target register and overflow is ignored. This instruction is directly translated to the ρ-VEX instruction “add”.

```
1 rd = rs1 + rs2
```

opcode = “0110011”, funct7 = “0000000”, funct3 = “000”

A.2 sub

This instruction performs subtraction between the contents of two registers and stores the result in the target register. This instruction can be translated directly to the ρ-VEX “sub” instruction. Attention must be taken in the order of subtraction, RISC-V subtraction is rs1-rs2 and not rs2-rs1 as in ρ-VEX.

```
1 rd = rs1 - rs2
```

opcode = “0110011”, funct7 = “0100000”, funct3 = “000”

A.3 sll

This instruction performs logical left shift, i.e. performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2. Zeros are pushed in and the result is stored in rd. This instruction can be directly translated to the ρ-VEX “shl” instruction.

```
1 rd = rs1 << rs2(4 downto 0)
```

opcode = “0110011”, funct7 = “0000000”, funct3 = “001”

A.4 slt

This instruction performs signed comparison between rs1 and rs2, if $rs1 < rs2$ then $rd < -'1'$. This instruction can be directly translated to the ρ-VEX instruction “cmplt”.

```
1 if (signed(rs1) < signed(rs2))
2 {
3   rd = 1
4 }
5 else
6 {
```

```
  rd = 0
}
```

opcode = “0110011”, funct7 = “0000000” funct3 = “010”

A.5 sltu

This instruction performs unsigned comparison between rs1 and rs2, if $rs1 < rs2$ then $rd <-'1'$. This instruction can be directly translated to the ρ-VEX instruction “cmplt”.

```
if (unsigned(rs1) < unsigned(rs2))
{
  rd = 1
}
else
{
  rd = 0
}
```

opcode = “0110011”, funct7 = “0000000” funct3 = “011”

A.6 xor

This instruction performs logical xor between rs1 and rs2 and stores the result in rd. This instruction can be directly translated to the ρ-VEX instruction “xor”.

```
rd = rs1 xor rs2
```

opcode = “0110011”, funct7 = “0000000” funct3 = “100”

A.7 srl

This instruction performs logical right shift, i.e. performs logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2. Zeros are pushed in and the result is stored in rd. This instruction can be directly translated to the ρ-VEX “shru” instruction.

```
rd = rs1 >> rs2(4 downto 0)
```

opcode = “0110011”, funct7 = “0000000”, funct3 = “101”

A.8 sra

This instruction performs arithmetic right shift, i.e. performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2. The sign bit is pushed in and the result is stored in rd. This instruction can be directly translated to the ρ-VEX “shr” instruction.

```
rd = rs1 >> rs2(4 downto 0)
```

opcode = “0110011”, funct7 = “0100000”, funct3 = “101”

A.9 or

This instruction performs logical or between rs1 and rs2 and stores the result in rd. This instruction can be directly translated to the ρ-VEX instruction “or”.

```
1 rd = rs1 or rs2
```

opcode = “0110011”, funct7 = “0000000” funct3 = “110”

A.10 and

This instruction performs logical and between rs1 and rs2 and stores the result in rd. This instruction can be directly translated to the ρ-VEX instruction “and”.

```
1 rd = rs1 and rs2
```

opcode = “0110011”, funct7 = “0000000” funct3 = “111”

A.11 addi

This instruction adds the contents of register rs1 to a sign extended immediate value and stores the result to rd. This instruction is directly translated to the ρ-VEX instruction “add”.

```
1 rd = rs1 + sign_ext(imm)
```

opcode = “0010011”, funct3 = “000”

A.12 slli

This instruction performs logical left shift, i.e. performs logical left shift on the value in register rs1 by Imm value. Zeros are pushed in and the result is stored in rd. This instruction can be directly translated to the ρ-VEX “shl” instruction.

```
1 rd = rs1 << imm
```

opcode = “0010011”, funct3 = “001”

A.13 slti

This instruction performs signed comparison between rs1 and sign extended imm, if $rs1 < imm$ then $rd < -`1'$. This instruction can be directly translated to the ρ-VEX instruction “cmplt”.

```
1 if (signed(rs1) < signed(sign\_ext(imm)))
2 {
3   rd = 1
4 }
5 else
6 {
7   rd = 0
8 }
```

opcode = “0010011”, funct3 = “010”

A.14 sltiu

This instruction performs unsigned comparison between rs1 and imm, if $rs1 < imm$ then $rd < -'1'$. This instruction can be directly translated to the ρ-VEX instruction “cmplt”.

```
1 if (unsigned(rs1) < unsigned(imm))
2 {
3   rd = 1
4 }
5 else
6 {
7   rd = 0
8 }
```

opcode = “0010011”, funct3 = “011”

A.15 xori

This instruction performs logical xor between rs1 and sign extended imm and stores the result in rd. This instruction can be directly translated to the ρ-VEX instruction “xor”.

```
1 rd = rs1 xor sign_ext(imm)
```

opcode = “0010011”, funct3 = “100”

A.16 srli

This instruction performs logical right shift, i.e. performs logical right shift on the value in register rs1 by imm amount. Zeros are pushed in and the result is stored in rd. This instruction can be directly translated to the ρ-VEX “shru” instruction.

```
1 rd = rs1 >> imm
```

opcode = “0010011”, funct3 = “101”, funct7 = “0000000”

A.17 srai

This instruction performs arithmetic right shift, i.e. performs arithmetic right shift on the value in register rs1 by the imm amount. The sign bit is pushed in and the result is stored in rd. This instruction can be directly translated to the ρ-VEX “shr” instruction.

```
1 rd = rs1 >> imm
```

opcode = “0010011”, funct3 = “101”, funct7 = “0100000”

A.18 ori

This instruction performs logical or between rs1 and imm and stores the result in rd. This instruction can be directly translated to the ρ-VEX instruction “or”.

```
rd = rs1 or rs2
```

opcode = “0010011”, funct3 = “110”

A.19 andi

This instruction performs logical and between rs1 and imm and stores the result in rd. This instruction can be directly translated to the ρ-VEX instruction “and”.

```
rd = rs1 and rs2
```

opcode = “0010011”, funct3 = “111”

A.20 beq

This instruction performs comparison between registers rs1 and rs2. If they are equal then a branch is performed. This instruction is translated into two ρ-VEX instructions that are executed separately into two cycles, and two different bundles with one instruction each. On the first cycle a “cmpeq” is performed with the result stored in a random branch register (br3 in this implementation). On the second cycle and in a separate bundle a “br” is performed. PC is paused for one cycle.

```
If (rs1 = rs2)
{
  branch;
}
```

opcode = “1100011”, funct3 = “000”.

A.21 bne

This instruction performs comparison between registers rs1 and rs2. If they are not equal then a branch is performed. This instruction is translated into two ρ-VEX instructions that are executed separately into two cycles, and two different bundles with one instruction each. On the first cycle a “cmpne” is performed with the result stored in a random branch register (br3 in this implementation). On the second cycle and in a separate bundle a “br” is performed. PC is paused for one cycle.

```
If (rs1 != rs2)
{
  branch;
}
```

opcode = “1100011”, funct3 = “001”.

A.22 blt

This instruction performs comparison between registers rs1 and rs2. If rs1 is less than rs2 in signed arithmetic then a branch is performed. This instruction is translated into two ρ-VEX instructions that are executed separately into two cycles, and two different bundles with one instruction each. On the first cycle a "cmplt" is performed with the result stored in a random branch register (br3 in this implementation). On the second cycle and in a separate bundle a "br" is performed. PC is paused for one cycle.

```
If (signed(rs1) < signed(rs2))
{
  branch;
}
```

opcode = "1100011", funct3 = "100".

A.23 bge

This instruction performs comparison between registers rs1 and rs2. If rs1 is greater than rs2 in signed arithmetic then a branch is performed. This instruction is translated into two ρ-VEX instructions that are executed separately into two cycles, and two different bundles with one instruction each. On the first cycle a "cmpgtu" is performed with the result stored in a random branch register (br3 in this implementation). On the second cycle and in a separate bundle a "br" is performed. PC is paused for one cycle.

```
If (signed(rs1) > signed(rs2))
{
  branch;
}
```

opcode = "1100011", funct3 = "101".

A.24 bltu

This instruction performs comparison between registers rs1 and rs2. If rs1 is less than rs2 in unsigned arithmetic then a branch is performed. This instruction is translated into two ρ-VEX instructions that are executed separately into two cycles, and two different bundles with one instruction each. On the first cycle a "cmpltu" is performed with the result stored in a random branch register (br3 in this implementation). On the second cycle and in a separate bundle a "br" is performed. PC is paused for one cycle.

```
If (rs1 < rs2)
{
  branch;
}
```

opcode = "1100011", funct3 = "110".

A.25 bgeu

This instruction performs comparison between registers rs1 and rs2. If rs1 is greater than rs2 in unsigned arithmetic then a branch is performed. This instruction is translated into two ρ-VEX instructions that are executed separately into two cycles, and two different bundles with one instruction each. On the first cycle a "cmpgtu" is performed with the result stored in a random branch register (br3 in this implementation). On the second cycle and in a separate bundle a "br" is performed. PC is paused for one cycle.

```
If (rs1 > rs2)
{
  branch;
}
```

opcode = "1100011", funct3 = "111".

A.26 lb

This instruction loads a byte from the memory at the address of rs1 + signed(sign_ext(imm)), sign extends it, and stores it to rd. This instruction is directly translated to the "ldb" ρ-VEX instruction.

```
rd = sign_ext(mem(rs1 + signed(sign_ext(imm))))
```

opcode = "0000011", funct3 = "000"

A.27 lh

This instruction loads a halfword from the memory at the address of rs1 + signed(sign_ext(imm)), sign extends it, and stores it to rd. This instruction is directly translated to the "ldh" ρ-VEX instruction.

```
rd = sign_ext(mem(rs1 + signed(sign_ext(imm))))
```

opcode = "0000011", funct3 = "001"

A.28 lw

This instruction loads a word from the memory at the address of rs1 + signed(sign_ext(imm)), and stores it to rd. This instruction is directly translated to the "ldw" ρ-VEX instruction.

```
rd = mem(rs1 + signed(sign_ext(imm)))
```

opcode = "0000011", funct3 = "010"

A.29 lbu

This instruction loads a byte from the memory at the address of rs1 + signed(sign_ext(imm)), zero extends it, and stores it to rd. This instruction is directly translated to the "ldbu" ρ-VEX instruction.

```
rd = mem(rs1 + signed(sign_ext(imm)))
```

opcode = “0000011”, funct3 = “100”

A.30 lhu

This instruction loads a halfword from the memory at the address of rs1 + signed(sign_ext(imm)), zero extends it, and stores it to rd. This instruction is directly translated to the “ldhu” ρ-VEX instruction.

```
rd = mem(rs1 + signed(sign_ext(imm)))
```

opcode = “0000011”, funct3 = “101”

A.31 sb

This instruction stores a byte from the 8 lower bits of rs1 to the memory address of rs2 + signed(sign_ext(imm)). This instruction is directly translated to the “stb” ρ-VEX instruction.

```
mem(rs2 + signed(sign_ext(imm))) = rs1(7 downto 0)
```

opcode = “0100011”, funct3 = “000”

A.32 sh

This instruction stores a halfowrd from the 16 lower bits of rs1 to the memory address of rs2 + signed(sign_ext(imm)). This instruction is directly translated to the “sth” ρ-VEX instruction.

```
mem(rs2 + signed(sign_ext(imm))) = rs1(15 downto 0)
```

opcode = “0100011”, funct3 = “001”

A.33 sw

This instruction stores the contents of rs1 to the memory address of rs2 + signed(sign_ext(imm)). This instruction is directly translated to the “stw” ρ-VEX instruction.

```
mem(rs2 + signed(sign_ext(imm))) = rs1
```

opcode = “0100011”, funct3 = “010”

A.34 jal

This instruction performs a direct jump and link where the signed immediate value is added to the current PC to obtain the jump address, while at the same time the next PC value PC+1 is stored in the link register. This instruction utilizes the J-type format 3.3.

The software calling convention assigns register "x1" as the link register and "x5" as the alternate link register, see table 3.3. The immediate value is obtained by concatenating the following instruction fields: imm = instr(30) & instr(19 downto 12) & instr(20) & instr(30 downto 21) & '0'

A zero is also concatenated at the end, since the offset value is a multiple of 2 and the actual final '0' of the offset, is not stored. This instruction can be directly translated to a ρ-VEX instruction with a small modification however. Even though there is a similar ρ-VEX instruction the actual jump target in ρ-VEX architecture is calculated as PC + 1 + offset instead of PC + offset. As a result this needs to be dealt with by subtracting a '1' locally from the offset.

```
PC = PC + 1 + sign_ext(signed(imm) - 1)
rs63 = PC + 1
```

opcode = "1101111"

A.35 jalr

This instruction performs an indirect jump and link, where the signed immediate value is added to the value of rs1, while the next PC value PC+1 is stored to the link register. This instruction utilizes the I-type format 3.3. The immediate value is obtained by the following instruction field: instr(31 downto 20). The approach for this instruction is the same as "jal" mentioned above, however instead of subtracting a '1' from the offset locally this needs to be done with emulation in the following manner:

```
sub rs1, 1, rs1
nop
nop
nop
jalr
```

opcode = "1100111"

A.36 lui

This instruction is used to create 32bit constants. It utilizes the U-type format3.3 and places the 20bit immediate value in the top bits of register rd, filling the remaining bits with zeros. This instruction is not translated directly to any ρ-VEX however it can be simply implemented by inserting the value as an immediate and using an instruction like "or" with register r0 (hardwired to zero) to put the value in rd.

```
rd = r0 or imm << 12
```

opcode = "0110111"

A.37 auipc

This instruction is used to create PC relative addresses. It utilizes the U-type format3.3 and creates an immediate value by putting the 20bit immediate value in the higher order

bits of a 32 bit value filling in the rest with zeros. This value is then added to the current PC and then stored to rd. The procedure mentioned here 3.3.4 can be used to acquire the PC value.

```
rd = (signed(imm) << 12) + PC
```

opcode = “0010111”

Appendix: ARM Instruction Translation

B

This appendix describes the translation procedure for all the individual ARM instructions to ρ-VEX ones. As mentioned earlier 4, the ARMv4 subset is used due to its simplicity, compared to other ARM architectures, and highest compatibility with the ρ-VEX architecture. From the translation procedure, a few instructions are omitted and more specifically the ones that are related to execution privileges, system calls and co-processor instructions. Execution privileges are not implemented since ρ-VEX is responsible for handling access privileges. Most system calls were also omitted except the termination call to signal the code execution end. Co-processor instructions were also omitted since they are not applicable.

B.1 ALU instructions

All the ALU instructions are listed in this section. The first operand is always a register source and the second can be either an immediate or a register value.

B.1.1 AND

31-28	27-21	20	19-16	15-12	11-0
cond	0000000	S	Rn	Rd	second_operand

```
if predication == true then
    Rd = Rn AND
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

The “AND” instruction performs a bitwise AND operation on two values. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.
- Rd: Target register.
- second_operand: The second operand value.

This instruction is directly translated to the ρ-VEX instruction “and”.

B.1.2 EOR

31-28	27-21	20	19-16	15-12	11-0
cond	0000001	S	Rn	Rd	second_operand

```
if predication == true then
    Rd = Rn XOR second_operand
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

The “EOR” instruction performs a bitwise XOR operation on two values. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.

- Rn: First operand.
- Rd: Target register.
- second_operand the second operand. See section 4.5.

This instruction is directly translated to the ρ-VEX instruction “xor”.

B.1.3 SUB

31-28	27-21	20	19-16	15-12	11-0
cond	0000010	S	Rn	Rd	second_operand

```
1 if predication == true then
2    Rd = Rn − second_operand
3    if S == 1 then
4        N Flag = Rd[31]
5        Z Flag = if Rd == 0 then 1 else 0
6        C Flag = NOT BorrowFrom(Rn − second_operand)
7        V Flag = OverflowFrom(Rn − second_operand)
```

The “SUB” instruction performs a subtraction between two values. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.
- Rd: Target register.
- second_operand the second operand. See section 4.5.

This instruction is directly translated to the ρ-VEX instruction “sub”, with the only difference being the order of subtraction being reverse. i.e. ρ-VEX performs b - a instead of a - b, as a result Rn and the second_operand should be swapped.

B.1.4 RSB

31-28	27-21	20	19-16	15-12	11-0
cond	0000011	S	Rn	Rd	second_operand

```
1 if predication == true then
2    Rd = Rn XOR second_operand
3    if S == 1 then
4        N Flag = Rd[31]
5        Z Flag = if Rd == 0 then 1 else 0
6        C Flag = shifter_carry_out
7        V Flag = unaffected
```

The “RSB”(Reverse Subtract) instruction performs a subtraction between two values. First value comes from a register and the second comes from a register and can be either shifted or not. Same as the normal “SUB” instruction but the order is reversed. i.e. second_operand - Rn instead of Rn - second_operand. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.
- Rd: Target register.
- second_operand: the second operand.

This instruction is directly translated to the ρ-VEX instruction “sub”.

B.1.5 ADD

31-28	27-21	20	19-16	15-12	11-0
cond	0000100	S	Rn	Rd	second_operand

```
1 if predication == true then
2     Rd = Rn + second_operand
3     if S == 1 then
4         N Flag = Rd[31]
5         Z Flag = if Rd == 0 then 1 else 0
6         C Flag = CarryFrom(Rn +  second_operand)
7         V Flag = OverflowFrom(Rn +  second_operand)
```

The “ADD” instruction performs an addition between two values. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.
- Rd: Target register.
- second_operand: the second operand.

This instruction is directly translated to the ρ-VEX instruction “add”.

31-28	27-21	20	19-16	15-12	11-0
cond	0000101	S	Rn	Rd	second_operand

B.1.6 ADC

```
if predication == true then
    Rd = Rn + second_operand + C Flag
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn +  second_operand + C flag)
        V Flag = OverflowFrom(Rn +  second_operand + C flag)
```

The “ADC” (Add with carry) instruction performs an addition between two values and a carry in. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. The carry in is the value of the C flag. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.
- Rd: Target register.
- second_operand: the second operand.

This instruction cannot be directly translated. Even though the ρ-VEX instruction “addcg” can perform this operation it cannot be invoked independently since the carry in needs to be located in a branch register. As a result a “tbit” instruction needs to be called initially to copy the C flag to the branch register. In order to translate the “ADC” instruction 3 ρ-VEX instructions are needed and these are the following:

```
tbit ba, rf, 1\\copy the C flag from the flag register to a branch register
addcg
goto -2\\jump back to normal execution
```

B.1.7 SBC

31-28	27-21	20	19-16	15-12	11-0
cond	0000110	S	Rn	Rd	second_operand

```
if predication == true then
    Rd = Rn - second_operand - NOT(C Flag)
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - second_operand - NOT(C Flag))
        V Flag = OverflowFrom(Rn - second_operand - NOT(C Flag))
```

The “SBC” (Subtract with carry) instruction subtracts a value and a carry from another value. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. The carry in is the value of the C flag. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.
- Rd: Target register.
- second_operand: the second operand.

This instruction cannot be directly translated. As a result this instruction is performed in 4 stages. First the C flag is copied to a general purpose register. Next two subtractions are performed, Rn - second_operand - C flag. Note that the subtraction in ρ-VEX is reversed compared to ARM(b - a instead of a - B) so the values need to be swapped.

```
1 tbit ra, rf, 1\\copy the C flag from the flag register to a general
    register
2 sub
3 sub
4 goto -3\\jump back to normal execution
```

B.1.8 RSC

31-28	27-21	20	19-16	15-12	11-0
cond	0000111	S	Rn	Rd	second_operand

```
1 if predication == true then
2     Rd = second_operand - Rn - NOT(C Flag)
3     if S == 1 then
4         N Flag = Rd[31]
5         Z Flag = if Rd == 0 then 1 else 0
6         C Flag = NOT BorrowFrom(second_operand - Rn - NOT(C Flag))
7         V Flag = OverflowFrom(second_operand - Rn - NOT(C Flag))
```

The “RSC” (Reverse Subtract with carry) instruction subtracts a value and a carry from another value. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. The carry in is the value of the C flag. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.

- Rd: Target register.
- second_operand: the second operand.

This instruction cannot be directly translated. As a result this instruction is performed in 4 stages. First the C flag is copied to a general purpose register. Next two subtractions are performed, second_operand - Rn - C flag.

```
1 tbit ra, rf, 1\\copy the C flag from the flag register to a general
      register
2 sub
3 sub
4 goto −3\\jump back to normal execution
```

B.1.9 TST

31-28	27-21	20	19-16	15-12	11-0
cond	0001000	S	Rn	0000	second_operand

```
1 if predication == true then
2     alu_out = Rn AND second_operand
3     N Flag = alu_out[31]
4     Z Flag = if alu_out == 0 then 1 else 0
5     C Flag = shifter_carry_out
6     V Flag = unaffected
```

The “TST” instruction compares two values by ANDing them. No register is updated but the flags are updated based on the result of the calculation. Depending on the implementation this instruction can either dispatch a NOP if the flags are calculated inside the translator, or an AND instruction. In the latter case the result should be stored in a scrap register which will be used to calculate the flags. The ’N’ flag can be calculated with a “tbit” instruction in order to extract the 31st bit of the result and the Z flag with a simple “cmpeq” instruction.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.
- second_operand: the second operand.

B.1.10 TEQ

31-28	27-21	20	19-16	15-12	11-0
cond	0001001	S	Rn	0000	second_operand

```
if predication == true then
   alu_out = Rn XOR second_operand
   N Flag = alu_out[31]
   Z Flag = if alu_out == 0 then 1 else 0
   C Flag = shifter_carry_out
   V Flag = unaffected
```

The “TEQ” instruction compares two values by XORing them. No register is updated but the flags are updated based on the result of the calculation. Depending on the implementation this instruction can either dispatch a NOP if the flags are calculated inside the translator, or an XOR instruction. In the latter case the result should be stored in a scrap register which will be used to calculate the flags. The ’N’ flag can be calculated with a “tbit” instruction in order to extract the 31st bit of the result and the Z flag with a simple “cmpeq” instruction.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.
- second_operand: the second operand.

B.1.11 CMP

31-28	27-21	20	19-16	15-12	11-0
cond	0001010	S	Rn	0000	second_operand

```
if predication == true then
    alu_out = Rn - second_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = NOT BorrowFrom(Rn - second_operand)
    V Flag = OverflowFrom(Rn - second_operand)
```

The “CMP” instruction compares two values by subtracting them. No register is updated but the flags are updated based on the result of the calculation. Depending on the implementation this instruction can either dispatch a NOP if the flags are calculated inside the translator, or an XOR instruction. In the latter case the result should be stored in a scrap register which will be used to calculate the flags. The ’N’ flag can be calculated with a “tbit” instruction in order to extract the 31st bit of the result and the Z flag with a simple “cmpeq” instruction, and the C and V flags with the procedures described here 4.4. A complete example can be seen here (See table 4.5).

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.
- second_operand: the second operand.

B.1.12 CMN

31-28	27-20	19-16	15-12	11-0
cond	00010111	Rn	1111	second_operand

```
if predication == 1 then
     alu_out = Rn + second\_operand
     N Flag = alu_out[31]
     Z Flag = if alu_out == 0 then 1 else 0
     C Flag = CarryFrom(Rn + second_operand)
     V Flag = OverflowFrom(Rn + second_operand)
```

The “CMN”(Compare Negative) instruction compares on value with the two’s complement of another one. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. This instruction always updates the flags.

- cond: The instruction CZNV condition flags.
- RD: Target register.
- Rn: Source register.
- second_operand: the second operand.

This instruction can be directly translated to the “ADD” ρ-VEX instruction.

B.1.13 ORR

31-28	27-21	20	19-16	15-12	11-0
cond	0001100	S	Rn	Rd	second_operand

```
if predication == true then
    Rd = Rn OR second_operand
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

The “ORR” instruction performs bitwise OR to two values. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rn: First operand.

- Rd: Target register.
- second_operand: the second operand.

This instruction can be directly translated to the "or" ρ-VEX instruction.

B.1.14 MOV

31-28	27-21	20	19-16	15-12	11-0
cond	0001101	S	0000	Rd	second_operand

```
if predication == true then
    Rd = second_operand
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

The "MOV" instruction places a value in a register. The value comes from a register or an immediate, and it can be shifted, rotated or neither. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rd: Target register.
- second_operand: the second operand.

This instruction can be implemented with many ways one of which is ORing the value with zero. As a result the ρ-VEX "OR" instruction can be used for direct translation.

B.1.15 BIC

31-28	27-21	20	19-16	15-12	11-0
cond	0001110	S	Rn	Rd	second_operand

```
if predication == true then
    Rd = Rn AND NOT second_operand
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

The "BIC" instruction performs a bitwise AND operation between a value and the complement of another value. First value comes from a register and the second comes from a register or an immediate. The second register source can be shifted, rotated or neither. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rd: Target register.
- Rn: Source register.
- second_operand: The second operand.

This instruction can be directly translated to the "andc" ρ-VEX instruction. Note that the order of the values should be reversed since the ρ-VEX "andc" performs (NOT a) AND b while ARM requires a AND (NOT b).

B.1.16 MVN

31-28	27-21	20	19-16	15-12	11-0
cond	0001111	S	0000	Rd	second_operand

```
1 if predication == true then
2     Rd = NOT second_operand
3     if S == 1 then
4         N Flag = Rd[31]
5         Z Flag = if Rd == 0 then 1 else 0
6         C Flag = shifter_carry_out
7         V Flag = unaffected
```

The "MVN" instruction places the complement of a value in a register. The value comes from a register or an immediate, and it can be shifted, rotated or neither. This instruction cannot target the PC.

- cond: The instruction CZNV condition flags.
- S: If S = 1 then the instruction updates the CZNV flags, other wise it does not.
- Rd: Target register.
- second_operand: the second operand.

This instruction can be implemented with the "orc" ρ-VEX instruction. Note that the order of the values should be reversed since the ρ-VEX "orc" performs (NOT a) OR b while ARM requires a OR (NOT b).

B.2 Load/Store instruction

These instructions are the memory access instructions. Most of these instructions are translated into multiple ρ-VEX instructions mostly in order to calculate the address. For each of the following instructions, the address should be calculated according to the cases presented in the addressing mode section 4.6. For this reason some of the instruction encoding bits are indifferent(marked with 'X') regarding the type of store/load instruction,

however these bits are used to determine the addressing mode. Furthermore, regarding the “load” instructions, if the target is register R15(Program counter), then writing the PC with the new value should be treated as a jump to that location(see 4.3.3.2).

B.2.1 STRB

31-28	27-26	25-23	22	21	20	19-16	15-12	11-0
cond	01	XXX	1	X	0	Rn	Rd	XXXXXXXXXXXX

```
if predication == true then
    Memory[address,1] = Rd[7:0]
```

Stores the least significant byte of a register to the memory.

- Rn: Base address register.
- Rd: Data Source register.

The equivalent ρ-VEX instruction is “stb”. Furthermore, this instruction is executed in two stages, first the address is calculated according to 4.6, and then the equivalent ρ-VEX store byte instruction is executed. In case of the emulation it will be emulated by the following:

```
calculate address\\calculate address according to addressing mode
stb rx, 0\\the final address is located in rx
goto -n\\jump backwards. n is the amount of instructions the address
    calculation requires plus 2.
```

B.2.2 STRH

31-28	27-25	24-21	20	19-16	15-12	11-8	7-4	3-0
cond	000	XXXX	0	Rn	Rd	XXXX	1011	XXXX

```
if predication == true then
    Memory[address,2] = Rd[15:0]
```

Stores a halfword of a register to the memory.

- Rn: Base address register.
- Rd: Data Source register.

The equivalent ρ-VEX instruction is “sth”. Furthermore, this instruction is executed in two stages, first the address is calculated according to 4.6, and then the equivalent ρ-VEX store byte instruction is executed. In case of the emulation it will be emulated by the following:

```
calculate address\\calculate address according to addressing mode
sth rx, 0\\the final address is located in rx
goto -n\\jump backwards. n is the amount of instructions the address
    calculation requires plus 2.
```

B.2.3 STR

31-28	27-26	25-23	22	21	20	19-16	15-12	11-0
cond	01	XXX	1	X	0	Rn	Rd	XXXXXXXXXXXX

```
if predication == true then
    Memory[address,4] = Rd
```

Stores a word from a register to the memory.

- Rn: Base address register.
- Rd: Data Source register.

The equivalent ρ-VEX instruction is “stw”. Furthermore, this instruction is executed in two stages, first the address is calculated according to 4.6, and then the equivalent ρ-VEX store byte instruction is executed. In case of the emulation it will be emulated by the following:

```
calculate address\\calculate address according to addressing mode
stw rx, 0\\the final address is located in rx
goto -n\\jump backwards. n is the amount of instructions the address
    calculation requires plus 2.
```

B.2.4 LDRB

31-28	27-26	25-23	22	21	20	19-16	15-12	11-0
cond	01	XXX	1	X	1	Rn	Rd	XXXXXXXXXXXX

```
if predication == true then
    Rd = Memory[address,1]
```

Loads a byte from memory and zero extends it.

- Rn: Base address register.
- Rd: Data Source register.

This instruction cannot target the PC. The equivalent ρ-VEX instruction is “ldbu”. Furthermore, this instruction is executed in two stages, first the address is calculated according to 4.6, and then the equivalent ρ-VEX store byte instruction is executed. In case of the emulation it will be emulated by the following:

```
calculate address\\calculate address according to addressing mode
ldbu rd, rx 0\\the final address is located in rx
goto -n\\jump backwards. n is the amount of instructions the address
    calculation requires plus 2.
```

31-28	27-25	24-21	20	19-16	15-12	11-8	7-4	3-0
cond	000	XXXX	1	Rn	Rd	XXXX	1101	XXXX

B.2.5 LDRSB

```
if predication == true then
    data = sign_ext(Memory[address,1])
```

Loads a byte from memory and sign extends it.

- Rn: Base address register.
- Rd: Data Source register.

This instruction cannot target the PC. The equivalent ρ-VEX instruction is "ldb". Furthermore, this instruction is executed in two stages, first the address is calculated according to 4.6, and then the equivalent ρ-VEX store byte instruction is executed. In case of the emulation it will be emulated by the following:

```
calculate address\\calculate address according to addressing mode
ldb rx, 0\\the final address is located in rx
goto -n\\jump backwards. n is the amount of instructions the address
    calculation requires plus 2.
```

B.2.6 LDRH

31-28	27-25	24-21	20	19-16	15-12	11-8	7-4	3-0
cond	000	XXXX	1	Rn	Rd	XXXX	1011	XXXX

```
if predication == true then
    data = Memory[address,2]
```

Loads a halfword from memory and zero extends it. This instruction cannot target the PC.

- Rn: Base address register.
- Rd: Data Source register.

The equivalent ρ-VEX instruction is "ldhu". Furthermore, this instruction is executed in two stages, first the address is calculated according to 4.6, and then the equivalent ρ-VEX store byte instruction is executed. In case of the emulation it will be emulated by the following:

```
calculate address\\calculate address according to addressing mode
ldhu rx, 0\\the final address is located in rx
goto -n\\jump backwards. n is the amount of instructions the address
    calculation requires plus 2.
```

31-28	27-25	24-21	20	19-16	15-12	11-8	7-4	3-0
cond	000	XXXX	1	Rn	Rd	XXXX	1111	XXXX

B.2.7 LDRSH

```
if predication == true then
    data = Memory[address,2]
```

Loads a halfword from memory and sign extends it. This instruction cannot target the PC.

- Rn: Base address register.
- Rd: Data Source register.

The equivalent ρ-VEX instruction is “ldh”. Furthermore, this instruction is executed in two stages, first the address is calculated according to 4.6, and then the equivalent ρ-VEX store byte instruction is executed. In case of the emulation it will be emulated by the following:

```
calculate address\\calculate address according to addressing mode
ldh rx, 0\\the final address is located in rx
goto -n\\jump backwards. n is the amount of instructions the address
    calculation requires plus 2.
```

B.2.8 LDR

31-28	27-26	25-23	22	21	20	19-16	15-12	11-0
cond	01	XXX	0	X	1	Rn	Rd	XXXXXXXXXXXX

```
if predication == true then
    data = Memory[address,4]
```

Loads a word from the memory to a register

- Rn: Base address register.
- Rd: Data Source register.

The equivalent ρ-VEX instruction is “ldw”. Furthermore, this instruction is executed in two stages, first the address is calculated according to 4.6, and then the equivalent ρ-VEX store byte instruction is executed. In case of the emulation it will be emulated by the following:

```
calculate address\\calculate address according to addressing mode
ldw rx, 0\\the final address is located in rx
goto -n\\jump backwards. n is the amount of instructions the address
    calculation requires plus 2.
```

Furthermore if the target address of Rd is R15(PC) then a jump should occur to that address according to this 4.3.3.2.

31-28	27-25	24-23	22	21	20	19-16	15-0
cond	100	XX	0	X	1	Rn	register_list

B.2.9 LDM

```
if predication == true then
    address = start_address
    for i = 0 to 14
    {
        if register_list[i] == 1 then
            Ri = Memory[address,4]
            address = address + 4
    }
assert end_address == address - 4
if register_list[15] == 1 then \\PC has been loaded with a new value
    PERFORM JUMP
```

LDM(Load multiple) instructions loads multiple sequential memory locations. Since ρ-VEX does not support this kind of memory accesses the memory needs to be accessed repeatedly until all the required data is transfered. This instruction executes in two basic stages: first the start and end addresses are calculated according to 4.6.10 and 4.6.11. Once these two addresses are calculated then the translator starts to dispatch memory load instructions one per cycle. Within the instruction there is the register_list field. This is a 16 bit field that determines which registers will be updated, 1 bit for each general purpose register register_list(14 downto 0) and 1 bit register_list(15) for the PC. For each of these bits the translator checks if it is 1 or 0 and dispatches a load instruction if it is 1. If the value was 1 the address is also incremented by 4. Initially the 14 LSBs of the register_list are checked starting from the LSB and moving onward. When all 14 bits have been checked then the 15th is also checked. If this bit is set this means that the new value that updates the PC should be treated as a jump and the procedure shown here 4.3.3.2 should be followed. Lastly an optional step is followed where the assertions is performed to verify that the final address generated is equal to the base address. The register_list is available directly to the translator, as a result the bit checking can be performed inside the translator. If the bit is zero nothing is dispatched and the translator proceeds to check the next bit. If the bit is one then in the case of emulation the following instructions should be dispatched:

```
ldw rd, address, 0//load mem(address) to rd
add rx1, rx1, 4//increment the address register by 4
```

When all bits have been checked then if there is no jump to be performed from PC loading, then a simple “goto” instruction is dispatched that jumps backwards to the original PC+1. The backwards jump amount is equal to the number_of_registers_loaded * 2 + 1. This is with the assumption that the translation procedure followed in the start_address and end_address calculation shown here 4.6.10, 3.15 also resumes proper program flow i.e. the loading sequence starts with the base instruction PC value.

B.2.10 STM

```
if predication == true then
```

31-28	27-25	24-23	22-19	19-16	15-0
cond	100	XX	100	Rn	register_list

```
    address = start_address
    for i = 0 to 15
    {
        if register_list[i] == 1 then
            Memory[address,4] = Ri
            address = address + 4
    }
assert end_address == address - 4
```

The STM(Multiple store) is handled the exact same way as the previous instruction "LDM" B.2.9. The only two differences is that instead of "ldw" instructions the "stw" is used for the translation and also there is not check performed for PC jumps.

B.3 Multiplication instructions

This section presents the multiplication instructions as well as the multiply & accumulate instructions. Since ρ-VEX does not contain a full 32x32 multiplier all of these instructions needs to be emulated(see 3.7.10 of the ρ-VEX manual).

B.3.1 MUL

31-28	27-21	20	19-16	15-12	11-8	7-4	3-0
cond	0000000	S	Rd	0000	Rs	1001	Rm

```
if predication == true then
    Rd = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = unaffected
        V Flag = unaffected
```

- Rd: Destination register.
- Rm: First operand.
- Rs: Second operand.
- S: Determines whether the flags will be updated or not.

The instruction MUL multiplies two 32-bit registers and stores the 32 LSBs of the result to Rd. ρ-VEX does not contain a 32x32 multiplier however it contains a 16x32 multiplier, as a result the lower 32 bits of the result can be calculated by the following instructions:

```
mpylu rx1, Rs, Rm
mpyhs rx2, Rs, rm//signed
cmpne br1, r0, r0
addcg Rd, br1, rx1, rx2, br1
goto -4
```

This instruction cannot target R15(PC)

B.3.2 UMUL

31-28	27-21	20	19-16	15-12	11-8	7-4	3-0
cond	0000100	S	RdHi	RdLo	Rs	1001	Rm

```
if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32] /* Unsigned multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected
        V Flag = unaffected
```

- RdHi:Destination register for the upper 32-bit result
- RdLo:Destination register for the lower 32-bit result
- Rs: First operand.
- Rm: Second operand.
- S: Determines whether the flags will be updated or not.

This instruction multiples two unsigned 32 bit values and stores the result in two 32 bit registers, “RdHi” for the upper 32 bits of the result and “RdLo” for the lower 32 bits of the result. This instruction can be emulated with the following instructions:

```
mpylu rx1, Rm, Rs
mpylu rx2, Rm, Rs//unsigned
mpylhus rx3, Rm, Rs
mpyhhs rx4, Rm, Rs
cmpne br1, r0, r0
addcg RdLo, br1, rx1, rx2, br1
addcg RdHi, brx, rx3, rx4, br1
goto -7
```

B.3.3 SMUL

31-28	27-21	20	19-16	15-12	11-8	7-4	3-0
cond	0000110	S	RdHi	RdLo	Rs	1001	Rm

```
if predication == true then
    RdHi = (Rm * Rs)[63:32] /* signed multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
```

```
6         Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
7         C Flag = unaffected
8         V Flag = unaffected
```

- RdHi:Destination register for the upper 32-bit result
- RdLo:Destination register for the lower 32-bit result
- Rs: First operand.
- Rm: Second operand.
- S: Determines whether the flags will be updated or not.

This instruction multiples two signed 32 bit values and stores the result in two 32 bit registers, "RdHi" for the upper 32 bits of the result and "RdLo" for the lower 32 bits of the result. This instruction can be emulated with the following instructions:

```
1 mpylu rx1, Rm, Rs
2 mpyhs rx2, Rm, Rs//signed
3 mpylhus rx3, Rm, Rs
4 mpyhhs rx4, Rm, Rs
5 cmpne br1, r0, r0
6 addcg RdLo, br1, rx1, rx2, br1
7 addcg RdHi, brx, rx3, rx4, br1
8 goto -7
```

B.3.4 MLA

31-28	27-21	20	19-16	15-12	11-8	7-4	3-0
cond	0000001	S	Rd	Rn	Rs	1001	Rm

```
1 if predication == true then
2     RdHi = (Rm * Rs)[63:32] /* signed multiplication */
3     RdLo = (Rm * Rs)[31:0]
4     if S == 1 then
5         N Flag = RdHi[31]
6         Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
7         C Flag = unaffected
8         V Flag = unaffected
```

- Rd:Destination register
- Rm: First operand.
- Rs: Second operand.
- Rn: Added value.
- S: Determines whether the flags will be updated or not.

This instruction multiples two signed 32 bit values and adds a third value. The 32 LSBs are written to the destination register.

```
mpylu rx1, Rs, Rm
mpyhs rx2, Rs, rm//signed
cmpne br1, r0, r0
addcg Rd, br1, rx1, rx2, br1
add Rd, Rd, Rn
goto -5
```

B.3.5 UMLAL

31-28	27-21	20	19-16	15-12	11-8	7-4	3-0
cond	0000101	S	RdHi	RdLo	Rs	1001	Rm

```
if predication == true then
    RdLo = (Rm * Rs)[31:0] + RdLo /* Unsigned multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected
        V Flag = unaffected
```

- RdHi:Destination register for the upper 32-bit result
- RdLo:Destination register for the lower 32-bit result
- Rs: First operand.
- Rm: Second operand.
- S: Determines whether the flags will be updated or not.

This instruction multiplies two unsigned 32 bit values to produce a 64 bit result, the 32 MSBs of the result are added to the old value of “RdHi” and the result is written back to “RdHi”. Similarly for the 32 LSBs and the “RdLo”. The way this is emulated is by first calculating the 64-bit result and storing it in scrap registers(rx6 && rx5). Next two “addcg” instructions are used to sum this result with the previous “RdHi” and “RdLow”.

```
mpylu rx1, Rm, Rs
mpylu rx2, Rm, Rs//unsigned
mpylhus rx3, Rm, Rs
mpyhhs rx4, Rm, Rs
cmpne br1, r0, r0
cmpne br2, r0, r0
cmpne br4, r0, r0
addcg rx5 br3, rx1, rx2, br1
addcg rx6 br2, rx3, rx4, br3
addcg RdLo, br4, rx5, RdLo, br4
addcg RdHi, brx, rx6, RdHi, br4
goto -11
```

B.3.6 SMLAL

31-28	27-21	20	19-16	15-12	11-8	7-4	3-0
cond	0000101	S	RdHi	RdLo	Rs	1001	Rm

```
if predication == true then
    RdLo = (Rm * Rs)[31:0] + RdLo /* Unsigned multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected
        V Flag = unaffected
```

- RdHi:Destination register for the upper 32-bit result
- RdLo:Destination register for the lower 32-bit result
- Rs: First operand.
- Rm: Second operand.
- S: Determines whether the flags will be updated or not.

This instruction multiplies two signed 32 bit values to produce a 64 bit result, the 32 MSBs of the result are added to the old value of "RdHi" and the result is written back to "RdHi". Similarly for the 32 LSBs and the "RdLo". The way this is emulated is by first calculating the 64-bit result and storing it in scrap registers(rx6 && rx5). Next two "addcg" instructions are used to sum this result with the previous "RdHi" and "RdLow".

```
mpylu rx1, Rm, Rs
mpylu rx2, Rm, Rs//unsigned
mpylhus rx3, Rm, Rs
mpyhhs rx4, Rm, Rs
cmpne br1, r0, r0
cmpne br2, r0, r0
cmpne br4, r0, r0
addcg rx5 br3, rx1, rx2, br1
addcg rx6 br2, rx3, rx4, br3
addcg RdLo, br4, rx5, RdLo, br4
addcg RdHi, brx, rx6, RdHi, br4
goto -11
```

B.4 Branch Instructions

ARM architecture has two branch instructions, one that branches and one that branches and links.

31-28	27-25	24	23-0
cond	101	0	jump_value

B.4.1 B

```
if predication == true then
    PC = PC + (SignExt_30(jump_value) << 2)
```

This instruction branches to PC plus the jump value. The jump value is always aligned to 4 bytes as a result the 2 LSBs of the jump value are not stored. As a result the value needs to be shifted to the left by two and then sign extended to 32 bits. The jump offset can be calculated within the translator, as a result this instruction can be translated to a simple "goto" ρ-VEX instruction with the offset pre-calculated within the translator.

B.4.2 BL

31-28	27-25	24	23-0
cond	101	1	jump_value

```
if predication == true then
    PC = PC + (SignExt_30(jump_value) << 2)
    LR = PC+1
```

This instruction branches to PC plus the jump value while storing the next PC in the link register. The jump value is always aligned to 4 bytes as a result the 2 LSBs of the jump value are not stored. As a result the value needs to be shifted to the left by two and then sign extended to 32 bits. The jump offset can be calculated within the translator, as a result this instruction can be translated to a simple "goto" ρ-VEX instruction with the offset pre-calculated within the translator. This instruction can be directly translated to the ρ-VEX "call". Extra attention needs to be paid to set the link register in ρ-VEX as R63 and also map any instruction that targets or reads the link register as R63.

B.5 Miscellaneous Instructions

B.5.1 SWP

31-28	27-20	19-16	15-12	11-8	7-4	3-0
cond	00010000	Rn	Rd	0000	1001	Rm

```
if predication == true then
    temp = Memory[address,4]
    Memory[address,4] = Rm
    Rd = temp
```

- Rn: Contains the address of the memory

- Rd: Destination register
- Rm: Data to be written to memory

This instruction loads a value from the memory location pointed by Rn and stores it in a temporary register. Next another value in register Rm is stored to the same memory location pointed by Rn. Then the temporary register is copied to Rd. If Rd is the same as Rm the instruction essentially swaps a value from memory and register Rn. Since there is no swap instruction in ρ-VEX this instruction needs to be emulated with three instructions. One instruction to load the data to the temporary register, one instruction to store the data from Rm to memory and one instruction to copy the data from the temporary location to Rd. This can be done with the following instructions:

```
1 ldw Rx1, Rn, 0//load to Rx1
2 stw Rm, Rn, 0//store Rm to mem
3 or Rd, Rx1, 0//copy Rx1 to Rd
4 goto −3//jump back to normal program flow
```

B.5.2 SWPB

31-28	27-20	19-16	15-12	11-8	7-4	3-0
cond	00010100	Rn	Rd	0000	1001	Rm

```
1 if predication == true then
2     temp = Memory[address,1]
3     Memory[address,1] = Rm[7:0]
4     Rd = temp
```

- Rn: Contains the address of the memory
- Rd: Destination register
- Rm: Data to be written to memory

This instruction loads a byte from the memory location pointed by Rn and stores it in the LSbyte of a temporary register. Next the LSByte in register Rm is stored to the same memory location pointed by Rn, after it is zero extended. Then the temporary register is copied to Rd. If Rd is the same as Rm the instruction essentially swaps a value from memory and register Rn. Since there is no swap instruction in ρ-VEX this instruction needs to be emulated with three instructions. One instruction to load the data to the temporary register, one instruction to store the data from Rm to memory and one instruction to copy the data from the temporary location to Rd. This can be done with the following instructions:

```
1 ldbu Rx1, Rn, 0//load to Rx1
2 stb Rm, Rn, 0//store Rm to mem
3 or Rd, Rx1, 0//copy Rx1 to Rd
4 goto −3//jump back to normal program flow
```

31-28	27-24	26-0
cond	1111	immediate

B.5.3 SWI

```
if predication == true then
    execute software interrupt
```

This instruction is the system call of ARM. The only system call implemented is the termination call. The ARM system call convention is implemented with two ways, one the immediate field in the instruction indicates the system call, or two, register R0(in our case R1 see 4.1.2) indicates the system call. The termination call has register R1 == 24, as a result when a system call is read a check should be performed on R1 == 24 and then a branch to the original instruction. If the branch is taken this means that the system call was indeed a termination call, if not then the system call is ignored and execution proceeds. The following instructions show this procedure.

```
cmpeq br1, R1, 24
goto -2
br -1
```