SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Emulation of x86 binaries on the RISC–V architecture

Simon Kammermeier

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Emulation of x86 binaries on the RISC–V architecture

Emulation von x86 Binärprogrammen auf der RISC–V Architektur

Author:	Simon Kammermeier
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Dr. Redha Gouciem
Submission Date:	15.04.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.04.2023 Simon Kammermeier

Abstract

New computer architectures face many challenges trying to establish themselves in the computer landscape. One of the problems hindering their adoption is support for exiting applications. Binary translation provides a way to bridge this gap without having to compile and redistribute existing applications. There are different approaches for the point in time of the translation. Static translation chooses to do this expensive process before program execution, which reduces the expected overhead at runtime. As opposed to dynamic translation it faces big challenges to discover instructions in the binary file, which means it is not a general solution.

In this thesis we introduce a back end for dynamic translation from x86 to RISC–V. Its intended use case is to be a fallback solution for the static translation path of the Arancini project. At runtime any missing translations are filled in using our back end. We expected only infrequent use of this code path, so the implementation focuses on simplicity, correctness and quick translation times. This results in low quality of the generated code.

We tested our implementation using a variety of single threaded test programs. Due to its unfinished state we had to test with the static translation disabled. As expected from the initial design goals this caused major overheads in comparison to native programs. Our evaluation also shows that our approach has reached one of its main aims and produced correct translations of single threaded integer based x86 binaries to RISC–V. However, the low performance means it is impractical to use without the optimized static path or other major improvements of the generate code sequences.

Contents

1 Introduction

The x86 instruction set architecture (ISA), introduced and developed by Intel [Int23] and AMD [Adv23], has been the dominant CPU architecture for desktop and notebook computers for many years. Due to this market position there has not been a need for most applications to be available for any other ISA.

The ARM architecture has been dominant in low-power mobile settings in recent history. Almost all smartphones use chips based on ARM and it is quite popular in micro computers, e.g. Raspberry PI, and micro controllers as well. In recent years this architecture has started to see increased adoption in the desktop and server markets. Others, like RISC–V, have shown increased interest in industry as well.

To attract customers to switch to these chips the transition should be as seamless for users as possible. One of the important points here is allowing customers to use all the software they are used to. Especially in commercial settings a migration to new software would often involve unreasonable costs and effort.

For programs with available source code simply compiling for a new target architecture provides the best quality binaries. But many programs are unmaintained and do not have their source code openly available. Implicit assumptions of programmers about the CPU architecture make some recompiled programs behave wrong in very subtle ways. On top of that many programming languages offer target dependent ways to program, for example compiler intrinsics in C(++), that are frequently used to achieve better performing binaries than the compiler optimizations give you. These map directly to assembly instructions so the compiler cannot convert these to a new architecture.

Additionally, a big majority of the users of computer software never touched a compiler themselves, which means even programs that allow a simple recompilation for another architecture, would provide a serious barrier of switching for those users. Therefore, in many cases a simple recompilation does not provide a solution for allowing users to keep using their applications. A switch to another CPU architecture would mean loosing access to those legacy programs.

For these reasons, backwards compatibility has historically been a major selling point for a new CPU generation. This is also the reason today's Intel and AMD x86 CPUs are still compatible with programs written for the 16 bit Intel 8086 released in 1978.

Instead of providing hardware support for old ISAs another way to keep compatibility is to employ a technique called binary translation. All features and instructions of the old architecture are mapped to equivalent ones on the new architecture. In many cases this mapping is inefficient and causes significant overheads. Reducing the performance loss to a point that makes it feasible for every day use is key for the success of such a software.

This concept has been implemented previously in different programs. As an example, QEMU [Bel05] is the standard open source binary translator with support for many different target and source ISAs. Proprietary tools exist as well. The currently most popular example of this is Apple's Rosetta 2 [App23]. It implements translation from x86–64 used in previous Apple computers to the ARM based Apple silicon chips used in their current systems.

As an alternative to these solutions, we propose **Arancini**, a modern binary translator employing both static and dynamic translation. The aim of this project is to establish an efficient binary translator with support for multithreaded applications. It needs to correctly implement the memory semantics of the source binary regardless of the processor architecture it was translated to. But crucially it should not be limited to specific hardware, like existing proprietary tools. This thesis specifically worries about x86 as the source and RISC–V as the target architecture, but the project aims to provide a flexible framework allowing many different ISAs.

We add support to Arancini to generate native RISC–V code from the Arancini intermediate representation (IR). The main contribution is a dynamic back end with complete support for the scalar integer subset of the IR. Integrated with the existing x86 front end of Arancini and its runtime system this provides a proof of concept dynamic binary translator for single threaded programs from x86 to RISC–V.

In the following section we will cover the necessary background to understand the general concept of binary translation and explain the main concepts and differences of the x86 and RISC–V ISAs. Afterwards we will give an overview of the design and the general program flow of Arancini, the binary translator powering our implementation. Chapter 4 explains the design of the intermediate representation used in Arancini and presents the core components of the RISC–V implementation. Next, we present the implementation details of the translator focusing on the generated instruction sequences. Further, we evaluate the performance of our approach and its success in producing correct implementations. At the end we take a look at related work and future points for improvement.

2 Background

In the following chapter some required background information to understand later chapters will be given. It is important to understand basic concepts of binary translation including definitions of terms like basic block, dynamic and static binary translation.

2.1 Binary Translation

Binary translation is used to be able to run program binaries on a host system using an instruction set architecture (ISA) different from the one of the guest program. [Pro02; UC00]

2.1.1 Approaches

There are two main approaches used for binary translation, they will be explained in the following.

Dynamic Binary Translation

Dynamic binary translation (DBT) translates the code at run time. Whenever a program counter is discovered that does not have an associated translation yet, a new translation is created and added to a code cache. Typically a translation granularity bigger than a single instruction is used to reduce the overhead of the translator. The blocks used for this are typically chosen so they have a single entry and exit point. In this way the translation can assume all instructions in a block get executed once it was entered.

At the end of a block the next block is selected from the code cache or newly translated. This gradual translation, just in time (JIT) translation process distributes the overhead over the whole execution and minimises pause times. Not executed blocks do not need to be translated, so they do not cause any overhead. On the other hand extensive optimizations are not possible as that would increase the run time overhead.

Static Binary Translation

An entirely different approach is static translation ahead of run time. All instructions in the binary need to be converted to code runable on the target without using control flow information available only at run time.

The typical complications that arise using this technique is discovering all code paths. Some of them may only be reached by indirect branches, the targets of which typically cannot be determined without running the program. This is also leads to a second problem, the dynamic mapping of indirect jump targets to the translated program counters.

Another problem is the possibility of data mixed into memory regions containing instructions. In the typical von Neumann architecture there is no differentiation between data and instruction memory. So it is in general not possible to detect whether a specific address belongs to code before executing the program.

The main advantage of this approach is that any overhead caused by the translation process is only incurred before program run time. This speeds up program execution after the one time translation was finished. Therefore it is not a problem when the translation takes longer. So optimizations can be applied even if it takes a significant amount of time. Additionally, static translation allows for a more global view of the program. Often entire functions are lifted at once which provides better possibilities for optimization passes.

2.1.2 Basic Block

One very important concept of binary translation are **basic blocks** (BB) [SN05]. Basic blocks are a sequence of instructions that have exactly one entry and one exit. This means no instructions inside a basic block can be jump targets and only the last instruction can be a jump itself. So they are an ideal choice for the granularity of binary translation. Depending on the context there are two different types of basic blocks that play a role.

Static basic blocks are defined very strictly according to this model. Each instruction belongs to exactly one BB. Jumps and branches end basic blocks and the targets of branch and jump instructions start new ones. Due to the added semantic meaning in source code and the global view available, compilers can easily determine all jump targets, so they use this type of basic block in code generation and optimization. On assembly level arbitrary instructions can be targets of jumps from anywhere in the binary, so the end of static basic blocks is not identifiable without a global analysis.

To avoid this overhead the concept of *dynamic* basic blocks can be employed. Their definition follows the actual control flow at runtime. Only control flow altering instructions can end a dynamic BB.

Therefore the extent of a specific basic block is easy to identify, but jumps could theoretically target instructions inside of a basic block. This contradicts the definition of a basic block as single entry, single exit. As a solution to this problem jumps into the middle of an existing basic block simply starts a new basic block that overlaps with the existing one. This means any instruction can be part of arbitrarily many basic blocks.

Dynamic basic blocks are ideal as the granularity for DBT as they do not require any non-local analysis. Unless otherwise mentioned this is the type of basic block assumed in further parts of this thesis.

2.2 LLVM

LLVM started as a research project by Chris Lattner and Vikram Adve at the University of Illinois [LA04]. Their goal was to provide a modern compiler infrastructure based on a low-level language using single static assignment.

A popular model of the compilation process involves three stages. A front end converts the source code to a language dependent representation, most of the time based on an abstract syntax tree. Next, optimizer passes are run on this tree or an intermediate representation generated directly from the tree. The back end is then responsible to create the machine code from the optimized IR. Most importantly it has to generate code that accurately represents the computation the source code intends to do. But it also has to select good instructions using the hardware as efficiently as possible. A typical back end implementation performs steps such as register allocation, instruction selection and instruction scheduling.

Separating these phases from each other brings major benefits. One of them is that it allows easy retargetablity. Supporting a new programming language just requires a new front end and can reuse all existing code in the optimizer and back ends. Similarly, a new back end adds support for a new target architecture for all programming languages supported by an existing front end.

While this design has clear advantages, many of the popular compilers do not implement it. Other compilers, most notably GCC, are implemented according to this model, but are developed as a monolithic application. This prevents embedding them into other applications or using them as JIT compilers. LLVM wanted to develop a system from scratch strictly based on this approach. The components of it were designed to be completely modular. [Lat11]

The initial implementation of LLVM was focused on the C and C++ programming languages and only supported x86 and SPARC as target architectures. But soon its modularity made it worthwhile to be adapted by new programming languages. Nowadays LLVM provides the main (often only) back ends and optimizer for many programming languages, including Rust, Swift and Julia. Similarly, many architectures have a supported back end, which gives the supported programming languages a wide variety of supported targets.

2.3 Instruction Set Architectures

2.3.1 RISC–V

RISC–V is an ISA that was initially developed by Prof. Krste Asanović and his graduate students Yunsup Lee and Andrew Waterman in 2010 as part of the Parallel Computing lab at UC Berkley. [Wat+11] One of the main design goals was to provide a free and open source ISA, that is fitting for direct hardware implementation, while avoiding to optimize for any specific micro-architectural approach (in-order, out-of-order, micro-coded, . . .) or hardware implementation (ASIC, FPGA, . . .). Instead they wanted to propose an ISA that can work well under a wide variety of use cases and implementations.

As such, RISC–V is designed in a modular way consisting of many different standard extensions to the base integer (I) instruction set. These extensions include support for integer multiplication and division (M extension), single and double precision floating point numbers (F, D) and atomic memory operations (A). Further, RISC–V is available in 32-bit and 64-bit variants and there even exists a draft for an extension supporting an integer width of 128 bits.

This allows to tailor the hardware to the specific use case. Small and cheap embedded micro-controller can implement only the base 32-bit I instruction set. On the other hand a CPU intended for a desktop PC will probably implement 64-bit IMAFD, which is then also called RV64G, and maybe even more extensions.

RISC–V instructions are typically encoded using 32 bit instruction words. But it uses a variable length encoding that is currently defined for instruction lengths from 16 to 176 bits in 16 bit increments. Bigger lengths can be incorporated into the scheme if needed. [And19]

2.3.2 Differences between x86 and RISC–V

With the goal being to translate from x86 binaries directly to RISC–V the following section will take a look at some of the architectural differences between x86-64 and RISC–V64.

In contrast to x86, which follows the complex instruction set computer (CISC) design principle, RISC–V as its name suggests is a reduced instruction set architecture. These opposing design strategies give rise to a bunch of major divergences in features between these ISAs. In the following, we will list some of these and also give a first view how this might impact a translation from x86 to RISC–V.

First of all, RISC–V is a load-store architecture and does not allow memory operands for arbitrary instructions and instead uses specialized instructions for interfacing with memory. For translation this will require to separately insert instructions for executing the operation and, if required, the memory accesses, similar to how modern x86 CPUs translate those instructions to micro operations corresponding to these operations.

Additionally, RISC–V only has support for the simple addressing mode consisting of base and immediate displacement. In the default instruction set and the common extensions there are no instructions that generate addresses according to the x86 format consisting of base, index, scale. To have access to such instructions the address generation part of the bit manipulation extensions (B) is required, which is not the case for the processor we tested on. Instead at least two instructions (shift to scale and addition of base and scaled index) are required to support this.

Both of these mismatches cause significant overhead in instruction count and hence execution time, when using memory operands.

Further, of the four operand sizes that x86-64 has (8, 16, 32 and 64 bits) only 64 bits is supported for all instructions on RISC–V. An execution width of 32 bits is only supported for some instructions, but contrary to the zero-extension to full-width 64-bit registers on x86, RISC–V does a sign-extension in those cases. RISC–V does not have instructions for 8 or 16 bit computations. The special semantics of partial register writes for these instructions on x86 are not supported either and need to be emulated by masking and inserting the new bits. Again this causes significant overhead to recreate the needed semantics.

Another major difference is the omission of condition flags from RISC–V. On x86 these are updated on the execution of almost all of the basic arithmetical and logical operations, e.g. `add` and `or`. They indicate various basic conditions of the last operation such as signed and unsigned overflow, zero or negative result. These flags can be used as the conditions for jumps or to conditionally set a register. But after most instructions flags are actually unused and directly overwritten by the next instruction that sets them.

RISC–V on the other hand uses direct compare and branch or compare and set instructions. Calculating flags explicitly after instructions is possible, but incurs an overhead of at least 6 RISC–V instructions for each x86 instruction that sets flags. This is one of the major overheads in translated x86 code and therefore optimizing away unused flag computations has great potential.

But there are also features of RISC–V that make translation easier. RISC–V has 32 general-purpose registers (with x0 hardwired to 0) instead of the 16 from x86. This makes it possible to directly map each x86 register to one in RISC–V, while still leaving some registers free to be used for temporary values or internal purposes of the translator. In contrary to the 2 address form used in x86, where the destination is also the first source, RISC–V uses the 3 address form, so every register operand and the destination is separate, which allows to omit register moves on RISC–V in many cases where they would be needed on x86.

3 Overview

3.1 Hybrid Binary Translation

Arancini is a hybrid binary translator utilising both static translation before running the program and dynamic binary translation at runtime. With this approach the advantages of static translation can be exploited, while guaranteeing to get a translation covering all execution paths.

Translating completely ahead of time is not possible mainly for the following two reasons. The way the x86 instruction set is encoded uses variable lengths where a single instruction can be encoded by anywhere from a single to 15 bytes. There is no common bit pattern that identifies start or end of multi byte instructions, so determining the start of instructions is not always possible without actually executing the code.

As an example the instruction `mov al,0x50` is encoded as `0xb050`, but if instead the instruction pointer is one higher the `0x50` is decoded to `push rax`, a completely unrelated instruction. Additionally, data might be mixed into executable binary code, causing instructions to not be consecutive, but this is hard to detect without running the instructions and seeing what is executed.

So static translation is only possible if instruction boundaries are known before the execution. In many cases these can be inferred from information available in the binary meta data, such as symbol tables, but due to the existence of indirect jumps and jump tables this does not always give a full picture. There is always the possibility that on execution program counter values are discovered that have no pre-existing translation, which then necessitates a translation at runtime.

The second reason is not specific to x86 and applies to any architectures. Some programs might generate binary code at runtime or modify their own binary code. Since that code is not even available before execution it also can only be translated runtime leading to the same effect as above.

In contrast to that, translating all instructions dynamically at runtime allows to discover all execution paths. Therefore this approach supports programs where it is not possible to do a translation ahead of time, but it also has some major drawbacks. The process of translating basic blocks takes a significant amount of time. So on discovery of a new block there will be a delay in execution of the program, which might cause noticeable halts if it happens multiple times in short succession.

In typical programs a large portion of the executed basic blocks are needed independently of program inputs or other external factors that influence control flow at runtime. Regardless of this, due to the dynamic nature of the process these blocks are recreated on every run of the program. So the overhead of the translation process causes increased runtime and consequently energy consumption on all executions. To keep this performance impact relatively low, dynamic translation has to be fast and can't do any sophisticated optimization passes. This adds further overhead in form of possibly inefficient instruction sequences resulting from architecture mismatches.

Combining both of these approaches allows exploiting the advantages of both of them. The static pass can cover a large share of executed code providing good code quality by utilizing extensive optimization passes. Meanwhile the dynamic translation covers all blocks only discovered on runtime and therefore allows the translation to be complete. While it produces worse code and causes runtime translation overhead, it is only needed for a small fraction of translated code.

3.2 Program flow

Arancini operates in two stages. The first stage needs to be run ahead of the execution time to prepare a binary containing the static translations and the runtime environment. A second stage is responsible for any parts of the translation that require intervention at runtime, such as blocks requiring dynamic translation.

3.2.1 Ahead of time phase

Arancini first parses the ELF file header of the binary to determine necessary meta information. The symbol table located in this part of the binary file is examined to discover executable code. The contained function symbols are assumed to fully cover all executable code. On top of that we use the reasonable assumption that the there is no data mixed into the instructions of function symbols, so all instruction are always encoded right after the previous one without any gaps.

To decode the binary instructions in each of these functions the Intel XED library is used. As this is the official library for encoding and decoding x86 instructions developed directly by Intel, this guarantees support and correctness for all current instruction set extensions. Any of the fields encoded in the different parts of the x86 instruction are easily accessible in this.

Every instruction is converted separately into IRancini the intermediate representation (IR) used internally by Arancini. An intermediate representation is necessary, particularly due to the complexity of x86 instructions, which often perform multiple actions in one. One example of this are memory operands, which combine operations and memory access in one instruction.

The generated IR is then used as an input to the static translation path. There IRancini is translated to LLVM IR. While it might seem a bit counter intuitive to translate between different IRs, this is not uncommon. As an example the official Rust compiler (`rustc`) uses multiple different IRs (HIR and MIR) tailored specifically to Rust before converting to LLVM IR.

Using LLVM for the static translation has many advantages. It provides extensive optimization passes that can be used without high implementation effort. In addition to that, it takes care of all the necessary steps for code generation, which includes tasks typically handled by a compiler, such as register allocation, instruction selection and encoding. LLVM has support for a high number of architectures, so naturally the static translation supports many targets without manual work.

The next step is the generation of a new binary that can run on the target architecture. The machine code generated by LLVM is linked with the Arancini runtime library. Additionally, any program headers and any loadable data sections of the source binary need to be included to allow the runtime to setup the memory regions similar to how the operating system loader would usually do it. The output is a self sufficient binary that on execution uses the Arancini runtime to execute the translated code.

3.2.2 Runtime

In addition to the translated code there are also various tasks that need to be fulfilled at execution time. The runtime provides all these required services.

At the start the runtime is responsible for setting up all memory regions, including an emulated guest stack. It allocates a memory region for the guest and loads all data sections of the original binary into this region.

Another service it provides is emulation of system calls. In many cases this is just a mapping of the respective system call from source to target platform and only requires marshalling the register arguments and choosing the correct system call number for the target. But other system calls, for example the ones creating threads or allocating memory, require special handling to work in our special environment.

The usual program flow should normally mostly be covered by the statically translated code generated via LLVM. In this case the runtime system is not involved in selecting the next block to execute. But as explained before not all code paths might be covered by the code discovery of the static pass. Any of these unknown code paths are translated dynamically at runtime if they are needed.

If the static code discovers a program counter it does not know about, it asks the runtime system to execute it. For all dynamically translated blocks the translation cache maps the value of the emulated program counter to the host address of the block with the translated code. So for program counters which were previously translated, the runtime finds the corresponding translated block in the translation cache and jumps to the associated code.

3.2.3 Dynamic translation

For program counters not seen before the runtime triggers the start of the translation process. This works very similar to the static translation. The same front end that is also used for the static translation, converts the machine code first to assembly and then IR at runtime.

LLVM is a very heavy library, that has a big footprint. It includes a module for just in time code generation, but the performance impact when using it is very high. At the same time the relatively limited view at runtime using only basic blocks means many of the available optimizations are not really useful. Additionally, performing extensive optimizations takes a considerable amount of time. Blocks not covered by the static path should only be executed rarely, so this high up front cost is unlikely to be recouped. Therefore using the LLVM back end at runtime is not feasible and the decision was made to use custom back ends for dynamic translation.

Consequently, the dynamic back ends have constraints in that regard. In general they need to produce a translation fast and should not have a high memory usage. These are factors that influence the run time of the program significantly.

Therefore back ends convert the IR directly to assembly without a complex intermediate layer, like LLVM. All of the steps associated with code generation need to be performed directly by the back end. This involves converting the generated assembly to machine code, as well as allocating the available hardware registers to the virtual registers used in the translations.

Apart from these performance considerations the only other bounds are a correct lowering of the IR into native machine code. There is a significant degree of freedom in the implementation strategies that can be used to achieve this goal. For instance, the machine code could be generated in a single pass, or multiple passes can be used for example to assign physical registers after all instructions are available. These choices depend on a variety of factors, such as the level of complexity intended and the target architecture, among others.

4 Design

The following chapter presents some of the core design decisions and strategies that went into the development of Arancini with a focus on those influencing the RISC–V back end.

4.1 IRancini

Arancini uses a custom intermediate representation called *IRancini*. To be able to understand the process of translating instructions the following section will explain the general concepts of this IR.

As shown in Figure 4.1 IRancini is a graph-based language. It consists of two main types of nodes, called *action* and *value* nodes.

Action nodes are totally ordered among each other. They provide the execution flow as indicated by the blue and red edges. This type encompasses all nodes, which write values to memory or registers or influence control flow.

The remaining nodes perform tasks such as reading registers, accessing memory or performing arithmetical or logical operations. These operations produce output values which are provided by so called *ports*, hence the nodes earn their name as value nodes.

A node can have multiple inputs, corresponding to the different operands that are needed. While each input needs to be connected to an output port of another node, only a single port of any node needs to be used for the node to appear in the graph. Value nodes are only ordered by the usage of the output ports they provide, these connections are indicated by the black arrows in the diagram.

To give some structure to the translation process and subdivide the graph into manageable parts each function for static translation or basic block for dynamic translation is put into an independent *chunk*. Chunks are further divided into *packets*, which are shown as the boxes in Figure 4.1. These each represent a single instruction of the source binary.

4.1.1 Layout of packets

The general structure of packets is visible in Figure 4.2. It follows a very simple execution model of CPU instructions.

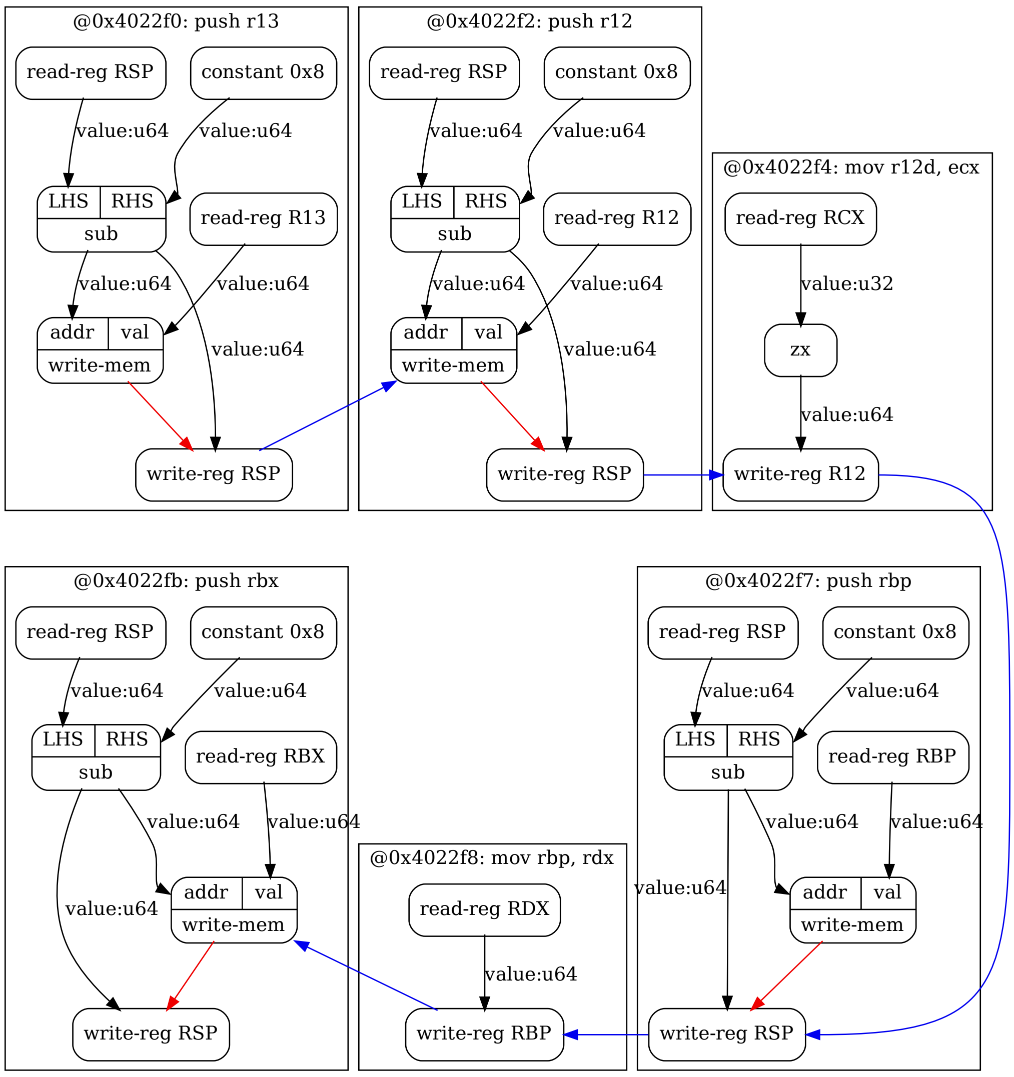


Figure 4.1: IRancini example

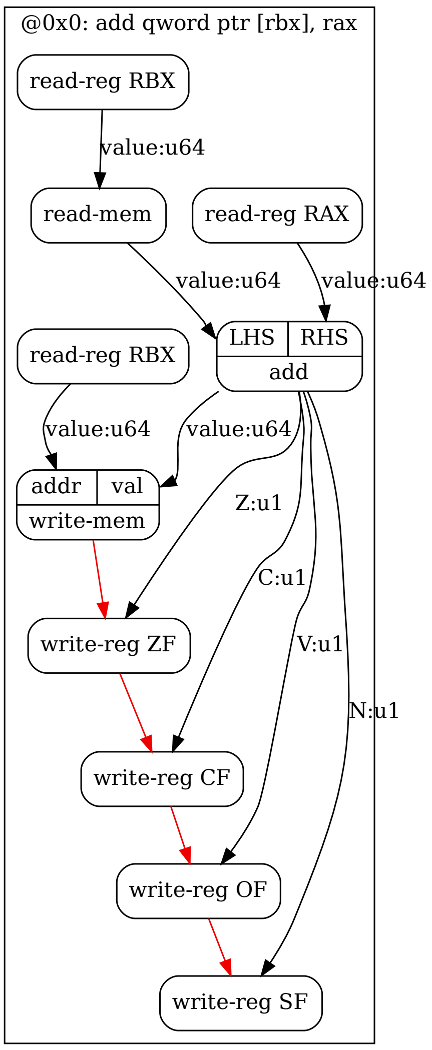


Figure 4.2: Packet layout example

Inputs

Firstly, all input operands are read in. There are three types that need to be considered here: Constants, register and memory operands. All of these are expressed explicitly in the IR by a specific node.

Registers are not direct operands and instead are read from the register file explicitly with `read-reg` nodes, this allows easier implementation on target architectures, which do not have enough registers to map all of the registers in the source ISA. In the case of x86 specifically, the status flags are represented as separate registers and as such are read like this as well if needed.

For accessing memory `read-mem` nodes obviously require an address to access, this is expressed as an explicit input port. Base and index registers are read explicitly and then combined with scale and displacement using addition and multiplication nodes.

Transformations

Next in the graph are all nodes transforming the input values. Depending on the specific instruction this can be a single node, e.g. for an `add`, like in the shown example. It can also expand into multiple nodes, e.g. to combine the individual flags forming composite conditions. As an example the "above" condition is true if both sign and overflow flag are not set, so they are first negated and then combined using an `and` node.

For more complex instructions this part of the IR can even form loops. One example of this are the x86 string instructions using the `rep` prefixes, which means the operation is performed either a predetermined number of times or until a condition is met.

Obviously, there are instructions which do not actually modify any values, such as register moves, or memory reads/writes. This means not all instructions include nodes of this category.

Generating the output values of flags is handled in this stage as well. To implement this the arithmetic and logical nodes have four separate output ports, carry, overflow, negative and zero, corresponding to the carry, overflow, sign and zero flags in the x86 ISA.

Outputs

Any modification of the state of the emulated CPU is done explicitly. So the last type of nodes in a packet are the ones writing the possibly transformed values to this state.

Depending on the specific translated instruction there are multiple possibilities for the destination. A register destination operand results in a write to the register file. Alternatively, if the instruction has a memory destination operand, there will be a node that writes to the memory location referenced in the instruction. Similar to the `read-mem` nodes, the `write-mem` nodes also have the address they access as an explicit input, in addition to the value they should write.

Since flags are just represented as regular registers, all written flags also require their own `write-reg` node. In addition to the clarity this provides it also allows specifying special semantics for the flags depending on the operation to be performed. There are several reasons why this option may be needed. For example, some instructions do not write all flags, like `inc` and `dec` that leave the carry flag unmodified. Additionally, certain instructions unconditionally clear or set certain flags, e.g. the bit-wise logical instructions (`and`, `or` etc.) that clear overflow and carry flags regardless of their input values.

Instructions that influence control flow in the source binary, also need to influence the emulated control flow of the generated IR. In accordance with the way other modifications of the state are done, changing control flow happens by explicit `write-pc` nodes. The program counter is also part of the register file, so the regular `write-reg` nodes would be sufficient for this fuinctionality. But having a separate node makes it easier to apply the needed special handling associated with terminating and switching basic blocks.

Deviations from this model

Atomic nodes are one exceptions to this model. Any atomic instruction has exactly one memory operand, which is both source and destination. They also can have a second source, which can be a register or an immediate.

To express the intent of executing the operation atomically the three steps of read, transform and write back are all implicitly included in an atomic node. The semantics associated with these nodes therefore include these three steps. First the old value is read from the memory location. Then it is transformed according to the operation to be performed, which can be a normal arithmetic operation or a special atomic action, like compare-and-swap or exchange-and-add. Lastly, the new values need to be written back to memory. Some of these instructions also write the value that was in the memory location prior to the transformation back to the source register, which is also done implicitly.

Some special instructions, like `syscall`, are implemented completely by the runtime, so they also do not fit into the three stage model. Instead they are represented by a single `internal-call` node that represents a switch to the runtime system.

4.1.2 Types in IRancini

IRancini is a typed language. The possible type classes include signed and unsigned integers, floating point numbers and vectors combing multiple values of the same type. To fully specify a type the type class is combined with a bit width.

For all nodes their meaning depends on what type they operate on. So, opposed to how most ISAs work, IRancini does not have different opcodes depending on which type is used. As an example a single precision floating addition, 16 bit integer addition and element-wise addition of four 32 bit integers are all expressed using the binary arithmetic node with an addition operation. The only difference is the type of the inputs, which also directly determines which type the value output port has.

The special semantics of x86 instructions, that do not operate on the full 64-bit width of the general purpose registers are not transferred directly to the IR types. Instead these are expressed expressed explicitly. The operations are performed on their specified bit widths and then the register value is built based on that.

For 32 bit values this means their translation includes an explicit zero extension to 64 bits. The 8 and 16 bit variants explicitly read the old 64 bit register value and use a `bit-insert` node to insert the updated bit range into the full width register without touching the remaining bits.

Since flag values are boolean, they use a special type. It is defined as an unsigned integer with a width of 1 bit. This type is only used for flags and the composite conditions built from flags. So checking for this type allows identify the usage of flags and possibly optimize based on that.

4.2 From IR to machine language

In the back ends we translate the IR into assembly and further into machine code. In the specific case of the RISC–V back end the program flow is as follows.

The interface between the IR and the back end are the action nodes. Since these fully describe the program flow, only nodes incorporated into a sub-graph ending at an action node influence the program state. So to generate the machine code the graph is walked from the action nodes back to the value sources.

Lowering a node therefore first materialises all of the nodes it references as an input port. These nodes can again have inputs, which need to be lowered, so this process continues recursively until a node without inputs is reached. Currently this is either a constant or read-reg node.

Nodes can have multiple output ports and the same port can be used by multiple nodes. Generating the code implementing a node multiple times can cause correctness issues, because involved register values might have changed by an interleaving `write-reg` node. For this reason every node is only lowered exactly once and the output register of a port is reused for all references.

The translation process itself operates on singular nodes at a time. Not taking into account cross node interactions in the translation will produce less optimal instruction sequences. But at the same time it massively reduces the complexity involved in the translation. The small number of node combinations that occur often in practice and can easily be merged. These are implemented as special casing in the node translations.

A prominent example for this are constant operands to arithmetic operations. Apart from only being used for translating the corresponding operations of x86, these are also used in many other places, e.g. to add offsets on jumps or memory accesses. So trying to optimize this provides good instruction savings. RISC–V also has instructions that directly use immediate constants as one of the inputs. So in the translation we check if the input ports are from constant nodes. If they are and the immediate fits in the instruction encoding, it is not separately generated and instead integrated in the translation of the arithmetic node.

A crucial step in translation that is also handled inside the back end is generating binary code for the target platform. Three components are responsible for enabling this.

4.2.1 Instruction Encoder

First of all there is the instruction encoder.

Different instructions require different operands. In RISC–V there can be up to three input registers and one output register. Additionally, a lot of instructions need an immediate operand.

There are different purposes for immediates with different needs: constants for arithmetic, offsets for memory references, jumps or branches and immediates to load into registers directly. Depending on the additionally needed register operands trade offs have to be made on the range of the immediates. So there is different formats to encode the immediates into the instructions. To accommodate these different immediate formats and register operands there is a number of different instruction formats.

RISC–V instruction encodings were designed from a hardware first perspective keeping the different fields in as little different places in the encoded instruction as possible. This reduces the number of muxes necessary in the instruction decoder for choosing the source of values. A byproduct of this approach is that immediate fields in the encoded instructions are not always in consecutive bits and they often are scrambled in order to increase overlap between the instruction formats. [And19]

The encoder exposes a function for every different RISC–V instruction, taking the registers and immediates as inputs. The immediate formats require quite a bit of work on the encoder side to split up and scramble them into the correct bit order. On the other hand the RISC–V encoding scheme has all been defined from the beginning, so there are no expansions that were added afterwards that use a different encoding scheme, like the various prefixes that are used on x86. Having only a limited number of instruction lengths that are used, also simplifies the encoding process by reducing the book keeping effort.

One of the major bottlenecks in current CPU architectures is instruction fetch bandwidth. Having each instruction require a full 32 bits in encoding space amplifies this issue. Therefore RISC–V includes an ISA extension for compressed instruction encodings in the C standard extension. These encodings are all 16 bits long and map directly to the normal 32 bit instructions. The most often used instructions are provided with 16 bit encodings by reducing the immediate ranges and encodable registers. In this way about 50 to 60% of instructions in a typical program can be replaced by instructions of half the size. [And19]

Similarly to the regular 32 bit encodings the compressed encodings also focus on increasing the overlap between different instruction formats to reduce the amount of hardware needed to decode the instructions. Due to the limited amount of available encoding space the used immediate formats are more focused on the actual use case, which results in more different formats. To still guarantee as much overlap as possible the immediate fields are again distributed over split ranges and scrambled.

The used assembler checks if the given register and immediate operands allow the compressed form to be used and automatically chooses these encodings in those cases. This increases the complexity by a fair amount requiring to check all operands for the conditions of the possible compressed instructions. But the savings in memory usage make this worth it.

4.2.2 Instruction Memory Management

Encoding human readable assembly into machine code is only the first step to getting executable code pieces. Another important step is putting the encoded instructions into executable regions of memory. Two components handle the memory management.

Allocating memory via the usual means in programming languages, for example `malloc` in C or `new` in C++ returns memory that is not executable. Executable memory needs to be allocated directly from the operating system. Therefore all the sophisticated algorithms to optimize allocations used in the library implementations of memory allocators are not available to us. On top of that the `mmap` system call used to obtain executable memory on Linux systems works in page granularity.

As allocating new memory pages is very expensive usage of them has to be minimized. Minimizing system calls will also help by reducing the number of context switches to the operating system. Achieving this requires a more advanced allocation strategy than using a separate allocation call for every request.

Therefore the used allocation strategy follows the memory pool or arena approach. At the first request a large number of pages is allocated from the operating system. Our specific use case does not require to free allocated memory blocks or resize memory blocks once the translation was finished. So the only state that has to be kept is the size of the pool, the size of the current allocation and a pointer to its start. Therefore making a new allocation is as simple as calculating a pointer that is past the end of the last allocation and updating the state to the size and base of this new allocation.

Writing the machine code to memory is handled by the second component, called the *machine code writer*. It handles all necessary steps for writing code to executable memory. This includes calling the necessary functions of the allocator for creating new allocations, enlarging the allocation and at the end shrinking it down to not waste unused memory.

5 Implementation

In this chapter we will give a detailed look at the RISC–V back end for Arancini. It will be shown how code is generated. A detailed explanation of the selected lowering for nodes will be provided.

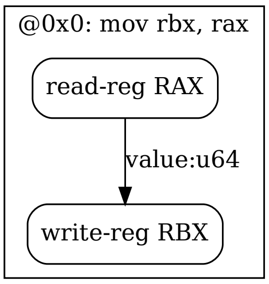
5.1 Translating from x86 to RISC-V via IRancini

5.1.1 Move between two registers

Moves between registers are the most simple instructions available in the x86 ISA. As a start it is a good introduction to translated code to take a look at this type of instruction.

Both the read-reg and write-reg nodes are translated by a single instruction. Since the instruction operates on the full width register load (`ld`) and store (`sd`) double word instructions are used. To access the register file the memory references are indirect via the `fp` register. This register is defined to always contain the base address of the struct storing the x86 CPU state of the simulated processor.

The `fp` register is defined by the RISC–V specification to be the frame pointer, holding a reference to the top of the current stack frame. But, like the `RBP` register on x86, this usage is not mandatory and the register can be used as a regular general-purpose



(a) x86 assembly and IRancini

```
ld t0, 8(fp)
sd t0, 32(fp)
```

(b) RISC–V assembly

Figure 5.1: Translation of a register move

x86 flag	RISC–V register
zero flag (ZF)	`s8`
carry flag (CF)	`s9`
overflow flag (OF)	`s10`
sign flag (SF)	`s11`

Table 5.1: Mapping of x86 flags to RISC–V registers

register. This is reflected by the second mnemonic that was assigned to this register in the specification: `s0`, which stands for callee saved register 0.

We overloaded the meaning of the frame pointer to be the base pointer of the CPU state. This provides an easy to remember way to access the CPU state. It also makes references to the state easy to recognize in the generated assembly which facilitates debugging.

An alternative to mapping registers to memory would be to assign a RISC–V register to each x86 register. All accesses to the x86 register would be translated directly as accesses to the corresponding RISC–V one. So this instruction could simplify to a single instruction, a register move. This optimization will provide a very noticeable speed increase, since it would allow to omit at least two memory accesses on most instructions. On the contrary it increases complexity and introduces a bunch of possibilities to get correctness issues. A decision was made to initially omit this and focus on a working implementation.

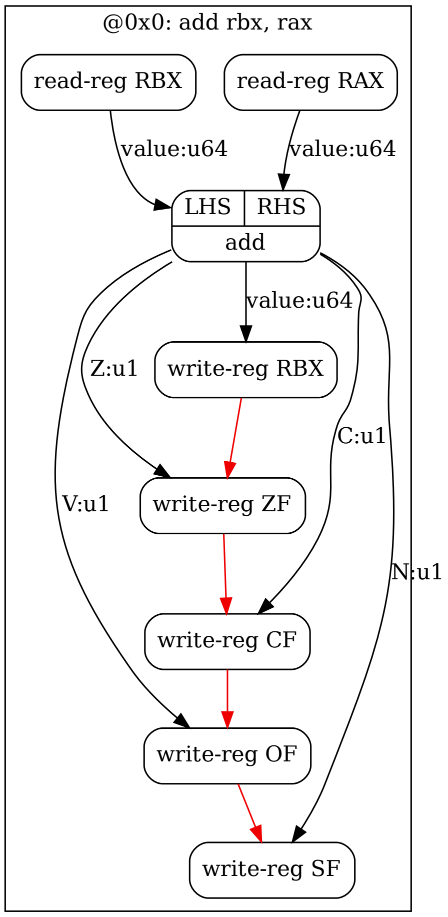
5.1.2 Addition with register operands

Taking the complexity up slightly Figure 5.2 shows an addition instruction. Even such a simple instruction already involves a decent amount of work.

The generated assembly can be broken down into a couple blocks separated by line breaks. First, we see the register loads that we already know. Next, as expected we translated the addition as an `add` in RISC–V assembly.

After that the next block calculates the flag values. We assigned each flag a register according to the mapping in Table 5.1. This makes it easy to track flag values in the assembly. It also simplifies code generation significantly because it allows to leave out register allocation for the flags. The register that contains a given flag is not dependent on the context, so it does not have to be tracked. The specific registers were chosen arbitrarily in ascending order of the writes in the IR.

The first flag calculated here is the overflow flag. This flag indicates that the result of the previous operation could not accurately be represented, if we interpret the operands



(a) x86 assembly and IRancini

```
ld a0, 32(fp)
ld a1, 8(fp)

add s1, a0, a1

sltz s9, a0
slt s10, s1, a1
xor s10, s10, s9
sltu s9, s1, a1
seqz s8, s1
sltz s11, s1

sd s1, 32(fp)
sb s8, 392(fp)
sb s9, 393(fp)
sb s10, 394(fp)
sb s11, 395(fp)
```

(b) RISC–V assembly

Figure 5.2: Translation of an addition of two registers

as signed integers. One way to calculate this specifically for an addition is to look at the relations of input and output values. Mathematically speaking the result of an addition can only be less than either of the addends if the other addend is negative. If this is violated for our instruction, an overflow has occurred. So the flag needs to be set when exactly one of the two conditions, "result is less than second addend" and "first addend is less than zero" is true.

To implement this in RISC–V, we use a `sltz` (set less than zero) instruction to check if the `LHS` operand is negative. The `s9` register normally represents the carry flag. Since this flag was not yet written in the translation, we use this register as a temporary. Utilizing unwritten flag registers for temporaries is a common theme that will continue to appear in more translations. Next, a `slt` (set less than signed) instruction sets the OF register if the result is less than `RHS`. To complete the condition a `xor` achieves the required property that exactly one of these two is true.

The other flags can all be calculated in a single instruction. The CF needs to be set on an unsigned overflow, which occurs if the result is less than any of the operands. Accordingly the `sltu` (set less than unsigned) instruction is used. A result equal to zero sets the ZF, which can is achieved using the `seqz` (set equal zero) instruction. Lastly, the SF is set on a negative result, which is computed by a `sltz` (set less than zero) instruction.

In the last block the outputs are written back to the register file implementing the `write-reg` nodes from the IR. For the regular register we have seen that already in the last example. The flags are stored in the register file each as separate single byte values, so this uses store byte (`sb`) instructions.

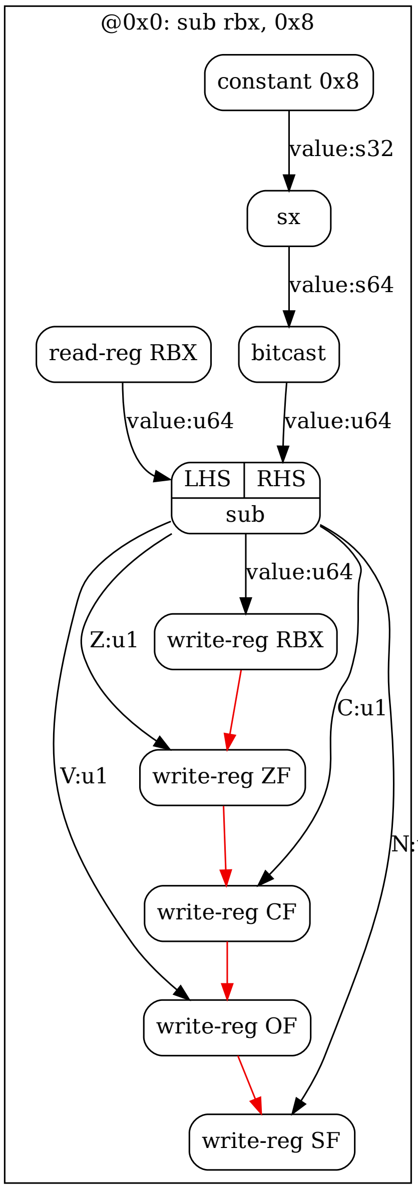
5.1.3 Subtraction with immediate operand

This operation is very similar to the addition explored in the last section. But there are some very interesting differences.

Figure 5.3 shows that the IR uses casts to bring the immediate into the correct type. But this is not reflected in the generated assembly.

RISC–V has variants of the basic arithmetic and logical instructions where one of the operands is an immediate directly encoded in the instruction. The range of these immediates is limited to 12 bits using sign extension to get to the full width value. So the possible values are between -2048 and 2047. If a value in this range is used, we recognize this and instead of generating the immediate into a register directly use the corresponding immediate instruction.

There is no subtract immediate instruction, because the add immediate instruction with the negated immediate covers this already. The range on a subtract immediate is thus changed slightly going from -2047 to 2048. Accordingly the generated code uses

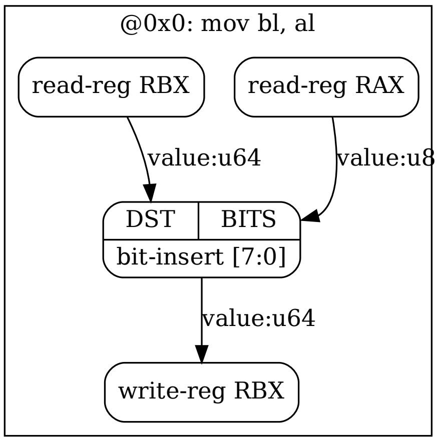


(a) x86 assembly and IRancini

```
ld a0, 32(fp)

addi s1, a0, -8

sltu s9, a0, s1
slt s10, s1, a0
xori s10, s10, 1
seqz s8, s1
sltz s11, s1

sd s1, 32(fp)
sb s8, 392(fp)
sb s9, 393(fp)
sb s10, 394(fp)
sb s11, 395(fp)
```

(b) RISC–V assembly

Figure 5.3: Translation of an subtraction of a register and an immediate



(a) x86 assembly and IRancini

```
lb s1, 8(fp)
ld a0, 32(fp)

andi a0, a0, 255
andi a1, s1, -256
or a1, a1, a2

sd a1, 32(fp)
```

(b) RISC–V assembly

Figure 5.4: Translation of a partial register move

an add with the negated immediate.

In comparison to the addition the conditions of overflow and carry flags are inverted. So we implement the carry using a `sltu` with the source registers switched.

The correct result on a signed subtraction is smaller than the minuend whenever the subtrahend is positive. When the result does not satisfy this there was an overflow. We translate these conditions similar to the addition. A `slt` checks if the result is smaller than the `LHS`. Then this value is combined using a `xor` with the comparison of `RHS` > 0. Since we know the value of `RHS` operand, there is no need to dynamically check it and we just use a `xor` with an immediate value.

Apart from that this translation is basically the same as the register-register addition seen previously. Sign flag and zero flag are computed in the same way. The write back also looks the same.

5.1.4 Move between partial registers

One of the features of x86 includes is the possibilities to execute almost any instruction on different register bit widths. The available widths include 8, 16, 32 and the full 64 bits. For the 8 and 16 bit widths the x86 behavior demands that writing to a register preserves the higher order bits. Therefore the IR reflects this by an explicit bit-insert node of the 8 or 16 bit value into the bigger register. Figure 5.4 shows this for the very simple example of an 8 bit register move.

The actual implementation in RISC–V assembly is pretty straight forward. For the 8 bit value we use the appropriate sized `lb` (load byte) instruction.

The simple instruction set of RISC–V does not contain a single instruction to perform a bit insert, so we have to emulate it with a sequence of multiple instructions. This example is a special case of a bit insert since the bit range in the destination starts at 0 and the bit length of 8 is small.

Therefore the masks to isolate the bits to be inserted and the ones to keep in the destination, both fit into the 12 bit sign extended format of the immediate instruction type. A mask of 255 has the lowest 8 bits set, so the first `and` clears bits 8 to 63. Applying sign extension -256 has all but the lowest 8 bits set, so the second `and` clears bits 0 to 7. The non-overlapping bit ranges are then combined using a bit-wise or.

As usual in the last step the resulting full width register value is stored back to the register file.

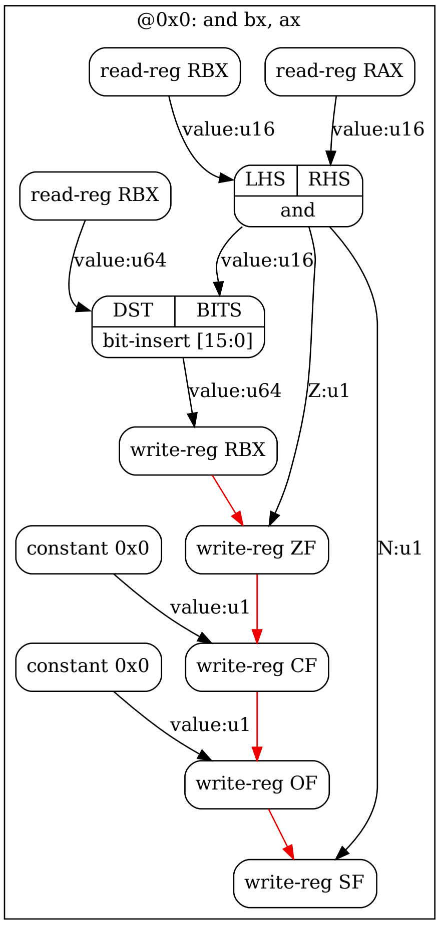
5.1.5 Arithmetic on partial registers

If the usage of partial registers is combined with any operation, this necessarily involves a large amount of instructions. Figure 5.5 shows the implementation of a 16 bit `and` operation. So the register loads are implemented using the 16 bit `lh` (load half word) instruction.

RISC–V only has comparison instructions operating on 64 bit width. So we have to calculate flags using the same comparisons as the 64 bit instructions, even though only the lowest 16 bits should influence the result. Therefore to utilize these instructions the 16 bit values need to be sign extended to the full width. If the values are properly sign extended before performing the computation, the value resulting from the `and` will preserve this property (the same is true for `xor` and `or`).

To ensure this, we introduced a convention to the RISC–V back end that all partial size values in registers are kept sign extended across node boundaries. Choosing this behavior also simplifies many other translations because sign extension is the intended behavior of RISC–V and the ISA is built around this. For this reason, loading register values from the register file uses the `l(b|h|w)` instructions that sign extend and not the zero extending variants `l(b|h|w)u`.

The length of the bit insert in this case is 16, so the masks cannot fit into the immediate of an `andi`. Therefore the bit insert has to be implemented differently. We generate the mask for isolating the higher order bits in the destination using the `lui` (load upper immediate) instruction, which loads its 20 bit immediate into bits 12 to 31 of the register and then sign extends the value to 64 bits. Preparing the bits to insert uses a combination of left (`slli`) and right (`srli`) shifts. First of all this is the standard way to zero extend 16 and 32 bit values in RISC–V. Additionally, it allows to do a zero extension of arbitrary length values and, if needed, the left shift necessary to offset the value to the right bit position in just two instructions.

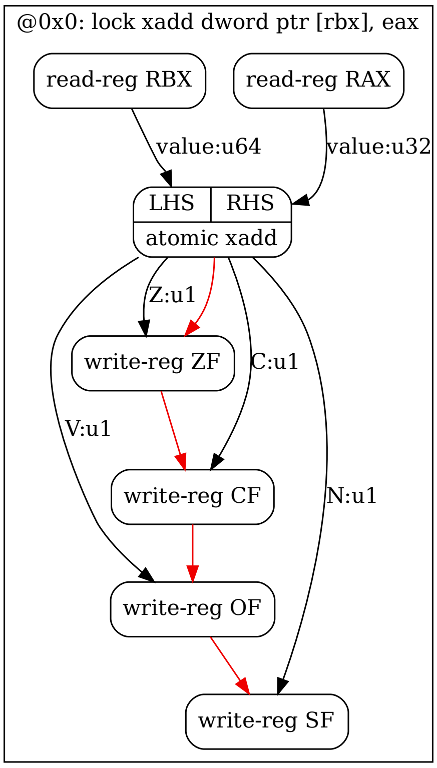


(a) x86 assembly and IRancini

```
ld s1, 32(fp)
lh a1, 8(fp)
lh a2, 32(fp)

and a0, a1, a2

seqz s8, a0
sltz s11, a0

lui a5, 0xffff0
and a3, s1, a5
slli a4, a0, 48
srli a4, a4, 48
or a3, a3, a4

sd a3, 32(fp)
sb s8, 336(fp)
sb x0, 337(fp)
sb x0, 338(fp)
sb s11, 339(fp)
```

(b) RISC–V assembly

Figure 5.5: Translation of a partial register and



(a) x86 assembly and IRancini

```
ld s1, 32(s0)
lw a0, 8(s0)

add a2, s1, t6
amoadd.w.aqrl a1, a0, (a2)
addw s11, a1, a0

#Flag calculation

slli a1, a1, 32
srli a1, a1, 32

sd a1, 8(s0)
#Flag writes
```

(b) RISC–V assembly

Figure 5.6: Translation of an atomic exchange and add

The bit-wise logical instructions unconditionally clear the CF and OF, so we only compute the zero and sign flags. On the write back the x0 register is the source instead for the overflow and carry flags exploiting the fact this register is hard wired to zero.

5.1.6 Atomic add and exchange

A next interesting group of instructions to take a look at are the atomic instructions. In Figure 5.6 we can see a 32 bit atomic exchange and add. As we can see the translation is quite involved.

The atomic semantics require a departure from the typical model used in building packets in IRancini. Operation, memory reads and writes have to be executed atomically and so can't be separate nodes.

In addition to that special atomic instructions exist that also place the value that was in the memory before the operation into the source register. One example for this is the exchange-and-add, which adds the register value to a memory location and places the previous memory value in the register.

RISC–V has special atomic operations that perform add, and, or, xor, max and min of a memory location and a register and load the old value from memory into the destination register. So we use the `amoadd` instruction with word (32 bit) granularity to model the `xadd`. The `LHS` input is the address and `RHS` is the register to add. So both the registers needed for this are loaded from the register file, using the correct width load instructions.

The memory region of the guest program is mapped at an offset. In the translation the `t6` register holds the base address of the guest memory. All translated memory accesses therefore account for this offset by adding it to the address.

For the memory ordering `aqrl` we use, which stands for both acquire and release. Therefore we can guarantee the full sequential consistency required from the x86 source.

`Xadd` sets the flags based on the addition. To calculate these in RISC–V both inputs and the result are needed, so an additional addition into a temporary register is used to obtain the result. In this case we use an `addw` which already does the sign extension from 32 bits. The actual instructions to compute the flags and store them to memory are the same as in the case of a normal addition so I omitted them from the assembly listing.

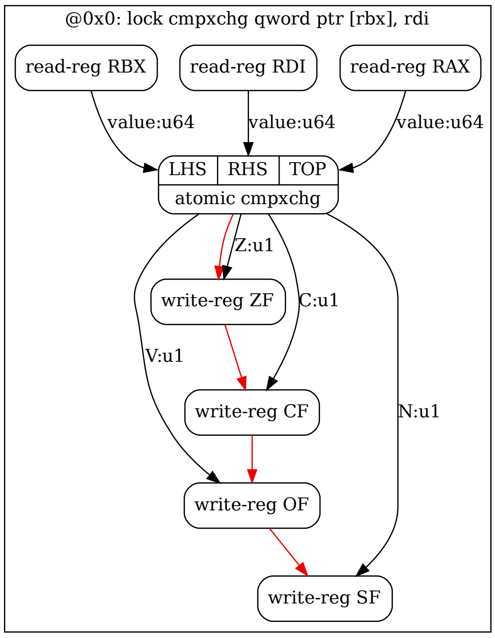
Since x86-64 zero extends on 32 bit register writes, this has to be reflected by zero extending the old value before writing it to the register file. Normally, this is expressed as an explicit zero extension node in the IR. But for atomics the register write is implicit and therefore the zero extension is as well. A zero extension on RISC–V is done via a shift left, shift right combination, so this is used here.

5.1.7 Atomic compare and exchange

A second very important atomic instruction is the compare exchange. RISC–V does not have a compare-and-swap (CAS) instruction. Instead the designers decided to include load reserved (`lr`) and store conditional (`sc`) instructions. Those allow to implement arbitrary operations that appear to be atomic from other hardware threads.

In particular this is the way CAS can be emulated on RISC–V like shown in Figure 5.7. The `lr` loads the value from memory and creates a reservation on the memory cell. Then, the value is compared to the expected value in the second input register. If they are not equal a branch is taken to code that handles the case of failing the CAS.

Otherwise, a `sc` is attempted. This will succeed, if the previously made reservation is still valid. All accesses to a memory location clear reservations on that memory location made by other hardware threads. This ensures that if the store happens, there hasn't been any memory accesses between this store and the preceding load. In case the `sc`



```
ld s1, 32(fp)
ld a0, 64(fp)
ld a1, 8(fp)
add a3, s1, t6

retry:
lr.d.aqrl a2, (a3)
bne a2, a0, fail
sc.d.aqrl a2, a1, (a3)
bnez a2, retry

li s8, 1
li s9, 0
li s10, 0
li s11, 0
j end

fail:
sub s11, a0, a2
#Flag calculation
sd a2, 8(fp)
end:
#Flag writing
```

(a) x86 assembly and IRancini (b) RISC–V assembly

Figure 5.7: Translation of an atomic compare and exchange

failed, its output register is set to a non-zero value and the whole sequence will be retried by branching to the `lr` again.

Two threads executing a sequence of instructions similar to what we use here, could always clear each others reservation. In that case, none of the two threads would make any progress. Therefore, RISC–V includes a forward progress guarantee in its specification. As long as only basic integer instructions are executed between the `lr` and `sc` and they are limited in count, the sequences are guaranteed to eventually succeed. This applies to the sequence used here, so this is a safe way to achieve the goal.

The `cmpxchg` nodes set all flags in accordance with the instruction from x86. In the case of success, we know that both values are equal, so the zero flag is set and all other flags are cleared. This is achieved with the `li` (load immediate) instructions.

In the "fail" branch we need to actually calculate the flag values. Since comparisons are just subtractions without storing the result, we perform a subtraction of the expected and actual values and calculate flags based on that.

After a failed `cmpxchg` the register with the exchange value contains the actual value from memory. So we write the loaded value back to the register file in this case.

5.1.8 Multiplication

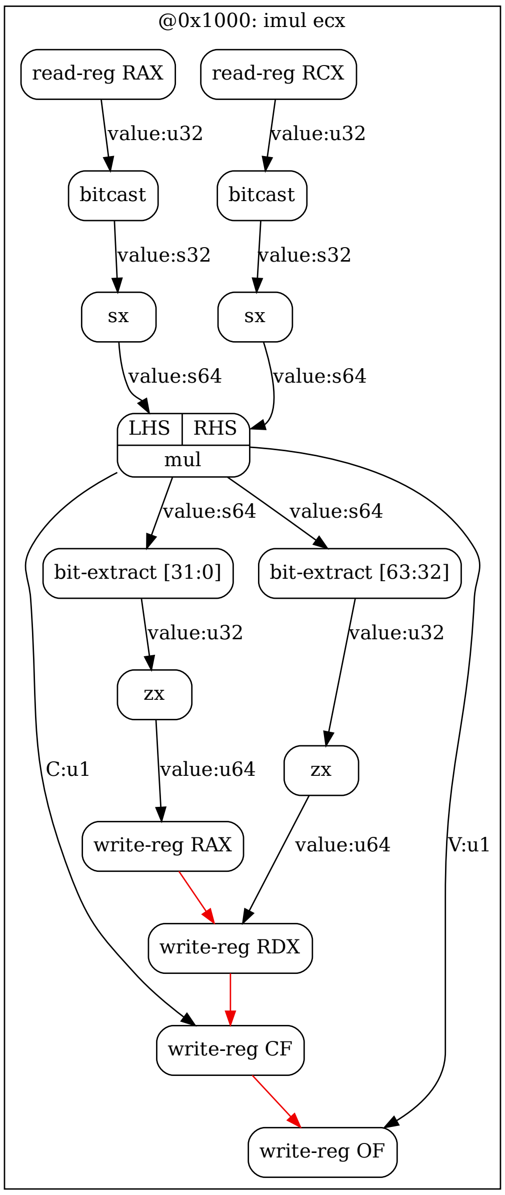
One more possible instruction that shows some interesting translations is the multiply instruction as shown in Figure 5.8.

The multiplication instructions on x86 have some special semantics. In particular there are variants that have one of the operands implicitly defined as `RAX` (or the smaller variants of it). Those perform a widening multiply writing the lower half to `RAX` and the higher half to `RDX`.

In IRancini by default values are represented as unsigned. Therefore, to perform a signed multiply casts have to be applied to the values. Additionally, IRancini does not have a widening multiply, so the operands have to be expanded to double width before the multiply. By the convention of the RISC–V back end all values are sign extended anyway so both of those are actually no-ops.

The next interesting part is again the flag generation. Both the CF and OF indicate that the upper half of the widened result was needed to accurately represent the product. To calculate this we sign extend the lower 32 bits to a 64 bit value using the `sext.w` instruction. Then this is compared to the actual result. There is no instruction to set a register depending on (in)equality of two registers. Therefore, instead we combine both values using a xor, which will clear all equal bits. An inequality check with zero then corresponds to an inequality check between the registers.

The x86 semantics demand the two halfs of the 64 bit result to be placed into two different registers as 32 bit values. Two `bit-extract` nodes implement this. For both



```
lw a0, 8(fp)
lw a1, 16(fp)

mul s1, a0, a1

sext.w s9, s1
xor s9, s9, s1
snez s9, s9
mv s10, s9

sext.w a2, s1
slli a3, a2, 32
srli a3, a3, 32
sd a3, 8(fp)

srai a4, s1, 32
slli a5, a4, 32
srli a5, a5, 32
sd a5, 24(fp)

#Flag writes
```

(a) x86 assembly and IRancini (b) RISC–V assembly

Figure 5.8: Translation of a multiply

of the registers the extracts are special cases. The first one just extracts the lower 32 bits, so according to the convention of keeping all values sign extended we implement this as a sign extension. In general bit extracts first discard all unneeded high bits using a left shift and then align the bits on the right side of the register. For the second bit extract no high bits need to be discarded so we omit the left shift. The arithmetic version of the right shift (`srai`) automatically performs the necessary sign extension.

To adhere with the zero extending property of x86 before writing the register values back they are zero extended. Before the write back we zero extend the register values to adhere with the x86 semantics. This again uses the standard left shift, right shift combination.

This translation performs a lot of redundant or unnecessary work. In particular the sign extensions immediately followed by zero extensions can easily be optimized away. But the current translation process handles all nodes separately so it does not allow any multi-node optimizations.

5.2 Register allocation

Allocating registers to the instructions is one of the essential parts of a code generator. The theoretically infinite intermediate values, which are represented by the ports in our IR, need to be mapped to a finite number of registers when generating assembly. Typically, this process involves analysis of the live ranges of variables to determine when registers are needed and when they can be reclaimed and used for a different variable. The challenge is that often more variables are live than registers are available to assign, so some of the registers need to be temporarily spilled to memory. It is not a trivial task to choose which registers should be spilled to optimally use the available resources.

RISC–V provides a relatively high number of general purpose registers. Some of its 32 registers have special uses either within the RISC–V ABI or internally by the translator. In particular the registers `x0` to `x4` are defined by the ABI as hard-wired zero, return address, stack pointer, global pointer and thread pointer. This is also reflected in their defined mnemonics of `zero`, `ra`, `sp`, `gp` and `tp`. Therefore we do not use those five registers. In addition to that, the frame pointer register (`fp` or `x8`) is assigned as base pointer to the register file in memory. So that one is also not available. The decision to statically assign register for the flags uses up another four registers (`s8-s11`). Lastly, `t6` (`x31`) holds the memory base address. This leaves 21 registers freely usable.

The design of packets with explicit reads and writes means that there is no ports that are used across packet boundaries. So no values are live across these boundaries. We observed that the 21 registers available are always enough to accommodate all values

Reduced register set		Extended register set			
`s1`	`a0-5`	`t0-2`	`a6-a7`	`s2-7`	`t3-5`
`x9`	`x10-15`	`x5-7`	`x16-17`	`x18-23`	`x28-30`

Table 5.2: Registers in the order assigned by the allocator

in a single packet. These two facts allow the register allocator to operate in a very simplistic way.

A table contains all the register in the order displayed in Table 5.2. Additionally, an index indicates which register from the table to assign next.

Choosing this register order prefers using `x9-15`. As visible in the table those registers belong to the special reduced subset of registers. Instructions from the C extension we have seen in Section 4.2.1 only encode up to two registers sharing one of the sources with the destination. Additionally, some of the instructions further reduce needed encoding space by only allowing registers from the reduced subset. The only other register available in this subset is `fp`, which sees much usage as the register file base pointer. By prioritizing those registers the possibilities of the assembler to choose C-encodings is greatly increased. There is basically no additional cost involved in doing this optimization, but it increases performance of generated code by reducing the needed instruction fetch bandwidth.

Ports can have multiple targets, so additionally for all allocations the port and allocated register are stored in a map. Before translating a node, this map is checked for an existing allocation of the requested port. In case one exists we stop the translation and use it instead.

At the beginning of a packet all registers are declared as free by setting the index to zero and clearing the map.

6 Evaluation

In the following chapter we will evaluate our implementation based on the following two research questions:

- Can the end product run real world x86 programs correctly on RISC–V?
- Is the runtime performance of the translation acceptable for practical use cases?

6.1 Departure from the original design goals

The end product of this thesis turned out to be slightly different from the initial plan. This plan was to add a dynamic translation back end that converts the generic IR into executable RISC–V code at runtime. It would only be invoked, when a translation could not be realized before runtime because the instructions could not be identified. It was also a goal to support multithreaded applications by providing multiple parallel runtime threads and correctly translating memory accesses in a way that respects the strong memory ordering guarantees of x86.

In the end the LLVM back end was not finished in time for this thesis. Therefore it was not possible to translate any instructions ahead of time. It is possible to turn the translator into a mode that ignores the static translation and instead treats any instruction pointer value like it was not discovered before runtime. In this mode the dynamic back end for the target architecture handles the translation for all instructions that are dynamically executed. Effectively this means Arancini runs as a dynamic binary translator.

The intention when developing the RISC–V dynamic back end was to only run it for a very small fraction of the executed code. With this in mind there was not a particular focus on generating efficient machine code. Instead the goal was to structure the translation in a way that is easy to understand to reduce the possibilities to introduce mistakes. The second main goal was to not spend a lot of time in the back end. Any overheads caused by the translation process were expected to be more significant than the gains of better instruction sequences. In the full dynamic translation environment the back end currently operates in this choice is less optimal.

Supporting multithreaded applications is something that needs to be implemented almost exclusively in the runtime system. Currently the runtime only supports a very limited number of system calls which does not include the ones necessary for implementing threads. Adding this support is not entirely trivial so it was not made a top priority. Moreover, the front end does not currently reflect the strong guarantees of the x86 memory ordering in all cases. This means multithreaded applications cannot be translated correctly at the moment.

6.2 Translation correctness

Overall the approach of implementing separate front and back ends allowed a much simplified implementation. Due to the way the x86 ISA has historically grown, there exist a very big number of different instructions and instruction variants. Many of these special and complex semantics were solved in the front end and did not even make it through to the back end. But even with this there was still a big number of possible nodes that needed to be implemented in target assembly. The different possible operand sizes caused a big increase in node variants as well.

The process of implementing and testing showed that the vast majority of instructions included in a compiled x86 binary belongs to a small subset of the ISA. This meant that it was relatively straight forward to start running simple binaries and confirming the implementation is correct. At the same time in some cases relatively rare instructions were wrongly translated.

Errors resulting from incorrect translations can often be difficult to pinpoint, as they can surface in ways that obscure the source of the mistake. There are several ways to identify errors in the generated code.

Segmentation faults in the execution of memory accesses provide an easy way to spot flaws in the translation. These can result from accessing an address that was incorrectly calculated. Another reason for this can be that the memory accesses is not supposed to be performed and incorrect control flow results in its execution. Segmentation faults cause the program to abort execution at the memory access. So there is a good starting point to look for wrong translations. In most cases the mistake was very close to the segmentation fault, which made it easy to identify.

The second main way to verify correctness is to compare the outputs computed by test programs on a native execution to the ones generated when running the translator. Mismatches between the two are a clear sign for a translation error. Tracing the wrong value back to its origin is often significantly harder than in the previous case. The reason for this is that these values are generally modified a lot of times before they are eventually printed out. Depending on the algorithm, even a single incorrect update can result in a significant change to the output. Therefore all of these modification have be checked to spot the flawed one.

Once we figured out an instruction that has a wrong translation the process of correcting the mistake is not always straightforward either. The design of the translator with separate front and back ends with an IR as the interface introduces multiple places for the wrong translation to originate from. The first possibility is that we implemented one of the involved nodes with the specific parameters wrongly in the back end or missed one of the corner cases. Another option is that the front end generates IR that does not accurately represent the instruction and so the back end produces a correct implementation of flawed semantics. Since a different developer independently implemented the back end identifying this type of error is sometimes more difficult as well.

Overall we validated the correctness of the translator using the same programs that were also used for the performance tests (Section 6.3.1). None of the applications exhibited unexpected program crashes. All of them also print the results of the calculations they perform. We compared the output of the test programs running using our translator to the same programs running on native x86 hardware. On the final version of our back end there were no mismatches.

While this is no guarantee for a fully correct implementation, it provides convincing evidence for the correctness of the subset of the ISA used in these binaries. Corresponding to the computation the test programs perform, this covers a big part of the x86 instructions that are perform standard integer actions.

6.3 Performance tests

6.3.1 Test setup

Test system

The performance tests were performed on the HiFive Unmatched RISC–V Development Kit by SiFive [SiF21]. This board is powered by the SiFive Freedom U740 SoC, which contains an U74-MC CPU consisting of four U74 application cores and a single lower performance S7 monitor core. The four application cores support the RV64GC instruction set configuration. This allows running a standard Linux based operating system on them. To integrate seamlessly with other available systems NixOS [1] was the distribution of choice. The Linux Kernel version used was 6.0.19.

[1] `https://nixos.org/`

Test programs

The performance evaluation was performed using the sample programs provided by Phoenix [YRK09]. Phoenix is a shared memory implementation of the MapReduce execution model for multi-threading data processing tasks. They cover a variety of different processing tasks, including integer, string and float tasks. As mentioned in the previous section there is no support yet for translating applications that require multiple threads. So the programs that were tested are the sequential versions of the programs.

The sample apps were all compiled using GCC 11.2.0. The used optimization level used was `-O3`. For the baseline run the programs were cross compiled to RISC–V on an x86 host computer. The binaries to test the translator were natively compiled for x86-64.

There is no support for dynamically linked libraries in Arancini so the test programs were statically linked. Instead of the *GNU C Library*, which is the default C standard library for Linux, the musl C library [Mus23] was used. This library is designed from scratch to allow efficient statically linked executables. Simplicity is one of its main design points which makes it a great fit for testing this project. This results in a far smaller footprint reducing the amount of code needing to be translated. Another reason this makes translation easier is that the usage of special instructions, like SIMD instructions, is far reduced, which reduces the number of instructions needing support in the front end and the used IR node combinations in the back end.

Translator

The translator including all of its runtime libraries were cross compiled to RISC–V on an x86-64 host using GCC version 12.2.0. The static translation path was disabled, so the preparation of the binary before runtime just generated the execution loop. The resulting binary was linked with all runtime library components of Arancini statically.

Test data

Some of the programs use randomly generated input data. To ensure comparability we used the same random data for the native and translated runs. For `kmeans` we used 5000 points instead of the standard 100000. In the `matrix_multiply` program we used matrices of size 256. All other programs use pre-generated data. For these we used the smallest available test data set. Without reducing the data sizes down this much the test runs of the translated binaries would not have completed in reasonable time.

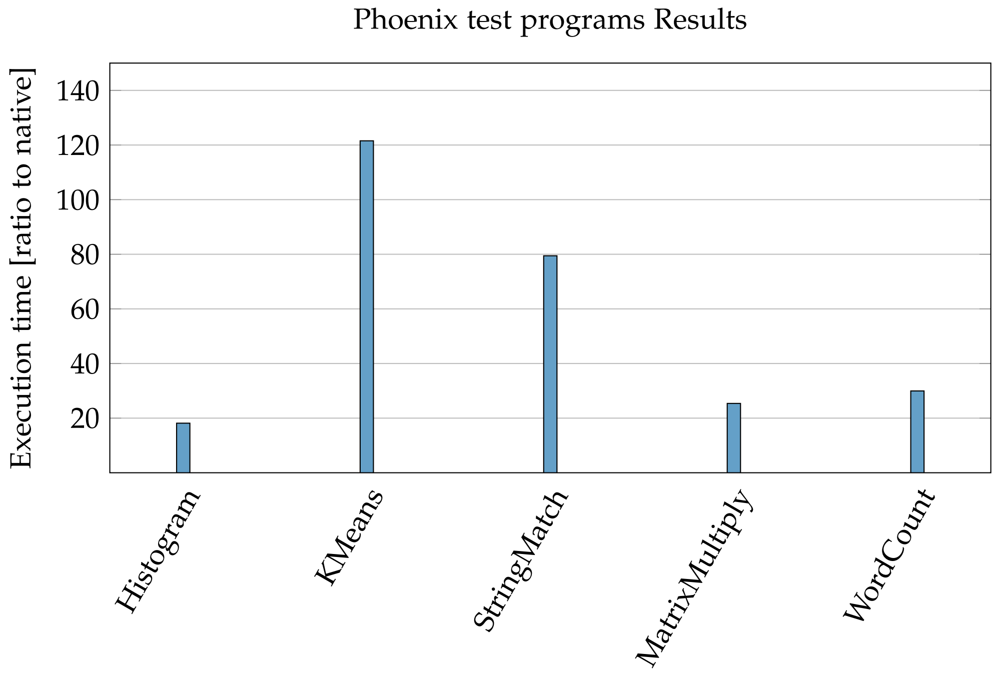


Figure 6.1: Results of the sample apps included in *Phoenix* (sequential versions) (normalized, lower is better).

6.3.2 Results

Figure 6.1 shows the normalized execution times of the executed sample apps included with Phoenix. All times were averaged over ten runs. From the original set of test programs we excluded `pca` because it is limited by the speed the result is printed to `stdout`, so it does not accurately represent translator speed. We also could not include the `linear_regression` test program because it is based on floating point math and the corresponding translations are unfinished, so it is not possible to run this test.

The results show the overhead of the execution of the test apps running using the RISC–V dynamic back end of Arancini in comparison to running the same program natively compiled for RISC–V. Of course this means it is not clear if all of the overhead observed can be attributed to the translation process.

The architectural differences between x86 and RISC–V, that we took a look at in Section 2.3.2, influence the relative execution times as well. There may be more efficient instruction sequences available for the specific task in either of the two ISAs. Due to its nature as a CISC architecture one would generally expect an advantage for x86 in this regard. Of course any instructions without equivalents on RISC–V have to be split up into multiple instructions in the process of translation. So the advantages of x86 are not relevant for the performance differences that can be observed. In the same way in some specific cases the RISC–V ISA may have an advantage, e.g. due to the higher register count.

Still most of the observed overhead should be caused by the binary translation. Some of the slowdown is caused by the translator infrastructure (decoding, translation, code cache lookups etc.). The other big contributor is the quality of the generated RISC–V code. The taken approach of translating every IR node in isolation from its predecessors and successors in the IR graph is very simple. This allows a very general support for any instruction but at the same time many instructions do not map efficiently in this way.

It can be seen that the overheads vary highly between the individual benchmarks. In general all of the tested applications run much slower than their native counterparts. The overhead ranges between about 18x in `Histogram` and 120x in `KMeans`. The reason for these huge differences lie in the type of computations these programs perform.

`Histogram` computes a histogram of the pixel values in each of the red, blue and green channels of bitmap picture file. It effectively counts the occurrences of every different byte value in a large memory mapped file. This involves almost exclusively memory and IO accesses, so the entire process is memory and IO bandwidth bound. This reduces the possible advantages of native execution and the overhead is comparatively low, albeit still large enough to make the translator it infeasible to use.

`KMeans` on the other hand groups randomly generated multi dimensional points into clusters such that every point is in the cluster with the closest midpoint. The used algorithm is iterative and reuses the same data multiple times in each iteration. At the same time the data set is comparatively small. Due to these two reasons most of the memory accesses can be served from cache. Therefore there is more importance on an efficient translation and infrastructure. As such the our unoptimized translator performs poorly.

For the other three test programs there are similar reasons that explain their performance numbers.

`StringMatch` splits the input file into lines and then compares each line, which contain exactly one word each, to multiple given strings. All of the comparisons happen in close succession to each other so again a lot of the memory accesses do hit the cache.

`MatrixMultiply` prints the full input and output matrices onto standard out. This is similar to the `pca` test case that I excluded. Different from that test case though there is still a reasonable amount of computation involved as matrix multiplication is generally a compute intense task. This program uses the naive matrix multiplication algorithm with three nested loops. So cache usage is also not optimal, which means the memory bandwidth might also contribute to less observed slowdown than other test cases.

`WordCount` counts the occurrences of each word in a text file. The algorithm of choice for this application keeps an alphabetically sorted array of the already discovered words along with their occurrence count. This means inserting a new word into the middle of the array copies all of the existing elements into the next cell, which again taxes memory bandwidth a lot.

Further analysis of the executed instructions shows that especially arithmetic operations are very slow. One of the main reasons for this is most likely that these operations set flags on x86 so their translations do as well, even though in a large majority of cases they are never read. This takes an additional four to six instructions depending on the specific instruction.

Additionally all of the register accesses are mapped to memory accesses to the register file. So any of the arithmetic instructions cause at least seven additional memory accesses: the reads of the two input registers and the write of the output register and the four flags. Due to the very frequent accesses to the register file, it can be expected that it resides fully in L1 cache. This means the effect on execution time is minimized. Still all of the arithmetic instructions, which in x86 includes instructions to compare, take an additional seven instructions for this.

A further reason for major slowdowns is that the partial register are not natively available on RISC–V in the same way as they are for x86. As shown in Section 5.1.4 these require to explicitly recreate the semantics implied by x86. None of the test programs use 64bit integers, so all instructions implementing the computations actually have overhead in this way.

7 Related work

7.1 QEMU

QEMU is the open source standard for machine emulation and virtualization. Similar to the way Arancini operates it also supports so called user mode emulation. This means programs are emulated directly on the host hardware and operating system. There is no separate emulated guest operating system or hardware. Like the name tells only binaries running fully in user mode are supported, so supporting system call emulation is enough to support different hardware. [Bel05]

On top of user space emulation QEMU also can do full system emulation. In this mode a complete guest operating system is emulated as well as all necessary hardware such as storage or network cards. The instruction set is emulated in the same way as the user mode emulator. This mode serves an entirely different use case to user space emulation. The user gets an environment that very closely resembles that of a system running this operating system on real hardware. So it it possible to run multiple translated programs in parallel and their interactions also run as they would on real hardware. [Bel05]

QEMU's translation approach is very similar to ours. First, a front end translates the source instructions into an intermediary language, which is called TinyCodeGen (TCG) ops for QEMU. Then the generated TCG ops run through a couple of basic optimization passes, e.g. dead code elimination etc.. The last step is to generate host instructions from the TCG ops in a back end. [Bel05]

This approach of decoupling the different stages allows QEMU to support a wide variety of architectures. In this way different people can develop front ends and back ends independently from each other. This really lets the open source nature of QEMU shine. It allows different parties with knowledge on different instruction sets to use their expertise.

Due to the already wide applicability a new architecture gains a lot of practicability being supported by QEMU. In this way vendors have an interest to have QEMU available for their architecture in the same way they need a compiler toolchain. SiFive, one of the main manufacturers of RISC–V cores, for example contributed large parts of the back end and front end for RISC–V.

Any translations that QEMU provides are done at runtime. There is no static translation path that runs ahead of time.

7.2 Rosetta 2

Apple recently stopped using Intel processors for their desktop and laptop computers and instead designed their own chips based on the ARM ISA. The only reason they were able to convince customers to buy products with these new chips was by making it possible to run nearly all programs written for the x86 ISA on them. To make this possible they introduced the Rosetta 2 binary translator. [App23]

Rosetta 2 translates most of the application ahead-of-time (AOT) when the program is run for the first time. Because x86 binary code is not completely statically translatable and they want to support runtime generated code as well, just-in-time (JIT) translation of x86 code into the ARM ISA is supported too. [Nak21]

Being in control of the design of the whole processor, they added extra hardware to resolve some of the major differences of x86 in comparison to ARM. Specifically the different memory consistency models and differing flag registers are supported via custom additions on top of the standard ARM design. [Joh22]

Rosetta 2 provides a very impressive framework for transitioning to the new Apple hardware. At the same time it is a proprietary software, so it is very restricted. It is only available as a package with Apple's hardware. This very specific field of application also allowed them to design it specifically for their CPUs, so it only supports the specific architecture of these.

8 Summary and Concluison

This thesis showed the viability of translating x86 binaries into RISC–V dynamically at runtime. We implemented the translation as part of the Arancini project. This approach meant the translation process is split into a front and back end. The front end generates an intermediate representation. The IR is passed into the back end to generate executable code. Our contribution is a back end for runtime lowering of the IR into RISC–V machine code.

First we established the required background information for understanding binary translation and the differences between x86 and RISC–V that need to be bridged. We presented the core design of the Arancini binary translator that translates all code it can discover in a static phase preparation phase before runtime. Only for not discovered code it starts a translation at runtime using the dynamic back end. We explained the design of the intermediate representation. Next, we presented the design principles of the RISC–V backend. Using some instruction examples we showed the instruction sequences we generate and explained the reasoning behind choosing these instructions.

Different from the initial intended application we ran the translator in a mode that ignores the static translation path because it is still incomplete. In the evaluation we showed that our approach correctly translates integer based x86 binaries on real RISC–V hardware. The performance of the current approach is magnitudes slower than native execution, so without further improvements it does not provide a feasible solution for real world tasks. Therefore the translator can be seen as a proof of concept rather than a production ready solution.

Every part of the translation is available as part of the "arancini-exploration" project on Github [Ara23]. The main part of our contribution is the RISC–V dynamic back end. The source files of this are organized in the `src/output/dynamic/riscv64/` and `inc/arancini/output/dynamic/riscv64/` directories. As part of working on the project we also touched files in other locations.

9 Future work

The performance results in Section 6.3 show that it is not feasible to use the current translator to run x86 programs on RISC–V hardware outside of test runs.

There are a couple very simple optimizations which when included would improve performance significantly.

A first very basic optimization is to map each x86 register to a RISC–V register for the length of a basic block. In many cases registers are used in multiple times instructions in a a short succession. In those cases this approach will reduce these registers to a single memory access per basic block. As memory accesses are generally one of the slowest parts of execution in a CPU this greatly speeds up execution speed.

A second basic optimization targets the slow switching time in between different basic blocks. Most basic blocks end with direct jump instructions. In the case of unconditional jumps or call instructions the following block is already decided before executing them. If the instruction is a direct branch there are only two possible following blocks.

Through a technique called *chaining* the switch to the runtime system to select the next block can be omitted in many situations. Instead a direct jump to the translation of the next block is inserted if it exists already. A further refinement of this technique is to retrospectively chain blocks once the jump target was translated. In this way subsequent execution of the block do not need to lookup the address again. The translation cache lookups are one of the slowest parts in the runtime, so reducing their number is a good way to increase performance.

There are many other potential ways to increase the performance. Especially targeting the quality of the generated code. Considering the end goal of this project to translate most parts ahead of time it remains to be seen if the amount of code that needs to be dynamically translated warrants the effort to implement any of these optimizations.

Another way to expand on the results of this implementation could be to add support for multithreaded programs. On top of requiring support for multiple execution and translation threads in the runtime this would also require to correctly map all memory accesses of the strong consistency requirements in the x86 architecture to the weaker ones in RISC–V. In light of the big and ever growing distribution of multithreaded programs, it appears to be essential for real world use to have this support.

List of Figures

List of Tables

Bibliography

[Adv23] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual.* `https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5`, visited 2023-03-10. Jan. 2023.

[And19] Andrew Waterman and Krste Asanović, ed. *The RISC-V Instruction Set Manual, Volume I: User Level ISA, Document Version 20191213.* RISC-V Foundation. Dec. 2019.

[App23] Apple Inc. *About the Rosetta Translation Environment.* `https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment`, visited 2023-03-31. 2023.

[Ara23] Binary Translation. *Arancini.* `https://github.com/binary-translation/arancini-exploration`. 2023.

[Bel05] F. Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track.* Vol. 41. 2005, p. 46.

[Int23] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual.* `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, visited 2023-03-10. Feb. 2023.

[Joh22] D. Johnson. *Why is Rosetta 2 fast?* `https://dougallj.wordpress.com/2022/11/09/why-is-rosetta-2-fast/` (last visited 2023-04-09). Nov. 2022.

[LA04] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04).* Palo Alto, California, Mar. 2004.

[Lat11] C. Lattner. "LLVM." In: *The Architecture of Open Source Applications.* Ed. by A. Brown and G. Wilson. Vol. 1. May 2011, pp. 155–170.

[Mus23] *About Musl.* `https://musl.libc.org/about.html` (last visited 2023-04-04). 2023.

[Nak21] K. M. Nakagawa. *Project Champollion: Reverse engineering Rosetta 2.* Version 0.1.0. 2021. URL: `https://github.com/FFRI/ProjectChampollion`.

[Pro02] M. Probst. "Dynamic binary translation." In: *UKUUG Linux Developer's Conference*. Vol. 2002. 2002.

[SiF21] SiFive. *HiFive Unmatched RISC-V Development Kit*. `https://sifive.cdn.prismic.io/sifive/d0556df9-55c6-47a8-b0f2-4b1521546543_hifive-unmatched-datasheet.pdf`. 2021.

[SN05] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[UC00] D. Ung and C. Cifuentes. "Machine-Adaptable Dynamic Binary Translation." In: *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*. DYNAMO '00. Association for Computing Machinery, 2000, pp. 41–51. DOI: `10.1145/351397.351414`.

[Wat+11] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Tech. rep. UCB/EECS-2011-62. EECS Department, University of California, Berkeley, May 2011. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html`.

[YRK09] R. M. Yoo, A. Romano, and C. Kozyrakis. "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system." In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 198–207. DOI: `10.1109/IISWC.2009.5306783`.