UNIVERSIDADE ESTADUAL DE CAMPINAS
Instituto de Computação

RAFAEL AULER

OPENISA, A HYBRID ISA

OPENISA, UM CONJUNTO DE INSTRUÇÕES HÍBRIDO

CAMPINAS
2016

Rafael Auler

OpenISA, a hybrid ISA

OpenISA, um conjunto de instruções híbrido

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Supervisor/Orientador: Prof. Dr. Edson Borin

Este exemplar corresponde à versão final da Tese defendida por Rafael Auler e orientada pelo Prof. Dr. Edson Borin.

CAMPINAS
2016

Agência(s) de fomento e nº(s) de processo(s): FAPESP, 2011/09630-1

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

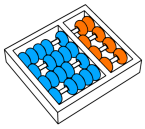
Auler, Rafael, 1986-
Au51o OpenISA, a hybrid ISA / Rafael Auler. – Campinas, SP : [s.n.], 2016.

Orientador: Edson Borin.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Máquinas virtuais. 2. Interpretadores (Programas de computador). 3. Computadores com conjunto de instruções reduzido. 4. Microprocessadores - Projetos e construção. 5. Compiladores (Programas de computador). I. Borin, Edson,1979-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: OpenISA, um conjunto de instruções híbrido
Palavras-chave em inglês:
Virtual machines
Interpreters (Computer programs)
Reduced instruction set computers
Microprocessors - Design and construction
Compilers (Computer programs)
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Philippe Olivier Alexandre Navaux
Fernando Magno Quintão Pereira
Rodolfo Jardim de Azevedo
Sandro Rigo
Data de defesa: 23-09-2016
Programa de Pós-Graduação: Ciência da Computação



UNIVERSIDADE ESTADUAL DE CAMPINAS
Instituto de Computação

Rafael Auler

OpenISA, a hybrid ISA

OpenISA, um conjunto de instruções híbrido

Banca Examinadora:

- Prof. Dr. Edson Borin
  IC - UNICAMP
- Prof. Dr. Philippe Olivier Alexandre Navaux
  INF - UFRGS
- Prof. Dr. Fernando Magno Quintão Pereira
  DCC - UFMG
- Prof. Dr. Rodolfo Jardim de Azevedo
  IC - UNICAMP
- Prof. Dr. Sandro Rigo
  IC - UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 23 de setembro de 2016

Acknowledgements

I would like to thank my invaluable colleagues during the development of this PhD thesis: Bruno Cardoso, with whom I shared the happiness of publishing the first book in English about LLVM; Raoni Fassina, for the many interesting discussions and podcasts recommendations; Tiago Falcao, for being a friend since my first year at UNICAMP when I was an undergrad; Maxiwell, for the insightful discussions about spirituality, chess, entrepeneurship and music; Joao Moreira, for everything we shared, including a house in the summer of 2013 in Seattle; Leonardo Piga, for showing me that hard work, endurance and persistence pays off; Thiago Abdnur, for the excellent reading recommendations; Daniel Nicacio, for always being in touch with us despite living far away in the Scandinavian Peninsula; Leonardo Ecco, for your companionship during the early years; Alexandro Baldassin, for presenting me to grad school; and Gabriel Ferreira, for showing me the importance of curiosity, diligence and the taste for writing.

I thank Microsoft Research for the fellowship and internship in 2013 and the people there that helped me develop a good project, including Peli de Halleux, Michal Moskal and Nikolai Tillmann. I thank Sony Computing Entertainment America for the internship in 2014, in special Alex Rosenberg, Rafael Espindola and Sean Silva. I thank Facebook for the internship in late 2015, in special Guilherme Ottoni, Maxim Panchenko and Surupa Biswas.

Above all, I thank my partner Raysa for the love and patience that was essential to make my life during the PhD such a great time I will remember dearly.

Finally, I thank my advisor, Edson, for all the support, guidance, meetings and conversations that lead to this thesis.

R. A.

Resumo

OpenISA é concebido como a interface de processadores que pretendem ser altamente flexíveis. Isto é conseguido por meio de três estratégias: em primeiro lugar, o ISA é empiricamente escolhido para ser facilmente traduzido para outros, possibilitando flexibilidade do *software* no caso de um processador OpenISA físico não estar disponível. Neste caso, não há nenhuma necessidade de aplicar um processador virtual OpenISA em *software*. O ISA está preparado para ser estaticamente traduzido para outros ISAs. Segundo, o ISA não é um ISA concreto nem um ISA virtual, mas um híbrido com a capacidade de admitir modificações nos opcodes sem afetar a compatibilidade retroativa. Este mecanismo permite que as futuras versões do ISA possam sofrer modificações em vez de extensões simples das versões anteriores, um problema comum com ISA concretos, como o x86. Em terceiro lugar, a utilização de uma licença permissiva permite o ISA ser usado livremente por qualquer parte interessada no projeto. Nesta tese de doutorado, concentramo-nos nas instruções de nível de usuário do OpenISA. A tese discute (1) alternativas para ISAs, alternativas para distribuição de programas e o impacto de cada opção, (2) características importantes de OpenISA para atingir seus objetivos e (3) fornece uma completa avaliação do ISA escolhido com respeito a emulação de desempenho em duas CPUs populares, uma projetada pela Intel e outra pela ARM. Concluímos que a versão do OpenISA apresentada aqui pode preservar desempenho próximo do nativo quando traduzida para outros hospedeiros, funcionando como um modelo promissor para ISAs flexíveis da próxima geração que podem ser facilmente estendidos preservando a compatibilidade. Ainda, também mostramos como isso pode ser usado como um formato de distribuição de programas no nível de usuário.

Abstract

OpenISA is designed as the interface of processors that aim to be highly flexible. This is achieved by means of three strategies: first, the ISA is empirically chosen to be easily translated to others, providing software flexibility in case a physical OpenISA processor is not available. Second, the ISA is not a concrete ISA nor a virtual ISA, but a hybrid one with the capability of admitting modifications to opcodes without impacting backwards compatibility. This mechanism allows future versions of the ISA to have real changes instead of simple extensions of previous versions, a common problem with concrete ISAs such as the x86. Third, the use of a permissive license allows the ISA to be freely used by any party interested in the project. In this PhD. thesis, we focus on the user-level instructions of OpenISA. The thesis discusses (1) ISA alternatives, program distribution alternatives and the impact of each choice, (2) important features of OpenISA to achieve its goals and (3) provides a thorough evaluation of the chosen ISA with respect to emulation performance on two popular host CPUs, one from Intel and another from ARM. We conclude that the version of OpenISA presented here can preserve close-to-native performance when translated to other hosts, working as a promising model for next-generation, flexible ISAs that can be easily extended while preserving backwards compatibility. Furthermore, we show how this can also be a program distribution format at user-level.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The ISA, Instruction Set Architecture, is the interface that allows applications and the processor to be developed independently. It is a contract. As long as the application code and the processor microarchitecture comply with the ISA, the processor should be able to properly execute applications. There are currently several different ISAs that software can use to leverage a circuit to do useful work, ranging from complex general purpose instruction sets to simpler ones, dedicated to graphics processing.

This chapter discusses challenges brought by this classic ISA definition, then proceeds to present the solution virtual machines provide and concludes with the hybrid ISA approach championed by this thesis. This chapter also shows how this thesis is organized and summarizes its contributions.

1.1 Challenges in the design of classic ISAs

Coping with the strict requirements of the ISA interface raises at least three big technical challenges:

1. *Hardware challenge*: Each ISA decision affects all future generations. In effect, changing features of an ISA that was already shipped to the market deems the previous generation incompatible. In practice, to keep backwards compatiblity, the hardware typically is limited to be only extended. This implies ISAs will always become larger, more cluttered and more inefficient as each new generation is introduced [92].

2. *Software challenge*: The software, once compiled to a specific ISA, demands a non-trivial effort to be ported to run on other ISAs.

3. *Market challenge*: Past ISAs accumulate a legacy of available software. Creating a brand new ISA is discouraged because of this significant amount of effort to port software. Since current ISAs are mostly proprietary, new processor manufacturers are not encouraged to exist in the market if they need to license the rights to reproduce the ISA from another company. This strengthens monopoly in the computer processors market.

These challenges create a vicious cycle. On one hand, a processor manufacturer is limited to choose from a handful of established ISAs owing to the size of their software base. On the other hand, as an ISA becomes old and mature, it accumulates a larger software base, putting a heavier weight on an ISA from the past rather than on innovative new technologies.

Hence, old families of ISAs grow stronger while true innovation is inhibited. Consider the Intel x86 ISAs, the dominant ISA family for desktops and servers: not only does it still have hardware support for BCD arithmetic, reminiscent of the days when x86 was used to implement a microcontroller, but also it encodes these old instructions with a single byte, the most efficient encoding available for x86 instructions. New x86 AVX extensions, on the other hand, are encoded with at least 5 bytes [92]. This is a consequence of relying on very old ISA designs.

Up until this point, all ISAs suffered with aging, but only a few accrued enough years in their lifetime to make evident the hampering effects of supporting old instructions in recent processors. The best example of such effect is seen in Intel x86 because it is the ISA with the longest lifespan that is still popular today. The variable-length encoding format of x86 provides it with the flexibility to support an almost unlimited number of extensions, but with each new extension, comes a price to pay – the longer encoding. The ISA invariably becomes inefficient and it is also impossible to fix past design mistakes without breaking compatibility.

An ISA does not only express a hardware design point [39]. In parallel with the hardware, software compiled to a specific ISA also suffers with its inflexibility. An ISA establishes a program distribution format with profound impact on the software deployment problem. Once a program is distributed in binary form, it is restricted to run on platforms that have a processor capable of understanding it. Maintaining software for multiple platforms is not trivial and demands active effort from developers, making the ISA an important contract with respect to software distribution. The semantic gap between different machines is expensive to bridge with existing emulation technology. Therefore, running a software compiled for x86 on ARM, for example, involves accepting large overheads as ARM tries to mimic the behavior of a virtual implementation of the Intel processor.

1.2 Virtual machines to ease software deployment

In the last two decades, the computer science industry went through an important development. Industry pursued a homogeneous platform for programmers despite the fact that the hardware platforms were so diverse: with each computing system (desktops, servers, smartphones, among others), a different processor and operating system. Since it is counter-productive for developers to support multiple platforms, Java-based virtual machines quickly gained traction because the virtual machine technology allowed developers an effective strategy of program deployment to a wide spectrum of different devices.

Java, however, was a partial success in the attempt to simplify software development. The intermediate language is high-level, rendering it unsuitable to solve the broader soft-

ware deployment problem. It only supports a single program paradigm with object-oriented programming, it has performance implications because it employs expensive memory usage shepherding, depends on a compiler and its language was not built to be efficiently implemented as a competitive processor.

Therefore, even in light of the easiness of deployment of the Java era, programmers still felt compelled to perform a significant fraction of software development in native mode, which is the classic software development flow that is invariably locked to specific hardware platforms. Since this was common-practice, when x86 processors entered the smartphone market dominated by ARM processors, a binary translator was employed to translate such native code, succeeding with arguable efficiency.

In the dawn of the Internet-of-Things (IoT) era, we are once again faced with the same deployment problem, but this time with a wider spectrum of different hardware platforms and some of them with such a scant amount of resources that using virtual machine technology is barely acceptable. Programmers want their applications to just run, while chip manufacturers each offer a completely different and incompatible platform.

The deployment problem highlights the issues in classic, concrete ISAs, but a complete solution must not limit itself to the processor, but encompass the entire software stack.

The Transmeta team [51] created the Crusoe series of processors and the Code Morphing Software (CMS), their DBT system that translated the x86 ISA to their own VLIW host ISA. Their system proved to be possible to translate old x86 code to a completely new system that explored different power tradeoffs and offered a new design option to x86 processors. However, their experience showed that it is tricky to support complex guest ISAs such as the x86, requiring many custom modifications in the host ISA to efficiently support x86 quirks [20,37,118]. In this thesis, we do the opposite and provide an analysis of guest ISA characteristics that offer easy and efficient emulation on popular host ISAs.

1.3 Introducing the hybrid ISA solution

This thesis investigates the guest ISA because it supports the idea that the solution to the classical ISA issues comes with the design of new, highly flexible ISAs. These ISAs should present three important properties that are promising to handle the technical challenges brought by the old, fixed ISA interface. The first and key property is the easiness of translation, giving freedom to programs that use the ISA instructions to be seamlessly emulated, via efficient binary translation, on other established architectures. The second, an innovative extensibility mechanism that allows the ISA itself to recycle its own instructions, being always as fresh as a newly designed ISA, free from the constraints of binary compatibility. An OpenISA processor may have part of the instructions implemented in hardware and part of them implemented in software, which gives its hybrid characteristic that also allows it to recycle part of the instruction set. The third, an open license, allowing anyone to manufacture processors that run the ISA code.

While the benefits on the hardware side are clear, that is, to enable it to freely evolve and recycle itself, on the software side, it is also a promising solution. Instead of relying on the complex and more abstract Java ISA, virtual machines can work with low-level

(machine) code and can have an uniform binary (and bug) compatibility across all targets [124], something Java-based technologies cannot offer by design.

Distributing programs encoded in machine code (at the ISA level) gives more freedom to the software, which is not tied to a specific programming paradigm. If the ISA is easily emulated by itself, it eliminates the necessity of elaborate, higher-level virtual machines that depend on distributing programs at the source-code level or similar. An ISA-level virtual machine also works for a wide spectrum of hosts: for the IoT use case, for example, the attractiveness of such virtual machines for resource-constrained devices comes from the higher simplicity: one does not need the Java Runtime Environment to support the Java ecosystem, but only a simple interpreter because the guest program is already compiled and optimized. The VM now only has to focus on how to bridge the differences between host and guest hardware platforms. Therefore, it is important that the guest ISA be as easy to be emulated as possible.

Throughout this thesis, we develop and discuss OpenISA [25], a flexible, hybrid ISA solution as both a guest ISA for virtual machines, highlighting its efficient emulation, as well as an ISA to shape the implementation of modern processor microarchitectures.

For the specific use case of IoT platforms, to ensure a lean platform in restricted platforms, an OpenISA virtual machine should focus on interpretation of OpenISA code. If a constrained platform needs performance but lacks the resources to run a multi-gear, optimizing virtual machine, this thesis introduces the use of CATs, cloud-assisted translations, a technique that relies on the cloud to perform expensive optimizations and deliver a completely translated binary back to the host.

Last, this thesis investigates a program-distribution format based on OpenISA. For userland programs that do not employ self-modifying code, we show a tool that is able to statically translate code in this format to two popular host processors, one from Intel and another from ARM.

1.4 Contributions and thesis organization

Overall, the contributions of this thesis are the following:

- Demonstrates the feasibility of building virtual machines with intermediate languages as low level as a RISC ISA and argue in favor of its advantages over traditional, higher-level VM languages such as Java, highlighting that RISCs allow for an efficient hardware implementation and are not language-dependent;
- Analyzes the binary translation problem from the perspective of the guest ISA instead of the host ISA;
- Presents a set of ISA properties that lead to efficient binary translation to both ARM and x86, illustrating two distinct categories of established ISAs;
- Presents an encoding and recycling mechanism that allows the ISA to evolve without breaking backwards compatibility;
- Investigates how guest registers mappings impact the ISA translation;

- Evaluates the full potential of binary translation using modern open source compiler technology, LLVM 3.6.
- Presents a mechanism to allow the virtual platform to execute either OpenISA or native code generated by an OpenISA to host translator executing in the cloud (assisted translation).
- Investigates a low-level program distribution format based on OpenISA that enables a static translation engine to perform offline translation of OpenISA programs to other hosts while preserving most of native performance.

This thesis is organized as follows. Chapter 2 analyzes what the current literature provides us on the topics of binary portability and binary translation. Afterwards, Chapter 3 introduces OpenISA, Chapter 4 explains its encoding and recycling mechanism, Chapter 5 discusses the design of the experimental framework, Chapter 6 presents the choices of ISA properties that foster better quality in binary translation, Chapter 7 shows experimental data to back such decisions, Chapter 8 presents other exploratory work developed in the context of this thesis and Chapter 9 provides the conclusion.

Chapter 2

Related work

The field of study concerned with translating software distributed in binary form from one processor to run on another is *binary translation* [123]. A *virtual machine*, on the other hand, is a broader concept that may or may not rely on binary translation to *emulate* the behavior of an existing platform [124]. This thesis investigates relevant work published in literature covering all solutions that propose a compelling program distribution format, a compelling new ISA, solutions that improves binary translation technology or that improves virtual machines or virtual instruction sets (V-ISAs) in general.

This chapter analyzes how each of these solutions differs from the approach used by OpenISA. It then builds the motivation and presents a survey of the literature on fast ISA translation to guide a plan on building an emulation-friendly ISA.

2.1 Basic concepts

Before delving into the details of each level, it is useful to define a few concepts.

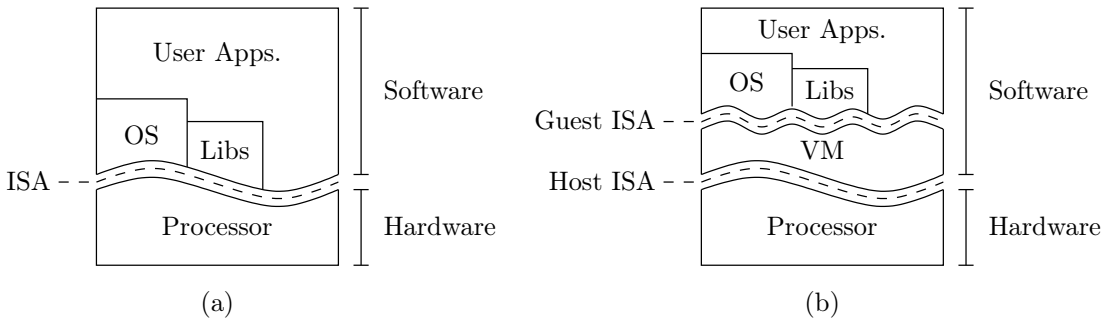


Figure 2.1: (a) Typical computer systems hardware/software execution stack. (b) Computer system software stack executed on top of a virtual machine (VM).

The oldest and most important concept is that of a virtual machine. Figure 2.1 compares a regular hardware/software stack (a) versus the one using a virtual machine (b). A virtual machine is a central topic to this work, as it can be defined as the framework to run a program that was not necessarily built to run on the same hardware platform that the virtual machine runs [124]. Whenever running a program intended for a platform A on a hardware platform B by using any virtual machine, we say that A is the *guest*

platform and B is *host platform*, while its respective ISAs are the Guest ISA and Host ISA.

When compiling, or translating a program from a source code to use the instructions of a specific hardware platform, we call this hardware platform the *target*.

2.2 Well-known virtual machines

Currently there are three successful systems that were designed with the specific objective of solving portability issues through a virtual machine:

- The Java Virtual Machine (JVM) [8, 102] by Oracle is quite well-known, but has important constraints that are not a problem in an OpenISA-based virtual machine. Java does not admit structs, stack objects nor pass by reference of stack values and programs must obey a strict type hierarchy. As in the JavaScript analysis provided in this thesis, discussed in Section 8.2, the JVM is interesting to many applications, but it cannot express arbitrary programs as a solution to the portability problem without hampering performance;

- The Common Language Runtime (CLR) [84] by Microsoft lacks adequate support for non-Microsoft platforms, albeit it benefits with a higher flexibility in the generated code. Moreover, we can expect a shift of strategy in Microsoft in the near future towards Open-Source Software (OSS) technology, although it is unclear whether their .NET VM technology will receive much attention. On the other hand, the code still depends on a heavy runtime to support it and on using the base class library (BCL). An ISA-level approach does not restrict the code to use any specific library and is transparent to the programmer.

- The Low Level Virtual Machine (LLVM) [45], an OSS project managed by Apple, Google, SCEA, among others, first appeared in literature in 2003 as a solution to the portability problem. The paper from MICRO [17] sketched an IR with a much lower level than that seen in CLR and JVM and described an elaborated system that not only would support compilation to multiple targets, but also continuous optimization based on data from actual runs (feedback-based).

The purpose of going a level lower is to provide thorough support for portability of all software, not just those that obey a specific language, including system software. In the case of HLL frameworks, such as JVM and CLR, not only do they can never support system-level code, but programs also avoid them when they need performance, since they incur either time or memory overheads, sometimes quite unpredictably due to the JIT technology used [117].

The next subsection presents the case for JVM and CLR, or high-level language (HLL) VMs in general, as well as the case for LLVM. LLVM works on a different level, but is still a bytecode-based VM, which is different than an ISA-based VM. The bytecode is not intended to be read by hardware and it does not impose a fixed memory layout on guest programs.

2.2.1 Low-level ISA vs bytecode guest architecture

HLL virtual machines typically execute an intermediate language representation that reflects important features of a specific class of languages. On one hand, this approach includes more semantic information from the source program, allowing the system to perform more aggressive runtime optimizations to improve the system performance. On the other hand, this approach also makes the virtual machine more dependent on source languages.

Different from HLL virtual machines, ISA virtual machines are capable of running low-level binary programs that were compiled for a given instruction set architecture, or ISA. This solution is typically employed to grant intrinsic compatibility with real processor implementations, allowing full binary compatibility when running the binary on systems with different ISAs.

Since processor ISAs are language agnostic, using an ISA VM instead of an HLL VM allow us to leverage any language that has a compiler compatible with the ISA to produce portable binary code. GCC [6], for instance, has front ends for C, C++, Objective C, Fortran, Java, Ada, and Go. Therefore, it is possible to cover all these languages by emulating a single GCC target, which can be x86, ARM, MIPS, among others.

When comparing an HLL VM and an ISA VM with respect to bug reproducibility, the latter can be bug-compatible because the guest program, when compiled to a concrete ISA, assumes a fixed memory layout and uses instructions with a well-defined behavior. Since the VM does not change the guest memory layout, it cannot accidentally hide memory bugs or expose others, increasing the likelihood bugs will remain the same across different platforms. If the ISA is designed to be easily emulated, we can build VMs with two striking advantages: superior compatibility and source-language independence.

In the web technology front, there is a similar battle to find a good intermediate language that can be reliably used as the language of the web – a role currently filled by JavaScript. Mozilla proposed asm.js [98] as a subset of JavaScript that makes a good target for compilers, while Google proposed PNaCl [65] and its API, a new standard that would allow web applications to reach near-native speeds. More recently, a consortium of companies including Mozilla, Google, Microsoft and Apple has been working on WebAssembly [36], a binary format for the web. Their goals are similar to what an OpenISA program format would seek regarding the quest to find a good binary format for a wide range of hardware platforms. They have an explicit goal of compiling arbitrary languages to WebAssembly, including C and C++ and the whole standard library to be executed in a browser. Therefore, these projects may share common design decisions with this work.

The most compelling advantage of this work in comparison with WebAssembly, Java and other intermediate representations, however, is that it focus on the computer architecture facet of this problem and claims it is important to, instead, make the ISA easier to emulate while also defining protocols to allow the hardware to evolve without breaking backwards compatibility. An ISA that is easily emulated eliminates the necessity to distribute programs at source-code level, intermediate-language level or bytecode-level in the first place, since it is possible to run its code on other platforms via emulation.

There are several projects implementing real ISA virtual machines [2–5]. Most of them

are native system machines that emulate other computer hardware, including the entire instruction set. However, these projects do not propose to change the ISA with emulation performance in mind.

2.2.2 Comparison with recently developed ISAs

Other similar projects arose with the goal of redesigning ISAs, but not necessarily with ease of emulation as a goal. The dominance of proprietary and poorly designed ISAs spurred a few recent projects to change this prospect:

- The most notorious of all new ISA projects is RISC-V [24], championed by Asanovic and Patterson at Berkeley, which started roughly in the same years of OpenISA, with similar ideas. However, while RISC-V do share some aspects with this PhD thesis, more specifically, the case for open instruction sets, they do not focus their studies on emulation efficiency of the ISA nor do they propose any recycling mechanism to cope with ISA evolution. Therefore, OpenISA has not only a different design to achieve a shared goal with RISC-V, but also a different goal that RISC-V does not tackle, being a fundamentally different project.

- Another attempt at the redesign of ISAs, more recent than RISC-V, comes from Agner Fog with his essay "Proposal for an ideal extensible instruction set" [60]. While his work is non-academic, it has a lot of credit by being built with the comments and considerations of several experts that are also users of his well-known blog. Fog has been himself an x86 expert for many years who hosted a thorough extra-official guide for the Intel architecture at his website. He later discussed and decided to sketch the specifications for an ideal instruction set, according to his views, with respect to extensibility. He also does not discuss emulation performance. With regard to ISA extensibility, the main difference of this thesis is that it relies on a recycling mechanism to enable seamless evolution of the ISA. We do not endorse the view that it is possible to design the perfect ISA from scratch, but rather that design errors are unavoidable. Thus, the OpenISA approach relies on instruction recycling instead of an extensible design, a theme in which both works, RISC-V and Fog's, revolves around.

- Heterogeneous System Architecture (HSA) [116] is a project backed by a consortium of companies working together to build a single virtual ISA that will allow a programmer to use only one interface to parallel programming while the HSA compilation framework converts this program to a CPU or GPU underlying processor. In this case, HSA defines itself as a virtual ISA, which is not designed to be implemented by a real processor, while OpenISA defines itself as a hybrid ISA, which has all the characteristics of real ISAs, but is enabled with a recycling mechanism to support its evolution, sharing with virtual ISAs the idea of not implementing in hardware (selected) instructions.

2.3 The ISA level

OpenISA approach to the portability problem lies at the ISA level, the hardware interface with software. This section starts with a discussion of the original LLVM system [17] that is, to the best of our knowledge, the most similar work to the ISA level approach of this thesis. However, unlike OpenISA, LLVM IR cannot be implemented in hardware, which is a key distinction between this proposal and LLVM's.

Since the early days of the LLVM project until its most recent incarnations in the Clang compiler replacing GCC in the Software Development Kit (SDK) shipped by Apple, there has been much confusion about the goal of the project because it gradually changed from a purely research and speculative project to an industry-embraced project used in production. In this process, LLVM lost some of its original intents.

Chris Lattner, the LLVM original author, along with other project contributors such as Dan Gohman, Talin Viridia and Kenneth Uildriks, help us to understand and demystify the idea that LLVM is a virtual machine framework. Despite its acronym, it is not. Their discussion in the LLVM developer's mailing list is entitled "LLVM IR is a compiler IR" – and not a VM IR, as the name implies [91]. Their argument observed that the LLVM infrastructure has important limitations that precludes it from being suitable for hosting a complete virtual platform that seamlessly translates to any machine. There are differences in the LLVM IR design from both real virtual platforms, e.g. JVM, and real hardware platforms, e.g. ARM.

It is consensus that the acronym LLVM remains for historical reasons: the original LLVM paper presented it as a virtual machine that was able to run the virtual ISA LLVA (low-level virtual architecture). Now, the LLVM IR has the following characteristics that do not conform with portability, extracted from the original Dan Gohman's letter:

1. Target-specific features, like the x86_fp80 type;
2. Target-specific ABI code;
3. Implicit target-specific features (linkage types);
4. Target-specific limitations for seemingly portable LLVM features (each backend interprets information in its own way);
5. No undefined behavior defined - if you break an LLVM rule, the behavior will be unpredictable, unlike high-level VM that has precise exceptions defined or hardware platforms that has detailed behaviors specification;
6. The IR is intentionally vague and not formal;
7. Floating-point arithmetic is not always consistent.

The point that Gohman makes is that the LLVM IR contains many target-dependent annotations, besides being conceptually designed to be target-independent. Therefore, it is not possible to translate a generic LLVM IR fragment to an arbitrary target, but only for the target that the frontend was aware of when it generated the IR fragment.

In fact, currently, even the frontend Clang has target-specific optimizations, such as the early expansion of the `va_arg` C function that varies according to each target ABI.

Other points raised by other developers are as follows:

1. The LLVM bitcode, the representation in disk, is big when compared to the bytecode of other VMs (CLR and JVM);

2. The LLVM IR is not stable across versions;

These two points alone are sufficient to question whether the LLVM IR is an adequate representation for programs that demand portability. In fact, Gohman argues that it is not. LLVM IR has instead evolved to be strictly a compiler IR that can be stored on disk.

Other projects addressed the portability problem at a similar level as well. The Architecture Neutral Distribution Format (ANDF) [53], by the Open Software Foundation, started in 1994 with a goal to enable portable binaries to be distributed and run on conforming UNIX systems, independent of the underlying hardware platform, but it was never put in practice. The ANDF language was similar to a compiler IR with no target dependence at all, different from the LLVM IR, operating at a higher level.

2.4 ISA translation

We discuss translation technology by starting with the approach used in the ArchC project [115]. ArchC is an Architecture Description Language (ADL) [57,58,67,72,96,104] framework that notoriously eases the building process of Instruction Set Simulators (ISS). It also helps as an ISA modeling tool. This kind of simulation is especially important for OpenISA because it mimics only the high-level behavior of the program and its instructions, while ignoring microarchitectural details. This technology is straightforward to apply in binary translation since the high-level behavior of each instruction is enough information to build a translation tool.

A typical processor model in ArchC has 3 main files. In MIPS [82], for example, you would structure your project as follows: `mips.ac` describes high-level hardware structures, including the number of registers, word size in bits and cache hierarchy; `mips_isa.ac` contains instruction formats, definitions and encodings; and `mips_isa.cpp` has C++ program snippets to implement the simulation of MIPS instructions when running a MIPS program.

However, standard ArchC simulators will not provide the best translation performance because they rely on interpretation to emulate the ISA. We call this *interpreted simulation*, an instruction-wise emulation where the overhead is typically high in comparison with native execution.

An alternative to interpreted simulation is Dynamic Binary Translation (DBT) [32,51, 75,100,109,124,130]. Aiming at reducing the decoding overhead, DBT techniques identify regions of instructions and translate them as a whole. Once the simulator reaches the address of a sequence of instructions, called region, it computes the change of state caused by an entire region instead of a single instruction. While this translation is more complex and slower to accomplish, once done, its result is cached and, if the region is executed

Name	Guest	Host	Benchmark	Performance
ArchC [31] (interpreted)	Mips	x86	Mibench	36 to 56 MIPS
ArchC [31] (interpreted)	PowerPC	x86	Mibench	28 to 43 MIPS
ArchC [31] (interpreted)	ARM	x86	Mibench	17 to 24 MIPS
ArchC [31] (interpreted)	SPARC	x86	Mibench	27 to 37 MIPS
Bochs [87] (interpreted)	x86 (all)	x86 (all)	SPEC CPU2006 (int)	31 to 95 cyc/ins
Bochs [87] (interpreted)	x86 (all)	x86 (all)	SPEC CPU2006 (fp)	64 to 213 cyc/ins
ArchC + LLVM [64]	Mips	x86	Mibench	2.8x to 169x
ArchC + LLVM [129]	ARM	x86_64	SPEC CPU2006 (int)	510 MIPS
Valgrind [99]	x86_64	x86_64	SPEC CPU2000	4.3x s.t.n.†
Pin [93]	x86	x86	SPEC CPU2000 (int)	1.54x s.t.n.
DynamoRIO [126]	x86	x86	SPEC CPU2000 (int)	1.42x s.t.n.
Pin [93] (counting BBs)	x86	x86	SPEC CPU2000 (int)	2.51x s.t.n.
DynamoRIO [126] (ditto)	x86	x86	SPEC CPU2000 (int)	5.08x s.t.n.
QEMU [38]	x86	x86_64	SPEC CPU2006 (int)	5.9x s.t.n.
QEMU [38]	ARM	x86_64	SPEC CPU2006 (int)	8.2x s.t.n.
LNQ [77]	x86	x86_64	SPEC CPU2006 (int)	4x s.t.n.
LNQ [77]	x86	x86_64	SPEC CPU2006 (fp)	6.76x s.t.n.
HQEMU [74]	ARM	x86_64	SPEC CPU2006 (int)	3.4x s.t.n.
HQEMU [74]	x86	x86_64	SPEC CPU2006 (int)	2.5x s.t.n.
LLBT [120] (static)	ARM	x86_64	EEMBC	1.66x s.t.n.
HERMES [133]	x86	MIPS	SPEC CPU2000	2.66x s.t.n.
IA-32 EL [33]	x86	Itanium	SPEC CPU2000 (int)	1.53x s.t.n.
StartDBT [41]	x86	x86	SPEC CPU2000	1.09x s.t.n.

† stn. = slower than native

Table 2.1: Comparison table with the performance of different translation systems

multiple times, DBT techniques surpass the performance of interpreted simulators [48, 108].

There are different techniques and trade-offs regarding the region identification [55,78, 80,81,125] and the quality of the region translation [23,28,46,50,68,105], providing a rich set of choices when enhancing a simulator with DBT. However, interpreted simulation can be important. If blocks that do not have frequent execution appear, DBT efforts to reduce the decoder overhead and its expensive compilation algorithms become harmful and unnecessary [41, 48, 49]. To mitigate the overhead in these cases, DBT can be combined with interpretation as in the CMS by Transmeta [51].

2.5 Performance supported by current technologies

Before discussing the goals in emulation performance of a new ISA, we must evaluate the performance achieved by translating regular ISAs with current technologies. We will first discuss the performance of interpreted simulators and then show how DBT techniques may enhance them.

Table 2.1 presents each ISA emulation technology along with the benchmark used to measure the slowdown relative to the native execution. Some papers do not report the

slowdown relative to the native execution, but use other metrics, which can be either millions instructions per second (MIPS), the number of guest instructions executed per second on the host machine, or cycles per instruction, the number of host cycles spent to emulate a single guest instruction. For example, line 6 shows that the Bochs emulator is able to run x86 programs on x86 machines by using interpretation at the speed ranging from 64 to 213 host cycles per emulated guest instruction in the floating point programs of SPEC CPU2006.

2.5.1 Intepreted performance

Table 2.1 shows that interpreted simulations suffer a big impact on performance, as expected. While the table reports ArchC performance in MIPS, we performed a simple experiment to detect ArchC simulators slowdown, which figured around 50x when emulating MIPS on an Intel Q6600 at 2.4GHz. Bochs [87] is an x86 interpreted simulator able to emulate the entire system and boot operating systems. We partnered with Cesar et. al. and evaluated Bochs performance in detail in the paper [48] we published in the 2013 IEEE International Symposium on Workload Characterization (IISWC 2013), showing that the simulator is able to emulate integer instructions at a rate of one guest instruction every 45 host cycles, on average. However, the host machine is capable of executing more than one host instruction per cycle, hence, we conclude that even high-performance interpreted simulators such as Bochs are not good enough in comparison with native execution.

2.5.2 Compiled simulation

Seeking to improve ArchC simulators, Garcia et al. [64] worked on enhancing the ArchC 2.1 simulation engine to support static translation in a technology coined *compiled simulator* [52,103,114,131], building on top of the work of Bartholomeu [34]. In this technique, a tool converts guest binary code into C++ source code and leverages the native compiler to translate the code. After compiling this source code, the simulator runs with a much higher speed because the compiler statically optimizes the execution path of the guest program. It is thus a compiled program instead of an interpreted one. They use their own region formation technique with gcc 4.4, achieving slowdowns as low as 2.8x for the sha program and as high as 169x for the fast fourier transform (FFT). Pvrikryl et al. [107,108] uses a similar region formation technique to enhance the ADL ISAC simulator engine and show it in a mathematical formal framework, using a different term called *translated simulation* to name the same technique. Their technique achieves up to 110 MIPS in bitcount, from Mibench, when simulating the MIPS architecture. Since they ran their experiments on an Intel Core 2 Quad at 2.8GHz that runs native x86 programs at a far greater rate of instructions per second, this translation arguably still suffers a considerable overhead in comparison with native execution.

2.5.3 Retargetable dynamic binary translation (DBT)

A known issue with compiled simulators is the long time required to statically compile the entire binary and the need to redo this process every time a new guest program

needs to be simulated. Moreover, similar to static binary translation, they cannot handle programs with self-modifying code (SMC). To overcome these problems, Garcia et al. [63] presented a retargetable DBT approach for ArchC 2.2. They stick with the same region formation technique, but change the compiler to LLVM 2.8 to allow the dynamic just-in-time code generation, achieving up to 139 MIPS, on average. Wagstaff et al. [129] further enhanced their own variant of ArchC with LLVM 2.9, similarly to Garcia et al., building a retargetable DBT engine. They were able to achieve, on average, 510 MIPS for an ARM target with EEMBC and SPEC benchmarks. Their technique surpassed the performance of SimIt-ARM [111], a fast ARM ISS, by 1.7 times. They performed their experiments on an Intel Xeon X5650 at 2.67GHz and, thus, 510 MIPS compared to the regular 2670 MIPS that this host processor is able to execute using an underestimated IPC rate of 1 suggests that the translation overhead is still large, if we compare it with native execution.

2.5.4 High-performing retargetable DBTs

The compiled simulator and just-in-time compilation techniques presented so far show a performance far from native. Hong et al. [74] created HQEMU in an effort to enhance QEMU [35, 38] with LLVM and try to achieve near-native performance. For fast translation of cold regions, the framework employs the standard QEMU tiny code generator (TCG) [10], while LLVM runs on another core to aggressively optimize traces of hot regions. The geometric mean of the overhead compared to native execution is 2.5x for x86 emulation on x86-64 (almost same-ISA emulation) and 3.5x for ARM emulation on x86-64 (cross-ISA setup), with an i7 3.3ghz as a host machine. This same work also evaluates the performance of QEMU as a base line, reporting 5.9x on the same-ISA emulation setup and 8.2x on the cross-ISA setup. Prior to HQEMU, LNQ [77] was the first attempt to integrate LLVM and QEMU, but achieved more modest results ranging from 4x to 6.42x slowdown ratios. In a more recent work, HERMES [133] proposes to drop the architecture of QEMU in favor of a host-specific data dependency graph, which allows to explore optimizations at a representation that is closer to the host instead of the generic IR of QEMU. HERMES achieves the performance of, on average, 2.66x slower than native for SPEC CPU2000 programs, which is excellent for a cross-ISA translation.

2.5.5 Other approaches

In a completely different approach, Kaufmann et. al. [83] tried to implement a widely portable ISS that generated Java bytecode to be just-in-time compiled by a Java Virtual Machine (JVM), leveraging the established Java architecture that runs on many platforms. This technique is similar to compiled simulation or just-in-time compiled simulation, except that it used Java instead of LLVM. They reported, on average, to achieve 78% of QEMU performance on EEMBC AutoBench 1.1 [106].

2.5.6 Specialized DBTs

Finally, one of the best performances we see in literature is due to IA32-EL by Baraz et al. [33], an ISA translator that runs x86 guest programs on the discontinued Itanium [119] architecture. They built a specialized DBT engine that runs x86 programs, on average, 1.53x slower than native Itanium programs, albeit their DBT is focused on only a specific guest and host machine pair. Similarly, DynamoRIO [43, 44, 126] performs same-ISA dynamic binary translation to achieve slowdowns as low as 1.42x, even though they are restricted to x86 emulation on x86 hosts. An Intel counterpart called StarDBT [41] performs x86 same-ISA translation with only 9% of overhead, on average, for SPEC CPU2000. In the category of static cross-ISA translators, LLBT [120], a static binary translator based on LLVM, achieves cross-ISA translation (ARM to an Intel Atom) with 66% of overhead, on average, for the EEMBC benchmark.

2.6 Dynamic binary instrumentation

ISA emulation, despite providing portability, is also performed to provide a virtual machine environment where a program is carefully analyzed at the architecture level. Since designing custom companion hardware circuits that collect statistics about program execution has a prohibitive cost, the emulation, the software counterpart, provides a flexible framework at the cost of performance. For example, Qin et al. [110] and Moreira et al. [97] both study a framework for real-time taint analysis that checks for security attacks by augmenting a virtual machine framework. Valgrind [99] is a memory check tool that provides helpful information to the programmer to identify memory leaks in C code. For Valgrind, there is an extensible framework for dynamic program analysis. Nethercote et al. [99] report the performance overhead of Valgrind to be 4.3x, on average, using the nullgrind extension. This extension does not perform any analysis and was created to allow the measurement of the Valgrind virtual machine overhead in isolation.

Simulators are also used to study architectural statistics about program execution, e.g. ArchC simulators. ESESC [21] is a multicore systems simulator (shared or non-uniform memory) that uses sample-based information retrieval to estimate energy and chip temperature. To speed up this simulator, Ardestani et al. [21] used QEMU to run the code that is not instrumented. On the other hand, simulators that aim at collecting detailed architectural information typically share a poor performance when compared to the fastest virtual machines. ESESC achieves 9 MIPS, which is slower than ArchC interpreted simulators.

2.7 Summary

On one hand, HLL frameworks such as Java can achieve good performance, but constrain programs. On the other hand, binary translation translates any machine language program, but may struggle to achieve a good performance. Now that we have a solid overview of the state-of-the-art performances of ISA translation, we know that despite the advantages of binary translation, there is a typical overhead when comparing the translated

program performance with native performance.

Previous work have shown that is is possible to emulate guest code at near-native performance with DBT when the guest and the host ISA are the same [41]. However, previous results also indicate that when the guest and host ISA are different the emulation overhead is typically high [35, 38, 74, 77, 111, 129, 133]. In this thesis, we argue that the problem lies in the fact that there are ISA features that are hard to emulate when the guest and host ISA are not compatible and we show that key ISA decisions can change this.

Chapter 3

OpenISA overview

OpenISA is a free, well-defined, emulation friendly and low-level language, an ISA, designed to free users from processor vendors and foster development of current and new processor architectures. It has two parallel goals for software and hardware, (1) to foster simplicity in software deployment by providing a common ISA whose programs compiled to it can be executed on OpenISA processors or easily translated to other hardware platforms and (2) to provide a flexible model for processor interfaces that allow them to evolve without a strong commitment with past versions and backwards-compatibility.

This chapter explores the first goal, while the latter will be explored in the next chapter, which discusses the OpenISA encoding.

3.1 OpenISA objectives

The defining characteristics of OpenISA are detailed below:

- Emulation friendliness: the ISA must be emulation friendly to allow its efficient emulation on current off-the-shelf microprocessors, such as x86 and ARM, without any hardware specialization requirements.
- Low-level: a low-level language uses operations that represent direct functions supported by a processor. While it is implicit that an ISA must operate at a level lower than other languages, there are some ISAs, so called *virtual ISAs*, that were not designed to be implemented by real hardware. The OpenISA goal, instead, is to limit itself to hardware-synthesizable features just like a regular processor. It also introduces the concept of *hybrid ISA*, since some instructions may be left out of the processor implementation as well.
- Well-defined: a well-defined and concise ISA is important to prevent different behaviors when implemented by different hardware and virtual machines. This ensure the utmost level of compatibility between platforms, that of bug-compatibility, essential to ease the task of deployment in a write once, run everywhere paradigm.
- Free: OpenISA makes use of a permissive license to allow its free use.

OpenISA makes it possible for users to seamlessly switch their software subsystems to run on different virtual machines and/or different processors. Current processor manufacturers can support it by designing new OpenISA based microprocessors or OpenISA virtual machines tailored to run on their proprietary ISAs. OpenISA enables new players to compete in the processor industry, fostering innovation and competition.

3.2 Design trade-offs

In this section, we analyze language elements that are available at other forms of program representation but are not present at the typical ISA layer. Some of them are important to enable high-quality translation to other architectures. It is also discussed why they were or were not adopted in OpenISA, stressing the tradeoff of fair translation and ISA complexity.

Let's discuss two distinct approaches to program translation from a *guest platform* to a *host*: *recompilation* and *binary translation*. Recompilation handles the translation from an abstract model such as a compiler Intermediate Representation (IR). In many ways, it proceeds just like a regular compilation, with the difference that it is recurring a second time. For example, a user compiles a program to Java bytecode, and the bytecode is recompiled to a host. Since the bytecode is high-level, there is no need to perform binary translation and the result of recompilation exclusively uses native host mechanisms. In binary translation, the hardware platform for which the program was originally compiled for may use low-level features that are unavailable in the host, forcing software emulation. Most noticeably, binary translation needs to emulate the memory layout established by the guest program.

OpenISA relies on binary translation of a low-level model, but to first understand the differences of the competing, abstract, high-level language model, the list below provides an overview of basic entities used in an abstract model that supports the translation of programs to run in real hardware:

- Explicit functions: this entity is partially supported in hardware via special instructions such as *calls* and *returns*, therefore, if it is exposed in the program representation, the translation to the machine level can be made more efficient;

- Explicit stack, activation record or local variables symbol table: any structure that helps the identification of locals and memory locations that can be promoted to registers – important information to manage what lives in registers and what must be escaped to memory;

- Abstract memory layout: both stack and heap layout will be rewritten to conform with the host platform. If the language does not expose raw memory elements, it allows the compiler or binary translation system to reformat data in the most suitable way to a particular host;

- Data types: the perfect high-level, high-performing language will either stick to types that are directly supported by most processors or be vague enough to allow

the underlying system to implement them with the most suitable native type. There is often a high cost associated with emulating the behavior of a non-native type. For example, Intel x86 may use a non-standard 80-bit floating-point type to represent *doubles* in C, but the language is vague enough to allow this. The *long* data type in C may be either a 64-bit number or a 32-bit number, depending on the host.

Next subsections details the role of each of the high-level entities discussed so far.

3.2.1 Explicit functions

High-level languages provide a method to isolate code into different functions, which is not available at the ISA level. However, there is support for functions in hardware: call and return instructions, as well as instructions to support the implementation of the program stack, which provides efficient allocation and deallocation of limited-scope data. At the microarchitectural level, some processors also use a buffer of return addresses to ensure that the code flow change caused by returns does not stall the pipeline [122].

The most important aspect of functions is that it is the fundamental granularity at which compilers operate and at which programmers write code. Programmers are used to inline a performance critical function. They make the assumption that it will run faster if it avoids the creation of the stack frame associated with a call to a different function. Optimizing compilers focus on improving the code at a function body and may not be capable of doing a good job optimizing code that is scattered at different functions.

Since the function scope is lost at the ISA level, it is tricky to define a good region of code to operate on and perform local optimizations that will enhance the translation from guest to host. For instance, Garcia et al. [64] study region formation techniques that does not work with functions, but with traces. Hong et al. [74] report that augmenting traces to form larger regions with a more complex control flow graph (CFG) is an integral technique to guarantee good performance in their translator.

For the experiments in this thesis, it is enough to know function boundaries to ensure translation quality. In fact, as discussed by Zinsly [134], it may be possible to dynamically identify regions of code equivalent to functions by looking for instructions that support the execution of functions, such as calls and returns.

3.2.2 Explicit stack

The program stack is tightly associated with the execution of functions. However, knowledge about the latter is not enough to identify the former. The stack is responsible for storing the local variables that will be manipulated by different basic blocks of the function and that are crucial to perform data-flow optimizations. If this information is lost during the translation process, it will not be possible to further apply such optimizations.

A second implication of not knowing the stack is that, in the compilation process, the compiler may have spilled variables and transfered them from registers to stack positions because of the limited number of registers in a given processor. On the other hand, the host processor may have enough registers to promote some of these variables to registers. The translation to the host machine, therefore, could be forever bound to a poor performance

because the source ISA lacked enough registers, a crucial variable was spilled and the translator does not have enough information to promote the memory position to a register.

3.2.3 Abstract memory locations

If the exact memory positions associated with each datum is not yet assigned, the translator is free to rearrange data and build the best memory layout to any target. However, this is not the case at the ISA level, where the memory layout was already built and the translator system is constrained to access data at specific memory addresses.

This distinction is important enough to be used as the main aspect to distinguish between *recompilation* and *binary translation* in this thesis: if the system works with abstract memory positions and is allowed to modify the final memory layout, we say it is recompiling, otherwise, performing binary translation.

In this work, we stick with binary translation because a predefined memory layout is mandatory to allow the highest level of compatibility: *bug compatibility*. The memory layout specially affects whether or not existing silent bugs can be exposed when translating a program. This is the case when you have a buffer overflow [73]. If the variable allocated right after the buffer is not used, the bug (changing the variable value) may not affect the execution, hence it is silent. But if one allocates an important variable right after the buffer, overflowing it may change the behavior or break the program.

3.3 OpenISA virtual machine

The OpenISA virtual machine is the core technology that enables portability of OpenISA programs to other architectures. Figure 3.1 shows different options for the level at which the virtual machine may be inserted: user or system level. This work focuses on the user level virtual machine.

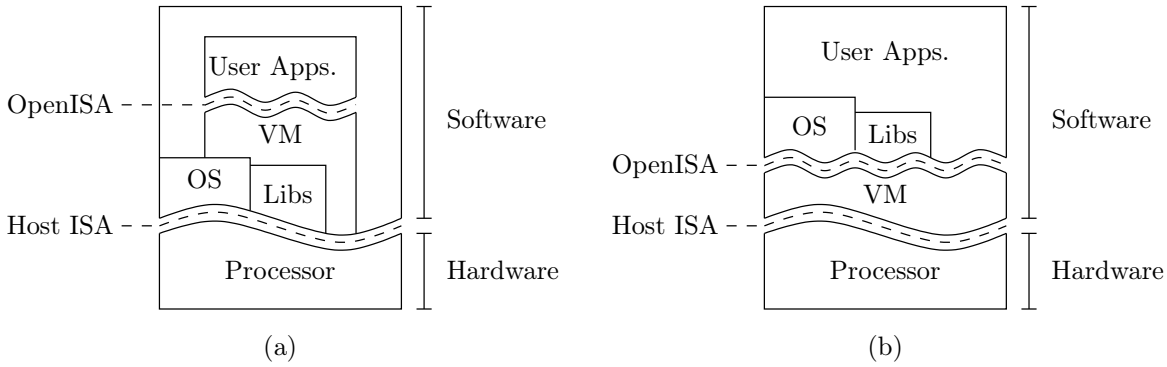


Figure 3.1: (a) OpenISA user level applications running on top of a process virtual machine. The OS and supporting libraries are running directly on the host ISA. (b) OpenISA user and system level applications running on top of a low level system virtual machine. In this case, the whole software stack is running on top of the VM.

The virtual machine translates guest OpenISA instructions to host instructions, and the associated overheads of such system have different causes. Below we present a list of

overheads identified by Borin and Wu [41] in StarDBT, a virtual machine that operates with x86 over x86 (same ISA).

1. Initialization overhead, usually negligible;
2. Cold code emulation overhead, which can be reduced with fast interpretation;
3. Profiling and hot code building;
4. Hot code emulation overhead, which can be separated into overhead due to code cache control transfer overhead [71] and code quality.

The code quality factor is not a problem for same-ISA virtual machines because the code is usually the same, except for privileged instructions that require emulation. In the case of OpenISA, however, its translation occurs to different ISAs. In this scenario, code quality can easily become the prevalent overhead because a naïve translation of instructions from a different ISA can introduce bad quality code that performs poorly.

Since Borin and Wu already point out promising techniques to address the slowdowns of a same-ISA DBT, but does not look at the code quality factor that is of prime importance to the problem analyzed in this PhD thesis, we focus our attention on building an ISA that improves this code quality, aiming at achieving the same slowdowns of same-ISA virtual machines. To study how to improve code quality by design of the ISA, we built a prototype virtual machine on top of an existing compiler, the LLVM compiler. This existing infrastructure allows us to check, with existing compiler technologies, whether it is possible to emulate OpenISA programs on other ISAs with minimal overhead.

Next, we present different design possibilities of the translation prototype and discuss why we picked one of them.

3.3.1 Using the LLVM I.R. as the starting point

A good starting point for OpenISA would be to define its first ISA version to be exactly the same as the LLVM I.R. Since it is a well-documented static compiler I.R., OpenISA would start with all of these benefits. Future versions advancements through incremental development would remove features down to the point where OpenISA becomes a well-defined low-level ISA.

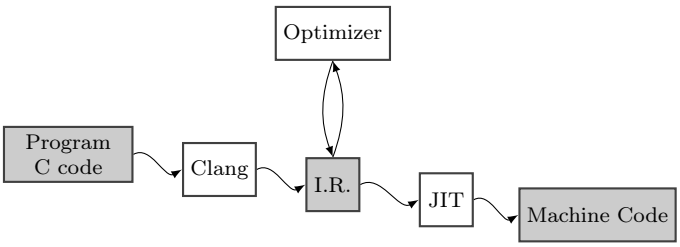


Figure 3.2: Diagram showing the usage of LLVM as the starting point for OpenISA

The problem lies in directly changing LLVM code and its tools to comply with removed features as part of the incremental development plan. Each time some aspect of the LLVM

I.R., at this point OpenISA ISA, needs to be changed, all tools that rely on reading the LLVM I.R. would break and could need maintenance to comply with the newest changes. Since LLVM is a big project, it is an impractical approach. Moreover, the original LLVM project continues to evolve and this particular OpenISA fork would soon become outdated.

3.3.2 Using LLVM indirectly

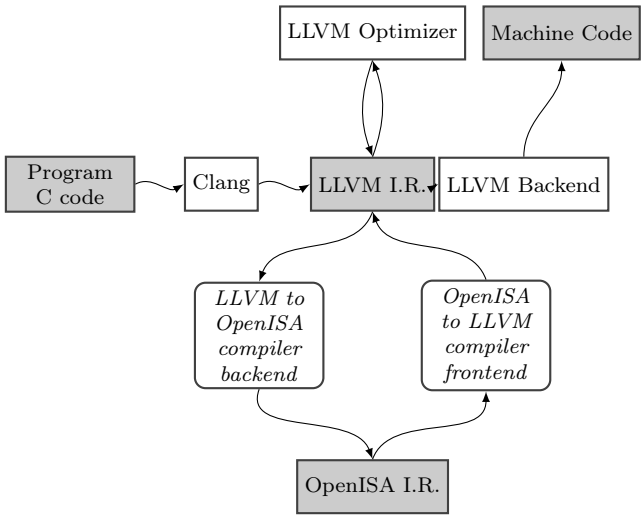


Figure 3.3: Diagram showing the relationship of OpenISA and LLVM

In order to tackle the previous problem, it is possible to develop two new components that interacts with the LLVM infrastructure and link a separate OpenISA project to the LLVM project, which is free to evolve independently in this manner.

Figure 3.3 illustrates this principle. The two rounded boxes show the components that link OpenISA to LLVM. In this new design, OpenISA starts as a RISC clone in its first version. As the OpenISA changes to embrace new features or to remove characteristics to allow easier translation, only the LLVM OpenISA Backend and the LLVM OpenISA Frontend needs to be updated in order to OpenISA to take benefit from all LLVM features.

Even though, in this design, OpenISA does not have tools that directly operates on it, any OpenISA fragment may be translated to the LLVM I.R. at the cost of the LLVM frontend execution. This brings the benefits of LLVM static compilation to OpenISA:

- High-quality code generator;
- Targets any machine whose backend is available in LLVM;
- May use any optimization implemented for LLVM.

3.3.3 The Complete Framework

The complete framework[1] implemented to design OpenISA is depicted in Figure 3.4. The nodes show system components, including the ones borrowed from the LLVM framework

[1] Available at `http://github.com/rafaelauler/openisa`

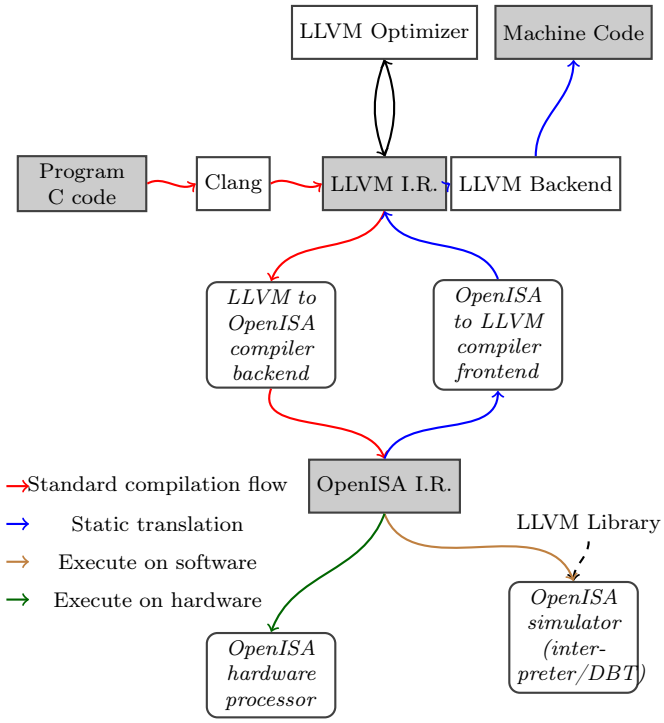


Figure 3.4: Diagram showing the complete OpenISA framework

to provide both the compiler from C code to OpenISA code and the static compilation framework to translate from OpenISA to any other machine language (red and blue paths shown on the diagram). Once the user program is translated to the OpenISA language, she can either run it on an OpenISA processor (green path) or run it on any other host machine using the OpenISA virtual machine (brown path). Alternatively, the user may statically translate it to another machine language (blue path).

The real OpenISA processor implementation is a long-term goal. The primary goal of the OpenISA project is to develop the OpenISA virtual machine, which employs interpretation and dynamic binary translation techniques to run OpenISA programs on any platform. However, it is essential to first develop the LLVM backend and frontend to support the design of OpenISA with a careful study enabled with the help of LLVM. The final virtual machine may, but is not required to, use LLVM libraries to aid in the task of translating OpenISA fragments to host machine code. The virtual machine may also borrow code and techniques from the LLVM OpenISA to LLVM frontend component. As an example, the blue path (static compilation) could be adapted to support the virtual machine dynamic binary translation process.

3.4 Summary

This chapter outlined goals and design constraints of OpenISA: it is a free, well-defined and low-level language that supports its execution on a real processor, can be translated to other platforms, and has a flexible interface such that a sequence of processor generations can evolve without a strong commitment with backwards compatibility. The latter will be detailed in the next chapter. This chapter also identified useful components of a high-

level language that makes the translation to any machine language effective, analyzed how this may affect OpenISA by lacking some of this information and finally presented the framework developed to explore the design of the OpenISA ISA.

Chapter 4

OpenISA encoding and recycling mechanism

Perhaps the most well-known aspect of an ISA is its encoding. Is it fixed-length or variable-length? How many instructions can be encoded? How many registers can ALU instructions refer to? Which addressing modes are available? Is it inspired by a RISC design or is it inspired by a CISC design?

This chapter elaborates on the design of the OpenISA 32-bit encoding and summarizes our experience and views reported in two previously published papers: one discussing the recycling mechanism along with a discussion about encodings presented in ACM ISCA 2015 [92], written in collaboration with Bruno Cardoso, Luiz Ramos and Rodolfo Azevedo, and another about an OpenISA virtual machine for resource-constrained devices, including an 8-bit microcontroller, published in the computer architecture workshop WSCAD 2015 [95], written in collaboration with Carlos Millani and Alisson Linhares.

4.1 Word size and endianness

There are two ISA parameters with strong impact on the ISA emulation performance on different hosts: *word size* and *endianness*. These special guest ISA parameters control characteristics responsible for changing instructions semantics in such a way that may demand a large effort from the host to emulate the guest ISA.

Word size is problematic for emulation because the address size is often tied to it, requiring all memory operations to work with arithmetic operations of this size. Once the word size of the guest ISA is larger than that of the host ISA, emulation performance is significantly affected. The number of host instructions required to emulate each guest instruction will suffer with a large penalty as the host employs extra instructions to increase its datapath bitwidth.

The host overhead when simulating a larger datapath is the result of mismatching abstractions used in the guest ISA and host ISA. The necessity of a specific address space size, as in a 64-bit flat memory addressing, is a consequence of the size of the program and the size of its maximum memory working set. The OpenISA ArchC-based interpreter, for example, simulates a system with 512MB of primary memory and allocates this space in a

contiguous pool of addresses – this program cannot be compiled to a 16-bit ISA with flat addressing model (non-segmented) because it uses too much memory. Even though it is a demand of the program, word and pointer sizes are specified in the ISA, rendering some programs impossible to be encoded in small-word-size ISAs, except when some tricks are used to expand the address space such as in the old x86 segmentation mechanism.

Similarly, *endianness* is another property with high impact on emulation performance. Once there is a mismatch between host and guest regarding this property, emulation will suffer a high degree of overhead to compensate for it.

Both *word size* and *endianness* are orthogonal to all other characteristics of the ISA, but they have a high impact on emulation performance. The most reasonable approach is to define multiple versions of OpenISA, one for each tuple *word-size* - *endianness*.

Throughout this work, we assume OpenISA is a 32-bit little-endian architecture. The reader may assume other versions are available to better fit other host architectures, or accept a larger overhead when running on big-endian hosts or hosts with a smaller datapath width.

Both aspects are exclusive of programs represented in levels as low as the ISA. This is a trade-off between representing the program in a low-level format, specifically one in which the memory layout is already defined, and representing it in a high-level format in which it is not possible to derive anything about the memory layout, such as in Java. OpenISA chooses the low-level approach in order to avoid leaving undefined aspects in the program representation, which undermines bug-compatibility and makes debugging harder across different platforms. This happens because the same bug may manifest differently in two platforms, since in high-level representations, the final memory layout is decided based on the platform. OpenISA also chooses this level to be language-independent. In order to code in C, for example, the underlying program representation must expose the memory layout and pointer sizes. If we choose to hide these parameters from the programmer, we are leaving one of the most popular programming languages out of the scope of supported platforms, diminishing an important goal of OpenISA that is to be platform-independent.

Finally, as different designs increasingly converges to the little-endian organization with 64-bit word size, one should expect this issue to be less relevant with time.

4.2 Encoding

4.2.1 Instruction sizes

OpenISA uses a fixed instruction size of 32 bits in order to allow the efficient implementation of the decoder in a multi-issue microarchitecture, in which more than one instruction is decoded per cycle. If the instruction uses variable-length, it is harder for the decoding unit to figure out when the next instruction begins, increasing the complexity of the pipeline stage that is responsible for decoding more than one instruction per cycle. Even though the fixed instruction length became a requirement of OpenISA, the binary translation experiments completed in the scope of this thesis showed that long immediates can be a problem.

Long immediates in fixed-length architectures are encoded either with a load instruction addressing a constant pool or with multiple shift-add instructions that support a smaller immediate. To make emulation of the programs as easy as possible, the load alternative was ruled out because it promotes a constant to a mutable variable in a memory position, reducing the scope of optimizations that can be applied. Therefore, in OpenISA the immediate is encoded in the instruction.

In immediates encoded in multiple instructions, it is common for the compiler to (1) schedule the two instructions far apart or (2) apply common subexpression elimination and share one of the instructions in more than one immediate-loading combo of instructions. Both actions contribute to make it harder for the translation engine to recover the original immediate. While the ideal case would be to encode the entire immediate in a single instruction, this is not possible if the immediate is 32-bits wide. Therefore, OpenISA uses two 32-bit instructions that only have a valid semantic when scheduled together, and part of the immediate is encoded in one instruction and the rest in the second instruction. This is one of the exceptions to the 32-bit size of instructions, since this instruction has an effective size of 64 bits. The other exception is the indirect jump instruction that also has 64 bits to encode extra information about the jump.

4.2.2 Instruction formats

OpenISA instruction formats are split into opcode and payload. Figure 4.1 shows an example: the encoding of a `load` instruction. The opcode comprehends the necessary bits to uniquely identify an operation, while the payload is used to encode operands. The opcode can be an arbitrary number, as long as it is unique, while operands encode information used as input to the operation. The line between opcode and operands in other ISAs can be blurry. For example, in ARM, `andeq` can be interpreted as the `and` opcode while `eq` is an operand that tells the processor to conditionally execute this instruction only when `CPSR`[1] (ARM status register) stored the result of a comparison that concluded two numbers to be equal. It can also be interpreted as a single opcode `andeq`, in which `eq` is not an operand but part of the opcode. In OpenISA, each opcode has a unique mnemonic in assembly language as well and is not just a didactic separation of concerns. The opcode division from operands is important because it delimits the unit of recycling in case this instruction is ever removed from the ISA. We will further elaborate on this idea in the recycling section.

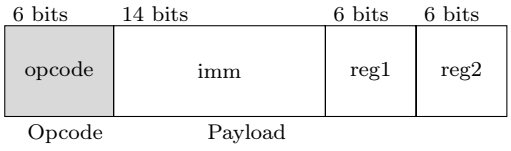


Figure 4.1: Load instruction encoding diagram

Similar to the load example, all formats begin with the opcode and end with the payload. For example, if an instruction needs a payload of 12 bits to encode two register

[1]Current program status register

operands, its opcode will be encoded in the higher 20 bits while the payload will be encoded in the lower 12 bits. Thus, the smaller the payload, the larger the opcode space. In the two-registers example, it would be possible to encode 2^{20} different instructions with 12 bits of payload if the entire OpenISA opcode space is used to encode only this type of instruction. In reality, the total number of opcodes given the payload is slightly more complicated to calculate.

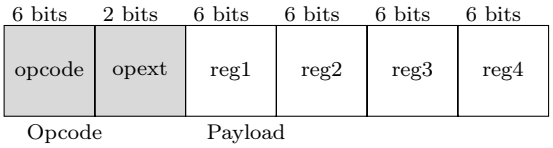


Figure 4.2: Mul instruction encoding diagram

Consider now that we need to encode the `mul` instruction, shown in Figure 4.2, which multiplies the contents of two registers and puts the higher 32 bits in a register and the lower 32 bits of the result in another register. This uses a total of 4 registers, or a payload of 24 bits, leaving 8 bits for instruction identification. However, we do not have the entire opcode space of 8 bits (or 256 instructions) to encode instructions that use a payload of 24 bits because, as we learned in Figure 4.1, the first 6 bits may be used to encode instructions with 26 bits of payload as well. Therefore, in the diagram of the `mul` instruction, a second gray box named an "opcode extension" was drawn to make it clear how the opcode space is fragmented.

Opcode and opcode extension fields are called *opcode fields*, for brevity. Any given format may use one or more opcode fields until the desired payload size is achieved. In the decoder perspective, any given opcode field may either identify a unique instruction or specify that the next opcode field specifies the instruction. In the latter case, we call this opcode field *prefix bits*.

4.2.3 Opcode space utilization

Payload size	Usage rate	Used encodings	Possible encodings
26 bits	59.37%	38	64
24 bits	44.44%	16	36
20 bits	16.96%	19	112
18 bits	76.92%	40	52
16 bits	55.00%	11	20
12 bits	55.21%	53	96
6 bits	1.56%	2	128
0 bits	4.69%	3	64

Table 4.1: OpenISA opcode space utilization by payload size

Table 4.1 shows the opcode space utilization[2] for OpenISA according to the encoding scheme discussed in the previous subsection. The total number of instructions is 139 and most of the core instructions semantics were based on MIPS, with the exception of operand sizes. It is important to first analyze the most spacious payloads first. If there is

[2] We made available the tool that calculates this information for any ArchC model at http://github.com/rafaelauler/ISAStatistics under GPLv3.

no room for new instructions with a payload of 24 bits, for example, it is always possible to mutate an opcode of a 26-bit-payload instruction into *prefix bits* to four 24-bit-payload instructions.

According to this reasoning, the first line of the table says that there are 26 available 26-bit-payload opcodes, or 104 24-bit-payload opcodes, or 1664 20-bit-payload opcodes, and so on. Notice that we cannot state that we have a fixed number of free opcodes until we know exactly which payload size must be encoded.

The most important opcode space is that of the 26-bit-payload instructions, which is 59.37% occupied. It is the most important because it has more payload bits that we can mutate into opcode fields to encode more smaller-payload instructions.

4.3 The recycling mechanism

4.3.1 Overview

The recycling mechanism is thoroughly described in the fourth section of the paper *SHRINK: Reducing the ISA complexity via instruction recycling* [92] and in the patent application BR 10 2015 005838 (pending). The conference paper focus on the use case of increasing the efficiency of the variable-length encoding of Intel x86 by discarding old opcodes. However, this mechanism is fully incorporated into OpenISA as well, despite being an ISA with a fixed-length encoding, in order to avoid ever running out of opcode space to encode newer instructions, to avoid being stuck with a mistaken design, to allow for ISA extensions and to make it more flexible as a true *hybrid* ISA that can be both virtual and concrete. The recycling mechanism was extended to fit the use cases of OpenISA and is slightly different from the description used for x86.

The extended recycling mechanism works by assigning an ISA version to the software, called `SR` (*software revision*), specifying which ISA specification the software was compiled for, and ISA versions to the processor, called `PRs` (*processor revisions*), specifying which ISA specifications the processor implements. A single processor may implement multiple `PRs` if it supports more than one ISA specification. Different ISA versions are similar, except for unused or mistakenly designed instructions. Two consecutive ISA versions will never differ by a large amount of replaced opcodes and will never replace opcodes that are heavily used.

A regular execution occurs if `SR` matches one of the implemented `PRs` in the processor. If not, it means that hardware and software may disagree with respect to the semantics of a specific opcode. This opcode may have been recycled to be used by another instruction in a different version of the ISA. In this case, the hardware traps if the software ever uses an opcode for which processor and software disagrees about its current semantics. The exception handler then emulates the semantics of the instruction according to what the software expects.

The set of all ISA versions comprise the complete OpenISA virtual ISA, whereas the subset implemented by a given processor implementation is the concrete ISA. An OpenISA virtual machine is able to emulate the entire virtual ISA.

This mechanism does not apply only to fix mistakes with backwards-compatibility.

It also applies to gracefully enable forwards-compatibility to older processors when new extensions are released. If the processor does not implement any extension, it will emulate it in software.

Moreover, the version history does not need to be linear. Extensions may fork and merge in the future, similar to software version control. As long as the version numbers are different, the processor will correctly trap and emulate mismatching opcodes.

4.3.2 Mechanism description

The recycling mechanism is the process that allows a given operation to be dissociated from an opcode, be no longer supported and open the possibility to associate a new operation to this opcode. In this way, different than standard ISAs, an operation, for example, `ADD`, is no longer forever tied to a specific opcode, for example, `0x30`, but instead is associated with this opcode for a possibly limited time frame. Most instructions, however, will likely remain implemented and associated with an opcode throughout the entire ISA lifetime.

In this mechanism, each opcode has a revision number called `UR` (unique identifier revision). When this opcode is associated with a new operation or when it is dissociated from an operation, the `UR` is increased. When any `UR` in the ISA is changed, the ISA version is increased as well.

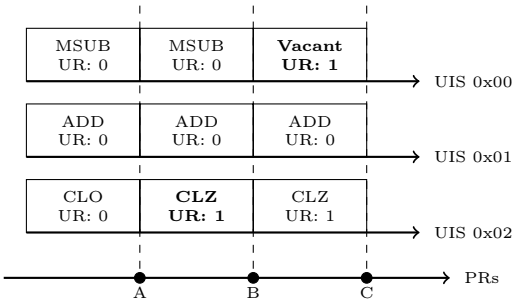


Figure 4.3: UR and PR evolution in the lifetime of an ISA

Figure 4.3 illustrates this process. In PR `A`, the first version of the ISA, the opcode 0x02 refers to the operation `CLO`, *count leading ones*. In PR `B`, the second version of the ISA, this opcode was recycled and is now associated with the operation `CLZ`, *count leading zeros*. The UR of the opcode 0x02 is now bumped by one to indicate that this opcode now encodes a different operation. Similarly, PR `C`, the third version of the ISA, removed the `MSUB` instruction from the ISA. Its old opcode 0x00 has its UR now increased by one.

The set of all URs at any time will uniquely identify the current ISA version. The set of all URs of the ISA version implemented by the processor is then memberwise compared to the set of all URs of the ISA version used by the software. Any differing URs will signal that a trap needs to be generated to emulate the correct behavior of the corresponding opcode. This is called the *trap mask*.

Extensions of the mechanism described in the original conference paper

1. The original conference paper suggests that the trap mask should be implemented in hardware. For the purposes of OpenISA, the trap mask should be able to be updated via software, allowing new ISA versions to be inserted and enabling forwards-compatibility for older processors to easily support emulating new extensions to the ISA.

2. The second modification of the original conference paper for the purposes of OpenISA is an extension to make it possible for a processor implementation to support in hardware two or more ISA versions, instead of just one. In this way, the trap mask is built by evaluating whether the URs for each opcode in the ISA version implemented by the software is different than the URs of this opcode in *all hardware supported ISA versions*. Furthermore, to correctly decode the instructions, the `SR` (software revision), that is the ISA version that the software was compiled for, is also an input to the instruction decoder instead of being an input just to the trap-generating hardware.

3. The third modification expands the number of bits that encode the `SR`. In the original paper, since it is an extension of the x86 instruction set, only 4 bits are dedicated to store different ISA revisions the running software may implement (see Figure 4.4). This decision takes into account the limited space in the x86 page table entry, where the `SR` is stored. While OpenISA processors should still use the page table entry to encode this information, it must use 6 instead of 4 bits, since it is intended that OpenISA uses the recycling mechanism more heavily than an ISA that can be implemented by only two processor manufacturers. Since OpenISA is free, it is expected that a larger number of forks can appear.

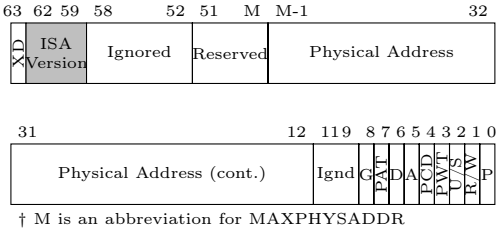


Figure 4.4: Format of the Intel IA-32e page table entry extended with SR

4.3.3 OpenISA formats philosophy

Notice that OpenISA formats are intentionally designed to be vaguely descriptive. The formats are referred by the number of payload bits instead of their functionality, such as in *Format of 26-bit payload* instead of *Format for jump instructions*. The motivation is to comply with the recycling ISA versioning philosophy: it is nicer to recycle an opcode whose format is limited to state the number of bits used for operands. Otherwise, we would end up in the future with an incongruent ISA where some unrelated instructions are encoded using, for example, *Format for jump instructions*. OpenISA specifically does

not imply there is any semantic relationship between two instructions sharing the same format other than *both use the same payload width.*

4.3.4 OpenISA versioning scenarios

This subsection presents three scenarios to illustrate real use cases where the OpenISA versioning scheme and the associated recycling mechanism shows themselves to be a robust solution in comparison with traditional ISAs.

Embracing multiple versions

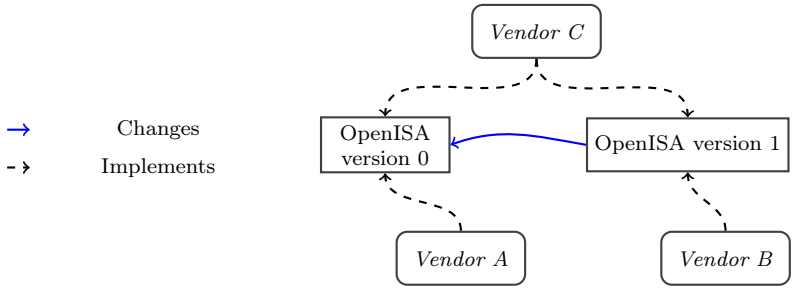


Figure 4.5: OpenISA versioning timeline for embrace scenario

Suppose vendor A and vendor B both want minimalistic cores. Vendor A uses instruction CLO (count leading ones, payload size of 24-bit), present in OpenISA core (version 0) since it is important for its cryptographic applications. Vendor B decides it will not implement this instruction, forks OpenISA and creates version 1, in which the opcode for CLO is recycled. It then implements 64 SIMD instructions (using 18 bits as payload to encode 3 registers) using the opcode previously used by CLO as prefix bits.

Vendor C creates a big-die implementation of OpenISA implementing both versions 0 and 1 in hardware. Figure 4.5 illustrates this.

Resulting scenario: all processor implementations run all versions via trap-based emulation. The processor marketed by vendor C runs all versions with maximum performance.

Merging two versions

Suppose again that vendor A and vendor B both want minimalistic cores. Vendor A uses instruction CLO (count leading ones, payload size of 24-bit), present in OpenISA core (version 0) since it is important for its criptographic applications. Vendor B decides it will not implement this instruction, forks OpenISA and creates version 1, in which the opcode for CLO is left unused. It then implements 64 SIMD instructions (using 18 bits as payload to encode 3 registers) using, this time, a vacant opcode previously unused by version 0 instead of the CLO opcode.

Vendor C creates a big-die implementation of OpenISA by merging version 0 and 1 into a new version 2, which incorporates both versions. Figure 4.6 illustrates this.

Resulting scenario: all processor implementations run all versions via trap-based emulation. The processor marketed by vendor C runs the latest version, which supports all instructions in hardware.

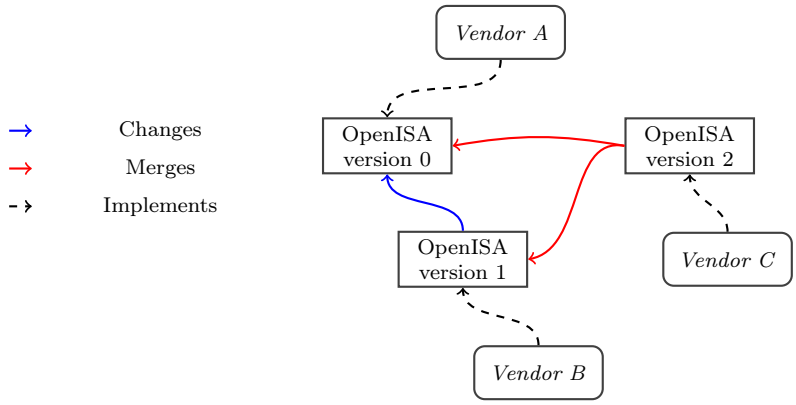


Figure 4.6: OpenISA versioning timeline for merge scenario

Phasing out past mistakes

For this scenario, suppose OpenISA has been established for 15 years. However, its SIMD instructions are now quite outdated as the community converged towards conditional execution for all SIMD instructions. Vendor A releases version 1 implementing new SIMD instructions with conditional execution in vacant opcodes, coexisting with the old instructions. After 10 years, the software industry has mostly converged towards the new way of doing SIMD in OpenISA. Vendor A now releases version 2 of the ISA, in which the opcodes used by the older SIMD instructions are now vacant. It now ships a leaner version of the processor that implements only the newer way of performing SIMD computation (version 2).

Resulting scenario: Full backwards-compatibility at all times. Processors successfully phased out a hardware extension that was short-sighted when created. ISA was redundant for a period of 10 years during the software adaptation period.

4.4 Summary

OpenISA encoding formats were co-designed with its versioning and recycling mechanism. It is a fixed-length format that eases the implementation of a multi-issue decoding unit, but also allows larger instructions, with sizes multiples of 4 bytes, in case more information needs to be encoded in a single instruction. However, each 32-bit part is handled as a separate instruction and its complete semantics may only make sense when all parts are read, such as in the load 32-bit immediate instruction.

The versioning and recycling mechanism make OpenISA, to the best of our knowledge, the first *hybrid* ISA, in which part of the specification may be emulated in software if needed. It also allows it to evolve while maintaining backwards and forwards compatibility at all times.

Chapter 5

Experimental framework

A general guideline used in the design of OpenISA is that it needs to be guided by experimental results and refined by iterative processes. Therefore, as important as the findings of this thesis is the methodology workflow used to arrive at them.

This chapter discusses the implementation details of the experimentation infrastructure[1], which was based on LLVM version 3.6 and on the original LLVM MIPS backend. Our public repository also features a Docker [40, 94] recipe to easily install, build and reproduce all experiments described here. We first published a version of this infrastructure in a paper [25] presented in the 2015 AMAS-BT workshop, co-located with CGO, HPCA and PPoPP 2015. This chapter, however, significantly extends and updates the contents of this paper.

5.1 Opening remarks

In general, it is not a good idea to translate a program all the way down to typical ISAs if you want to later run this binary on another ISA. For example, Table 2.1 from Chapter 2 shows that the best ARM-to-X86 static binary translator system, LLBT [120], will incur 1.66x of overhead, on average, to the translated binary.

An important goal of OpenISA is to change this scenario. The first question the experimental apparatus was designed to answer was whether it is possible to compile a program to an ISA while preserving its semantics in such a way that a later recompilation of it to another ISA would yield no performance losses.

The idea of this experiment design is to factor out any virtual machine overhead that is not attributed to the quality of the translated code (in which the guest ISA has impact). For example, the virtual machine initialization overhead as well as the hot code identification overhead, in a DBT system, will both exist whether the guest ISA is the same as the host (enabling a perfect translation) or not.

To this end, a static binary translator prototype, such as LLBT [120], was created. This is the most suitable design for our research because it uses an *offline* translation organization that provides us a virtually unlimited amount of time to perform optimizations.

[1] Available at `http://github.com/rafaelauler/openisa` and `http://github.com/rafaelauler/llvm-openisa`

The workflow this prototype provides us has the ability to answer which architectural feature loses semantic information. What was later found out is that it is not only possible to perform OpenISA translation with good performance, but also doable with a regular static compiler optimization pipeline, which means the user would typically have to wait no longer than the time a compiler takes to compile a program with optimizations.

5.1.1 A byproduct of the experimental framework

This static translator serves two purposes. It's not only a research apparatus to improve OpenISA, but its static translation capabilities have a direct application on another interesting problem: how a program distribution format based on OpenISA should look like. OpenISA's main goal is to have easy emulation on a virtual machine, such as a DBT system. Thus, although OpenISA is not primarily focused on static translation of its binaries, this thesis also covered all aspects of it. This is illustrated in the *static translation* path of Figure 3.4 of Chapter 3.

Our conclusion is that, despite the lack of ability to handle self-modifying code (SMC), the only major disadvantage of such approach to program distribution is that the program needed to preserve its full original image in the data section of the translated binary, increasing the static data in the binary. However, this is a restriction we cannot overcome if we want to reproduce memory layout bugs, since those bugs depend on the data memory being formatted in the same layout in all platforms. This is called a *low-level distribution format* in this thesis.

Subsequent versions of this prototype evolved to the status of a full static binary translator. For this format, we rely on an ELF binary envelope augmented with selected metadata information. We will further present the metadata by the end of Chapter 6, in Section 6.2.4. A real-world application and case study of the static translator is presented in Section 6.4 of the same chapter.

In the remaining of this chapter, we dive into the technical aspects of this experimental apparatus.

5.2 LLVM to OpenISA compiler backend

The LLVM library has enough components to allow the programmer to build a complete static compiler. It has several data structures to aid the entire compilation process. In this component, the OpenISA backend, we started with a regular MIPS backend and iteratively changed it to emit OpenISA code. At this step of the compilation, the input program is already converted to LLVM IR and the backend is responsible for converting LLVM IR to OpenISA code.

The Module, Function, BasicBlock, Instruction and Value classes are the basic in-memory representation of LLVM IR instructions in static single assignment (SSA) form. A module is the LLVM basic compilation unit that holds variables and functions definitions, similar to a .c file for the C language. Figure 5.2 shows an example of an LLVM module in textual form: line 1 assigns a name to the module (stored in memory in the Module class), lines 2-3 configures target-dependent data, line 5 defines a global string literal and

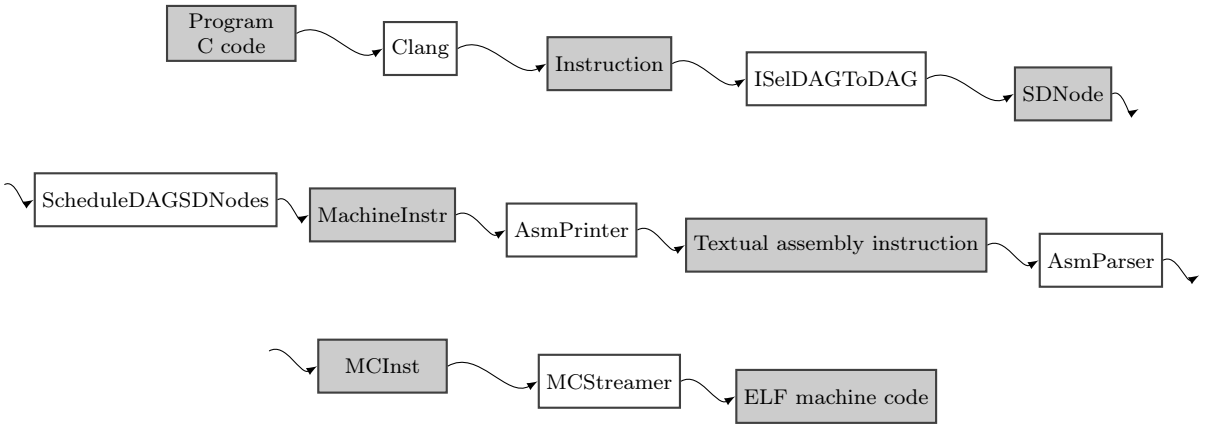


Figure 5.1: The life cycle of the minimal program particle, an instruction, traversing the LLVM backend

lines 7-16 defines a function "main" (stored in memory in the Function class) with a single basic block "entry" (line 9, stored in memory in the BasicBlock class). "alloca" (line 10) and "call" (line 11) are examples of LLVM instructions, stored in memory as subclasses of the Instruction class. Each instruction has a name and other instructions that use the value (Value class) generated by this instructions refer to it by its name. For example, line 12 stores the result of the value "call" calculated on line 11. Names uniquely identify a computation in LLVM because the IR obeys the SSA form. Line 18-19 shows examples of external function declarations, indicating that the final executable needs the help of a linker to include missing code.

```
1  ; ModuleID = 'fib.c'
2  target datalayout = "e-p:32:32:32-i1:8:8-...
3  target triple = "i386-pc-linux-gnu"
4
5  @.str = private unnamed_addr constant [21 x i8] c"myfunction(44) = %d\0A
       \00", align 1
6
7  ; Function Attrs: nounwind
8  define i32 @main() {
9      entry:
10     %d = alloca i32, align 4
11     %call = call i32 @myfunction(i32 44)
12     store i32 %call, i32* %d, align 4
13     %0 = load i32* %d, align 4
14     %call1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([14 x
           i8]* @.str, i32 0, i32 0), i32 %0)
15     ret i32 0
16 }
17
18 declare i32 @printf(i8*, ...)
19 declare i32 @myfunction(i32)
```

Figure 5.2: Example LLVM IR code that calls an external function and print its results

When there is the need to convert LLVM IR to a lower level, machine dialect, a backend uses the common library CodeGen to help in this process. The basic blocks

that originally contains Instruction instances then changes to use the SelectionDAG and SDNode classes to represent instructions in a directed acyclic graph (DAG) form, suitable for pattern matching and scheduling. SDNode instances that represent LLVM instructions are gradually converted to SDNode instances that represent machine instructions. Afterwards, the scheduler determines the order of instructions and converts the DAG to a quadruple format, and the register allocator assigns registers. The output of the scheduler algorithm changes again the program representation from SelectionDAG and SDNode to MachineFunction, MachineBasicBlock and MachineInstr. The backend implements a subclass of AsmPrinter that reads instances of MachineInst and writes the equivalent assembly language code with translation results.

At this point, the compilation process finishes with either an external assembler and linker, used in the past, or with LLVM's own assembler and linker, a recent addition to the LLVM project. We chose to create an LLVM-based assembler, but use an external GNU binutils linker. The linker is, again, a MIPS linker modified to fit OpenISA's own relocation needs. To represent program at this stage, LLVM uses a fourth program representation: the MachineCode library and the MCModule, MCFunction, MCBasicBlock and MCInst classes. The backend must implement a subclass of AsmParser that parses the program in assembly language textual form and converts the instructions to MCInst instances. Then, a subclass of MCStreamer converts MCInst instances to an object file format. We use the executable and linkable format (ELF) used in Linux. Figure 5.1 summarizes the life cycle of an instruction throughout the compilation process, showing in dark gray boxes the name of the data structures used to hold this instruction in memory and in white boxes the main classes responsible for converting from one format to another.

5.3 OpenISA ArchC-based simulator

ArchC is an open-source project developed at IC-UNICAMP that allows the user to easily generate instruction-set simulators (ISS) based on architecture description files.

The ArchC architecture description files can also be used as a readable documentation of the ISA because it contains formats and instructions specified in a clear and concise way. For the purposes of this thesis, an OpenISA ArchC model was created[2] as both documentation for the OpenISA encoding as well as an easy way to generate a working OpenISA (interpreted) virtual machine. This ISS can be used as a reference model for the implementation of other virtual machines and the experimental apparatus of this chapter.

We developed an ISA reference generator that reads an ArchC model and outputs the documentation of the ISA in order to generate the documentation of OpenISA, which is available for reference in the first Appendix of this thesis. The idea is to maintain the ArchC model as the central repository of truth about the OpenISA specification.

[2]Available at `https://github.com/rafaelauler/mips/tree/openisa`

5.4 OpenISA static binary translator

The OpenISA Static Binary Translator (SBT) was designed as a subclass of MCInstPrinter. MCInstPrinter is an LLVM class that reads MCInsts, instructions in the lowest level in-memory representation of LLVM, and prints them to an assembly-language file. The OpenISA SBT also uses a disassembler class that reads the OpenISA ELF file contents and creates the appropriate MCInst instances to represent program instructions, and then feeds them to the MCInstPrinter specialization. The MCInstPrinter traverses all MCInst instances, but instead of printing them to the standard output, it is modified to create LLVM IR fragments to represent equivalent LLVM code. Therefore, this component is a one-pass job. To translate from OpenISA to LLVM IR in one pass, the OIInstTranslate class stores references to an LLVM module, function and basic block, filling each basic block with the translated instructions. It is also responsible for creating additional basic blocks or functions upon reaching a control flow change instruction.

```
@.str = private unnamed_addr constant [13 x i8] c"Hello world!\00",
    align 1

; Function Attrs: nounwind
define i32 @main() #0 {
    entry:
      %call = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
          ([13 x i8]* @.str, i32 0, i32 0))
        ret i32 0
}
```

(a) LLVM IR

```

main:
  addi    $sp, $sp, -24
  stw     $ra, 20($sp)
  ldi     $4, %lo($.str)
  ldihi   %hi($.str)
  call    printf, 1
  ldw     $ra, 20($sp)
  add     $2, $zero, 0
  addiu   $sp, $sp, 24
  jumpr   $ra
$tmp0:
  .size   main, ($tmp0)-main
  .type   $.str,@object
  .section  .rodata.str1.1,"aMS"
$.str:
  .asciz  "Hello world!"
  .size   $.str, 13
```

(b) OpenISA equivalent assembly

```
Address: Encoding

 0:   5d 87 fe 3b
 4:   5f 47 01 2c
 8:   04 00 40 7c
 c:   00 00 04 80
10:   00 00 10 04
14:   5f 47 01 18
18:   02 00 00 38
1c:   5d 87 01 38
20:   1f 00 00 8c



9c:   48 65 6c 6c 6f 20 77 6f
a4:   72 6c 64 21 00
```

(c) OpenISA equivalent encoding

Figure 5.3: "Hello, World!" example in OpenISA part 1

```
; ModuleID = 'simple-oi.bc'
@ShadowMemory = global [4832 x i8] c"\F0\E3QBh\F8\0F\0A\00\00\00...
@reg1 = global i32 0
@reg2 = global i32 0
@reg3 = global i32 0
(...)
define void @main(i32, i32) {
  (...)
loopexit:                                         ; preds = %loopbody
  %11 = load i32* @reg29
  %12 = add i32 %11, -24
  store i32 %12, i32* @reg29
  %13 = load i32* @reg31
  %14 = add i32 %11, -4
  %15 = getelementptr [4832 x i8]* @ShadowMemory, i32 0, i32 %14
  %16 = bitcast i8* %15 to i32*
  store i32 %13, i32* %16
  %17 = load i32* @reg30
  %18 = add i32 %11, -8
  %19 = getelementptr [4832 x i8]* @ShadowMemory, i32 0, i32 %18
  %20 = bitcast i8* %19 to i32*
  store i32 %17, i32* %20
  store i32 %12, i32* @reg30
  store i32 0, i32* @reg1
  store i32 156, i32* @reg4
  %21 = load i32* @reg5
  %22 = load i32* @reg6
  %23 = load i32* @reg7
  %24 = call i32 (i32, ...)* @printf(i32 ptrtoint (i8* getelementptr
      inbounds ([4832 x i8]* @ShadowMemory, i32 0, i32 156) to i32), i32
      %21, i32 %22, i32 %23)
  store i32 0, i32* @reg2
  %25 = load i32* @reg30
  %26 = add i32 %25, 16
  %27 = getelementptr [4832 x i8]* @ShadowMemory, i32 0, i32 %26
  %28 = bitcast i8* %27 to i32*
  %29 = load i32* %28
  store i32 %29, i32* @reg30
  %30 = add i32 %25, 20
  %31 = getelementptr [4832 x i8]* @ShadowMemory, i32 0, i32 %30
  %32 = bitcast i8* %31 to i32*
  %33 = load i32* %32
  store i32 %33, i32* @reg31
  %34 = add i32 %25, 24
  store i32 %34, i32* @reg29
  ret void
}

declare i32 @printf(i32, ...)
```

Figure 5.4: "Hello, World!" example in OpenISA part 2: OpenISA to LLVM IR translation

```
main:                          # @main
   (...)
# BB#2:                        # %
   loopexit
  movl  reg29, %eax
  leal  -24(%eax), %ecx
  movl  %ecx, reg29
  movl  reg31, %edx
  movl  %edx, ShadowMemory-4(%eax)
  movl  reg30, %edx
  movl  %edx, ShadowMemory-8(%eax)
  movl  %ecx, reg30
  movl  $0, reg1
  movl  $156, reg4
  movl  reg5, %eax
  movl  reg6, %ecx
  movl  reg7, %edx
  movl  %edx, 12(%esp)
  movl  %ecx, 8(%esp)
  movl  %eax, 4(%esp)
  movl  $ShadowMemory+156, (%esp)
  calll printf
  movl  $0, reg2
  movl  reg30, %eax
  movl  ShadowMemory+16(%eax), %ecx
  movl  %ecx, reg30
  movl  ShadowMemory+20(%eax), %ecx
  movl  %ecx, reg31
  addl  $24, %eax
  movl  %eax, reg29
  addl  $24, %esp
  popl  %esi
  ret
```

(a) OpenISA to x86 translation results

```

    .file   "hello.c"
    .text
    .globl  main
    .align  16, 0x90
    .type   main,@function



main:
# BB#0:
    subl    $12, %esp
    movl    $7, 8(%esp)
    movl    $.L.str, 4(%esp)
    movl    $1, (%esp)
    calll   write
    xorl    %eax, %eax
    addl    $12, %esp
    ret
.Ltmp0:
    .size   main, .Ltmp0-main

    .type   .L.str,@object
    .section    .rodata.str1.1,"aMS"
.L.str:
    .asciz   "hello.\n"
    .size   .L.str, 8


    .section    ".note.GNU-stack"
```

(b) Native x86 compilation results

Figure 5.5: "Hello, World!" example in OpenISA part 3: comparison of the x86 code generated by the OpenISA static binary translator with no optimizations, to the left, and of the x86 code generated by a native x86 compiler, to the right.

OpenISA binaries have a predefined memory layout that we cannot change in order to provide binary compatibility. This memory layout is the position and sizes of data in memory. If the string "Hello, World!", for example, is compiled in the OpenISA binary to occupy the address `0x10000`, right after the last instruction of the `.text` section, then the translated binary will need to layout this data as is, while allowing load/store instructions to access this information at the exact same address. To address the memory issue, we load the binary image in a global byte array named ShadowMemory, declared in the LLVM module. Every load/store instruction is redirected to index ShadowMemory. Each OpenISA register operand, on the other hand, is mapped to a global variable that represents a specific register number.

The "Hello, World!" program is an easy-to-understand example that illustrates the entire framework, departing from LLVM IR to x86 assembly code, being translated through OpenISA. Figure 5.3 shows the first part: the LLVM IR code and the results of the OpenISA backend, which generates OpenISA instructions contained in an ELF binary. The LLVM IR code was generated by clang frontend that reads C code and generates LLVM IR code. The resulting OpenISA assembly contains a few different features from the original MIPS that inspired the first OpenISA version. Our preference for MIPS as the starting point and these additional modifications will be discussed later, allowing us to focus this section on the translation process.

Figure 5.4 shows the OpenISA translation to LLVM IR in practice. Line 2 shows the ShadowMemory global array with 4832 positions (the full initialization was omitted due to space restrictions). ShadowMemory contains the original OpenISA binary image, as well as space for the stack. The translation does not use the host stack, but maintains its own stack in ShadowMemory – this is not only a consequence of not having enough information to rebuild the stack using the host dialect, discussed in Section 3.2.2, but also important to preserve the original stack layout for compatibility.

Lines 3-6 declare the global variables that stores the contents of OpenISA registers. The full initialization of all integer and double registers was omitted due to space restrictions. The first lines of the sole function was also omitted because it is a template code that converts the main argv string array to use ShadowMemory-indexed strings. The translator will use as many functions as the original OpenISA binary has. This region formation is discussed in Section 3.2.1: we extract function boundaries from markers.

Lines 10-44 contains the result of the OpenISA translation to LLVM instructions. The number of instructions increased because each OpenISA instruction was broken down into simpler fragments, which will be matched by the LLVM SelectionDAG when compiling to run the code on another host machine. For example, Figure 5.5 shows the results of using an x86 backend in this code, which yields 29 x86 instructions if the initial template loop is discarded. It also shows the result of a native x86 compilation of the hello world example, which is considerably smaller. This difference suggests that emulating OpenISA programs on x86 would incur large overheads, however, we used a simple translation technique to easily work out an example. We will discuss later how we can enhance the quality of the translator to achieve native-execution performance.

5.5 Runtime library

OpenISA SBT, different from the OpenISA interpreter, identify function calls to the C library and adapts the necessary parameters to perform the call to the host C library, which means the C library is not translated and is subsequently not accounted for in benchmarking. This is for experimental purposes.

The OpenISA interpreter, built with ArchC, on the other hand, executes fully linked binaries. The OpenISA toolchain links user programs against a version of the RedHat newlib that is modified for OpenISA. The OpenISA toolchain will be exposed in the next section.

5.6 OpenISA toolchain

A toolchain is a set of tools that allows a developer to compile programs to a specific host platform. The OpenISA toolchain is built on top of Clang and LLVM, featuring the following tools:

- Compiler based on Clang 3.6 with integrated assembler
- Linker based on Binutils 2.24
- RedHat newlib version 2.1.0 and Linux headers version 3.10.14
- Disassembler based on LLVM 3.6 (llvm-objdump)

To compile a program to assembly, for example, the user can issue the following command:

```
1 $ echo 'int main() { printf("Hello, world!\n"); }' | clang -x c \
2     -S -o test.s - && cat test.s
```

In order to create an ELF relocatable object, the following command accomplishes the task:

```
1 $ echo 'int main() { printf("Hello, world!\n"); }' | clang -x c \
2     -S -o test.s - && cat test.s
```

If one wishes to use a disassembler to read the recently created object, she can issue the following command:

```
1 $ llvm-objdump -disassemble test.o
```

Finally, in the simplest case where the user wants to fully compile the C source code to a final ELF executable, the command would be the following:

```
1 $ clang input.c -o output
```

The compiler *driver* is a crucial component of a toolchain because it is responsible for locating all the tools, header files and library files of the target platform and building the command line (or command lines) to serve the task the user requested via command-line interface.

In order to customize a new toolchain, OpenISA implements a new instance of the Clang classes `Toolchain` and `Tool`. The OpenISA Clang implementation is programmed to find newlib and the Binutils linker in a relative path and to launch the compiler, the integrated assembler, which is an LLVM-based assembler that is built on top of MC classes, and finally the linker with appropriate libraries.

5.7 OpenISA evaluation workflow

Program	Description
ackermann	Calculates the Ackermann function [1] (recursive)
array	Repeatedly accesses and updates array elements
fibonacci	Calculates the Fibnoacci function (recursive)
heapsort	Sorts several random values (deterministic)
lists	Performs several operations on a doubly linked list
matrix	Calculates the result of a matrix multiplication
random	Repeatedly computes a modular equation typical in random value calculation
basicmath	Performs mathematical calculations, e.g., cubic function solving, integer square root and angle conversions from degrees to radians
susan	Recognizes corners and edges in MRI brain scans
dijkstra	Graph shortest path calculation
patricia	Exercises the Patricia trie [85] data structure for sparse trees
rijndael	A block cipher algorithm chosen as the Advanced Encryption Standard(AES)
fft	Computes the fast fourier transform and its inverse transform
adpcm	Computes the Adapative Differential Pulse Code Modulation
crc	Performs a 32-bit cyclic redundancy check on a file
stringsearch	Searches for words in a text file
sha	Computes the SHA 160-bit digest of an input
blowfish	Ciphers a block with a symmetric key of variable size
lame	Encodes a WAV input file into an MP3 file
bitcount	Counts number of bits with different algorithms
jpeg	Encodes or decodes between the uncompressed PGM file and JPEG
401.bzip2	Compression and decompression algorithm
429.mcf	Combinatorial optimization / Single-depot vehicle scheduling
433.milc	Physics / Quantum Chromodynamics (QCD)
458.sjeng	Artifical intelligence (chess playing)
462.libquantum	Physics / Quantum computing
464.h264ref	Video compression
470.lbm	Computational Fluid Dynamics, Lattice Boltzmann Method
482.sphinx3	Speech recognition

Table 5.1: Description of the selected benchmark programs

This section presents the evaluation methodology that will be used throughout this thesis to assess whether OpenISA is close to the goal of being easily emulated on other hosts or not. It is central to the development of the emulation friendliness argument and also has an important role by shaping the design of OpenISA. There is *no set of benchmark programs* that can ever represent all the programs used in a general purpose computer – we make a best effort of selecting a diverse set of programs to test the proposed framework. The theme of the selected programs covers many areas such as math, image processing, graph processing, network-related, signal processing and cryptography – but

it is important to limit the study to a fixed number of programs in order to have a target to focus on.

This methodology relies on three sets of benchmarks to assess OpenISA translation capabilities and performance. The first set has programs from the shootout [12] benchmark, which are small and simple programs that perform a repetitive computation. The second set has programs from the Mibench benchmark. We excluded from this analysis Mibench programs that failed to compile (mad, tiff, ghostscript, ispell and rsynth), that uses signal handlers (GSM and PGP), that spends most of its time in a libc function (qsort) and that triggers a bug in the LLVM infrastructure during translation. The last set are programs from SPEC CPU2006 [70] written in C, excluding 400.perlbench and 403.gcc, which are too large to fit in our evaluation flow[3]. Table 5.1 provides a brief description of each benchmark program.

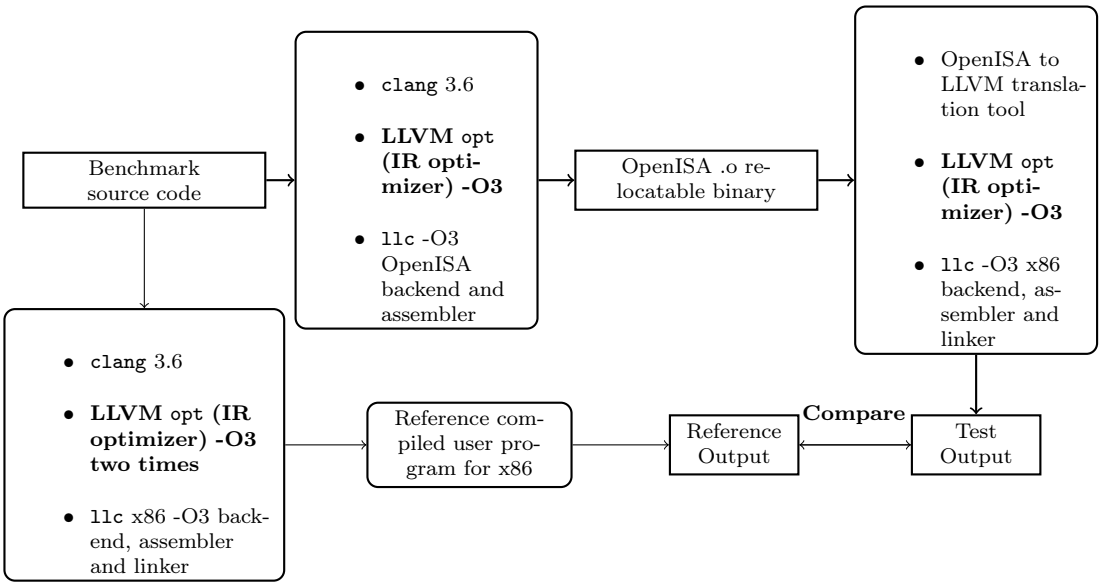


Figure 5.6: Experimental workflow used to test the performance of the ISA translation

The experimental workflow is depicted in Figure 5.6. The first box shows the benchmark source code, which can be a SPEC CPU2006, Mibench or shootout application. Afterwards, the `clang` frontend is used to convert the C source code to LLVM IR and optimize it with `-O3` (second box in the first row), the most complete LLVM target-independent optimization option. Then the OpenISA backend and assembler kicks in to translate the benchmark to an OpenISA relocatable file, which can be considered as the final executable file for the purposes of this experiment that was not built to evaluate the C library.

The next steps, in the 4th box of the first row, are responsible for running the actual static translation tool, the OpenISA to LLVM component, which produces an unoptimized LLVM IR, then the `opt` target-independent optimizer with `-O3` again, and finally

[3]We use LTO (link-time optimization) in our evaluation flow to produce the most optimized version of each benchmark program, but together with the static translator, the flow for 400.perlbench and 403.gcc takes several hours to complete. In the interest of time to build, debug and run the experiments several times, we removed them from our evaluation.

the translation to x86 by feeding it to the LLVM x86 backend, assembler and linker. Figures 5.3, 5.4 and 5.5 previously showed input examples to each major component of this framework.

In a separate compilation flow (second row in Figure 5.6), it is built the native program that will generate the correct reference output and allow the experimenter to measure the native performance, i.e., x86 without translation. The same LLVM compiler version must be used to avoid introducing a different optimization pipeline. Notice that it is used `opt -O3` once to optimize the clang output and another time to optimize the OpenISA to LLVM translation. If the experimenter optimizes the native counterpart only once, some OpenISA-translated programs will get faster than native in an unfair result because -O3 is used twice in the OpenISA workflow. Therefore, to reproduce the exact target-independent optimization pipeline that is used in the OpenISA translation workflow, it is important to use `LLVM opt -O3` two times in the native workflow as well. The remaining boxes illustrate the output validation activity, which confirms that the OpenISA-translated binary did not change the program behavior.

When assessing these benchmarks for translation performance, this workflow assumes that the developer already applied target-independent optimizations if she is interested in maximum performance. Therefore, when generating the OpenISA binary, the experimenter must compile it with -O3 to garner all optimization potential available at the C language level. OpenISA, as a low-level ISA, cannot aim at recovering all high-level optimizations that are available at the language level and depends on the OpenISA binaries being generated with optimizations enabled to avoid missing optimization opportunities, although it *is* able to recover and apply many optimizations. For example, the matrix benchmark gets two times faster if the user applies a second round of -O3 optimizations, and even if this second round is applied in the LLVM IR after the OpenISA to LLVM translation pass, the optimization is successful, showing that OpenISA preserves enough information to enable some target-independent optimizations.

5.8 Summary

This chapter presented the design of the experimental apparatus used to validate OpenISA's emulation friendliness aspects. There are other tools that were also crafted to create the OpenISA ecosystem. This involves a complete toolchain featuring a C compiler, C library, assembler, linker, disassembler and simulator.

The bulk of our experiments, however, relies on the static translator. Even though OpenISA is geared towards emulation in a DBT system, we developed a static translation framework to focus on code quality. However, as an static translator, this framework can be used to study another scenario: distributing binaries in OpenISA to be translated to the host ISA at install-time. This will be further explored in the next chapter.

Chapter 6

OpenISA design for easy emulation

In this chapter, we first present a simple discussion about why OpenISA started with MIPS and then we adopt an empirical approach to enhance the initial, MIPS-based OpenISA, allowing it to be easier to translate to other architectures. We also assess the performance of the static translation apparatus to show an upper bound of the OpenISA translation performance, in which there are no time constraints to apply optimizations to enhance the translation quality. The key idea in our experiments are based on the observation that if OpenISA do lose performance-critical information, the static translation and associated compiler would not be able to achieve the original program performance. We carefully show each corner case and discuss how to remove limitations by modifying the ISA design.

6.1 ISAs with faster emulation

An important tool to build one of the main arguments of this thesis is the ability to answer whether an ISA is fast or *easy* to be emulated. This *emulation* may refer to different techniques, for example, interpretation or binary translation. Since the latter provides us the best performance, we focus this thesis on showing good binary translation performance. However, this section discusses how we can provide evidence of which ISA has the best performing interpretation-based emulation, although it is not possible to formally prove it. This is used to show why OpenISA was initially based on MIPS.

Being the most straightforward approach to emulation, interpretation exposes an inherent aspect that is fundamental to any kind of translation: the semantics of each individual instruction. Regardless of the level and quality of the translation, the system must know how each target instruction changes the emulated processor state. The target-dependent component of the translation system is entirely dependent on which ISA is being emulated, which prompts the question "can an ISA make a difference in performance and be easier to emulate?"

A simple answer to this question is thoroughly discussed in the Smith and Nair Virtual Machines book [124] and states that a host machine with more registers than the target machine, as well as with hardware support for most of its mechanisms, will benefit with higher performance. The extreme case is the same-ISA emulation, where a program compiled for the same platform of the host is emulated to isolate it from the host oper-

ating system. Since the hardware directly supports the behavior of the instructions, the emulation speed will be high. On the other hand, we are not interested in finding an ISA that is easy to be translated to a specific host, but an ISA that is easier to be emulated on most of the popular architectures.

The value of ArchC in this research, therefore, is its clean separation of concerns and ability to model different ISAs, allowing us to focus on the ISA aspect of a simulator instead of specific translation techniques. In this sense, ArchC is particularly useful to craft an experiment that tackles this question. It is as simple as running ArchC simulators for different ISAs, compile the same program to different ISAs, run the program on these ArchC simulators and the fastest execution indicates the ISA that is the easiest to emulate among those tested.

However, the answer this experiment provides is not perfect because of the implementation quality of each ArchC model, which may vary, and the quality of each compiler. In ArchC, the designer must write C code to indicate the change of state that a single instruction in the ISA performs. This C code may be suboptimal. More importantly, the compiler toolchain may miss optimizations that makes the input program much better represented by a given ISA.

In theory, the problem of generating, for a given input program, the shortest equivalent program in a given ISA, that is, to use the *best compiler* for a given program and ISA, is *undecidable* – if the input program is an instance of the halting problem, for example, the compiler would need to solve the halting problem to generate a single-instruction program that reports whether the program halts or not.

In practice, it is not reasonable to expect that the *best compiler* exists and there will always be a missed optimization. Therefore, we do not attempt to answer with formal rigor which ISA is the most suitable for emulation. This experiment works with possibly faulty tools, but to overcome this, we build our argument on a qualitative examination of the ISA, not only based on the result of experiments.

First, to understand how an ISA may be a bad guest option for emulation, consider the ARMv7 ISA, the latest release of the 32-bit version of the ISA that dominated the mobile devices market. The first 4 bits of every ARM instruction but a few special ones indicate a predicate that must be evaluated together with the condition register to determine whether the instruction will be executed. While *predicated execution* is a valuable tool for VLIW [51,69] or GPU [101] architectures, it has a much smaller impact on general purpose CPUs, and this feature was removed from the 64-bit remake of the ARM architecture. Regardless of the usefulness of some features, the ISA made a remarkable success and, with it, came an entire ecosystem of 32-bit ARM applications.

When emulating predicated execution in a machine that lacks this support (as most do), the resulting code will have instructions that potentially change the control flow for every emulated guest instruction. Figure 6.1 shows an example code extracted from the ARM ArchC model, intended to run ARM instructions on any platform. The *switch* statement is implemented with an indirect jump, while the *if* statement as a conditional jump in the target architecture.

This expensive mechanism hints that some ISA are, in general, harder to emulate, i.e., inflict a larger performance penalty than others. ArchC simulators are simple interpreted

```
execute = false;
switch(cond) {
    case 0:  if (flags.Z) execute = true; break;
    case 1:  if (!flags.Z) execute = false; break;
    case 2:  if (flags.C) execute = true; break;
    // ... more 13 cases
}
```

Figure 6.1: Pseudo-code that determines whether a single ARM instruction should proceed to the execution stage

	ARM (s)	**Speed**	**MIPS** (s)	**Speed**	**SPARC** (s)	**Speed**	**PowerPC** (s)	**Speed**
basicmath	300.79	5386	**76.17**	17871	100.91	12933	111.46	13382
bitcount	6.77	6335	**2.5**	18237	3.57	13967	3.46	15128
qsort	2.67	5633	**0.83**	17364	0.97	14575	1.07	13945
susan	4.77	5908	2.22	15910	1.99	15117	**1.57**	17800
adpcm	4.92	5223	-	-	-	-	-	-
crc32	72.38	5915	**36.03**	17070	38.71	15182	37.70	14915
fft	168.23	5464	**48.89**	15556	49.86	14355	51.86	15867
gsm	4.39	6012	1.98	16496	**1.63**	14384	1.67	13916
dijkstra	9.17	5915	**3.21**	18510	3.86	13193	3.59	14161
patricia	59.08	5201	**16.53**	17495	21.98	12651	25.81	12268
rijndael	-	-	**2.21**	15255	2.33	13974	2.61	11336
sha	2.12	6619	0.85	15336	0.91	14565	**0.77**	15637
jpeg	4.33	5656	1.73	17037	1.63	15253	**1.44**	16695
lame	1870.42	5336	**448.03**	17931	528.52	14458	532.26	16293

The speed metric used is thousands of instructions per second (KIPS)

Table 6.1: Mibench results for different ISAs simulations with ArchC

machines, whose performance depend almost exclusively on the C code implementation of each alien instruction. Based on this observation, we conducted a simple experiment to give an overview of the complexity of each ISA, similar to what was discussed.

The ISA comparison experiment involves running 4 different ArchC simulators, each one emulating a different ISA, with Mibench programs [66]. Table 6.1 presents the total time required to run each program on each platform. Total time is a valuable metric that undeniably reflects the translation quality of each program from the guest architecture to the host. For example, the first line of Table 6.1 shows that the basicmath program was executed on 4 different ISAs by means of ArchC simulators, and took 300.79s to run using the interpreted ARM simulator, 76.18s for MIPS, 100.91s for SPARC and 111.46s for PowerPC.

This gives evidence that the MIPS ISA may offer the easiest translation: it presents the lowest run time to complete 9 out of 13 Mibench benchmarks. However, the total time takes into account other effects that we do not want to measure, e.g., the quality of the compiler used to translate from C code to the target machine. Nevertheless, the speed metric confirms that the MIPS simulator is running faster: it runs 10 out of 13 Mibench benchmarks with the highest number of thousands of instructions per second among the simulators.

	Altera Nios II			ARM 32			Imagination MIPS32		
Benchmark	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS
linpack	3,075,857,171	1.8s	1718	6,105,766,856	4.32s	1413	9,814,621,392	4.83s	2032
Dhrystone	1,810,082,387	1.08s	1676	2,250,079,359	2.08s	1083	1,795,088,667	1.05s	1710
Whetstone	5,850,887,389	2.67s	2200	1,185,959,501	0.96s	1238	1,890,420,892	0.8s	2368
peakSpeed2	22,000,013,458	3s	7335	22,400,008,766	4.6s	4872	22,800,009,853	3.07s	7427
	Xilinx MicroBlaze			ARM AARCH64			Imagination MIPS64		
Benchmark	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS
linpack	6,386,275,159	3.2s	2002	2,403,904,724	3.7s	650 †	1,558,856,686	0.75s	2079
Dhrystone	3,770,115,740	2.42s	1564	11,510,061,362	11.36s	1013	1,590,094,345	1.05s	1516
Whetstone	27,108,532,655	11.49s	2359	2,623,931,374	3.01s	872 †	2,133,926,552	0.88s	2453
peakSpeed2	22,000,023,433	4.38s	5034	44,800,003,885	6.7s	6687	17,100,018,075	3.26s	5249
	PowerPC			Renesas v850			Synopsis ARC		
Benchmark	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS
linpack	3,163,966,113	2.24s	1419	4,991,344,159	3.66s	1368	4,184,162,664	3.15s	1328
Dhrystone	2,205,068,239	1.53s	1441	6,410,133,101	3.63s	1766	3,155,082,476	2.23s	1412
Whetstone	6,424,865,755	3.34s	1929	10,296,940,591	6.16s	1674	7,883,567,047	3.77s	2091
peakSpeed2	22,400,002,937	4.37s	5126	22,400,007,569	3.29s	6809	22,000,002,100	3.83s	5744
All measurements on 3.50GHz Intel i7-4770K, Linux FC20, OVPsim 20140731.0									
† Hardware Floating Point Instructions									

Table 6.2: Table extracted from the OVP website [89] reporting their experiments on simulation speed of their own tool – *this is not original material from this thesis.*

While the performance of ArchC interpreted simulators are prone to other effects that do not reflect easiness of the ISA translation, these preliminary results indicate that MIPS is a reasonable candidate owing to the following characteristics:

- MIPS does not have a program status register, which is a hardware aspect that is hard to emulate on architectures with incompatible behavior [124];
- MIPS instructions do not have complex behaviors as the ARM conditional execution – the instructions are straightforward to implement;
- MIPS has fewer instructions, which leads to less emulation code, light cache requirements and higher probability of being easily supported by any other architecture owing to its simplicity.

It is also possible to garner evidence from other experimenters that MIPS indeed offers a fast emulation. One of the best performing and commercially available simulators is OVP (Open Virtual Platforms) [89], which, despite its name, is closed-source binary translation system and does not reveal which optimizations it uses. Table 6.2 is reproduced verbatim from the OVP website and reports the simulation speed and simulation run time for a variety of architectures – similar to the experiment presented here, but now for a DBT system instead of an interpretation-based simulator.

Upon a detailed examination of this table, both MIPS32 and MIPS64 appear as featuring the fastest simulation speed of all tested architectures, and they are also among the fastest overall run time for most of the benchmarks. One exception is for linpack, where MIPS32 performs badly with 4.83s but MIPS64 has the best run time with 0.75s, which suggests this benchmark depends heavily on 64-bit arithmetic support. The other exception is for peakSpeed2, in which Altera Nios II completes in 3s, being the fastest simulation for this benchmark, while MIPS32 completes in 3.07s, slightly worse than Altera Nios II, but still close.

This supports the conclusion that MIPS is among the fastest ISAs to be emulated on x86, which is the common host used in all experiments. Later, when testing OpenISA,

we will also present results for ARM as a host and provide a close examination of it in Chapter 7.

6.2 OpenISA design choices in favor of emulation

The establishment of the experimental framework enabled a thorough examination of the performance implications of each OpenISA different design option. The design options were not randomly chosen and tested, but instead were the result of the analysis of the root cause of poor performance on some benchmarks. About 8 of 17 programs, however, presented good translation performance on x86 since the beginning of this iterative process by using the bare MIPS ISA along with our choice of metadata information. This will be further discussed in Chapter 7.

An important insight of this thesis is that the crucial modifications to allow easier translation are all focused on how an ISA access data rather than how to process them. The experiments revealed that, for example, encoding conditional control flows to use the limited MIPS branching scheme does not impose difficulties when translating to a different ISA, but significant overheads arise if it is not trivial to understand how data is used in the program. This is a consequence of data aliasing problems that occur when converting a program in binary form back to a high level representation.

6.2.1 Primary design choices

The primary design choices are made as the result of direct investigation of the ISA necessities in order to facilitate emulation. Those are discussed next. The secondary design choices are compromise decisions in order to make it possible for the primary design choices to be implemented and are discussed afterwards.

Expansion of the register bank

The classic MIPS ISA already features plenty of integer registers: 32 of them, exceeding both ARM and x86. This is enough to encode in registers most intermediate values that are being calculated during most hot loop bodies. However, if there is even a small number of registers that were spilled to memory in a hot loop body in OpenISA code, there is significant impact on performance.

The concept of stack frame gives unlimited local memory capacity to hold any intermediate values that do not fit in the register bank at a given point during program execution. Similarly, the LLVM IR allows an infinite number of local variables to live in the context of a function call. The LLVM IR easily maps to the hardware because when the limited register bank is fully allocated, exceeding variables are spilled to the program stack, which resides in memory. After spilling, information about local variables has been lost and it is not possible to do the reverse operation, promoting memory operations back to registers after the binary has been generated.

Promoting memory operations to registers may break data layout and binary compatibility, which in turn may alter program behavior in significant ways, such as masking

memory bugs and creating coherence problems between the cached value in a register and the value in the memory position. Such problems may arise if the translator engine fails to correctly identify when a memory address has been saved and will be used to load a value and also in multi-threaded applications that may wish to intentionally share this value across different threads of execution.

Similarly, if a value has been spilled to memory, it precludes many dataflow optimizations from propagating information through this variable, which has the effect of limiting the scope of optimizations. It will also be difficult for this value to be promoted to live in a host register.

The best option at the ISA level to make the translation easier in face of the register problem is to expand the register bank, allowing the compiler to keep all local values on registers when compiling for the guest ISA. The experiments made in the scope of this thesis suggest that 64 registers make the cases of spilling rare and restricted to heavily unrolled loops that are aggressively seeking instruction level parallelism.

In 2011, Alipour et al. [19] concluded that the optimal register bank size for the Out-of-Order backend in Mibench and PacketBench [112] benchmarks is 80, which shows that after dynamic scheduling, 32 registers is certainly suboptimal to represent dataflow dependencies. For OpenISA, since this is the design of the user-visible register bank with a high-impact on the encoding (every instruction must address) this register bank, it is limited to 64 registers.

Design choice 1: To expand the integer register bank to 64 registers.

Accessing immediates

To understand why accessing immediates ought to be different in OpenISA, we will start with a practical example. In Mibench, dijkstra is the benchmark that tests the famous algorithm of the same name to calculate the shortest path from a given starting node in an input graph. The dijkstra function, where the dijkstra benchmark spends most of its time, has three nested loops. In the OpenISA translated version, the innermost loop unnecessarily loads values from memory even though they are known to be zero. Since it is the innermost loop, any unnecessary computation is high-impact. This load immediate instruction could be replaced by the constant zero and be eliminated – it corresponds to the high part of a 32-bit immediate that was split into the originally MIPS instructions `LUI` and `ORI`, a common MIPS combo used to load 32-bit immediates. It is expected that 32-bit immediates are split into upper and lower part to be encoded into 32-bit MIPS instructions, and this strategy is employed to load addresses of symbols as well, which are just another class of possible constants in a program. However, symbols are a special type of constant because it is only known at link time – the compiler does not know whether the address will fit into the immediate field of a single instruction and conservatively emits separate instructions to load the upper and lower parts. At link time, no optimizations can remove instructions and re-do function layout[1]. In order to

[1] The so-called "link-time optimizations" of Clang and GCC are a misnomer and are unable to perform this kind of optimization. There are, indeed, programs that are able to read a binary, perform microarchitectural optimizations such as improving cache utilization and rewrite the binary. However, they are

avoid the potentially harmful confusion involved in recovering immediates from OpenISA machine code, OpenISA adopts the following design decision:

> **Design choice 2**: To always encode the full literal in a single instruction without splitting it.

In dijkstra or in programs that abuse global variables, the upper part of the address is often zero and is responsible for making the OpenISA version of dijkstra to be 60% slower than native in LLVM 3.3, although the problem is mitigated in LLVM 3.6. Since this depends on the compiler competence, OpenISA encodes this in the ISA to avoid relying on expensive compiler algorithms to recover this information.

The key insight of this design choice comes from observing that compiler optimizations such as loop invariant code motion (LICM) can move instructions far apart from each other, preventing them from being easily fused back and hampering the performance for programs that abuse globals whose 32-bits address need to be encoded on constants larger than 16 bits. Even though it is not impossible to recover this information, the experiments revealed it to be surprisingly hard to fuse these back to a full immediate once a handful of optimizations on the OpenISA machine code move these values around separately. Therefore, this design decision is a straightforward solution to improve recognition of immediates.

The function call interface

A classical problem in compilers is to conciliate the data allocations happening at different regions processed as separate inputs to the register allocator. For example, consider a programmer wants to design a register allocation algorithm that operates on a basic block at a time. If the algorithm chooses to allocate variable *foo* into register `R1` in a basic block B_1 and to allocate *foo* into register `R2` in another basic block B_2 that may succeed B_1 in any given path, the compiler will need to insert compensation code somewhere between B_1 and B_2 moving `R1` to `R2`. This move instruction could be removed by a register allocation algorithm working with a broader view, with knowledge about data-flow dependencies across the entire function and cleverly allocating *foo* to a common register shared by both B_1 and B_2.

The region at which compilers typically operate is a function, which means that, for register allocation purposes, no registers are shared between different functions, except those explicitly stated in the ABI. The ABI specifies parameter passing registers and return value registers acting as a "data transfer zone" between caller and callee. The practical approach is, in presence of two functions that are closely intertwined, to inline callee into caller, allowing the compiler to emit better code by removing the function call interface at the cost of code duplication.

In the case of a translation engine that uses a full-blown compiler backend to emit code, such as the static binary translator studied in this thesis, the region at which the translator operates also has a high impact on translation quality. The translator, which is at the core of a virtual machine, needs to completely decode data dependency relations

special purpose and not part of regular compilers.

among instructions (its *data-flow graph*) and to allocate intermediate values into host registers. If there are few spills, it is possible to perform a decent job of pairing virtual to host registers.

The problem happens in a similar way to the analogy of the basic blocks register allocation: in the function call interface, when caller calls the callee, there is a mapping clash between two different translation regions. This will always happen if the translator operates on a function at a time, which is the case for the OpenISA static binary translator. Since different functions may have allocated different guest registers on different host registers, to synchronize both functions the translator emits an expensive sequence of instructions to push to memory a potentially large set of registers, and then, at the callee, pull all those saved registers from memory to the correct host registers according to the allocation performed at the callee.

When calling a function, the translator may proceed in two ways, depending on the compatibility mode. (1) To provide utmost compatibility, the translator will push *all registers* touched by the caller to the memory, providing to the callee the complete current state of the register bank. However, if the callee obeys the ABI rules, it is illegal to depend on the value of a register other than the specified by the ABI as register-passing. In those cases, it is unnecessary overhead to synchronize the full register bank upon calling a function. If the translator has no way to know that the callee is ABI-compliant, it needs to perform this expensive approach. (2) To increase performance, however, if the translator knows the callee will never depend on the value of a register that is not an ABI parameter-passing register, it can speculatively avoid updating such register.

In the case of approach (2), it is important to *always* know beforehand how many parameters need to be transmitted to the other region, or function. An interprocedural analysis is able to discover which registers are effectively used by the other function, helping us to find the correct number of parameters, however, we do not always know all functions at compile or translation time. Parameters often live in register, and registers are shared, in time, across several computations. The problem of conservatively assuming that a nonexistent parameter is valid is the same of assuming that a dead register is alive, and affects the performance of the translated code. For example, in dijkstra it is assumed that a call to the `printf` function would always use 4 parameters, in registers, thus simplifying the interface with the underlying libc. However, at the end of the loop computation, the program calls `printf` with 3 parameters. The nonexistent fourth parameter is listed as used, which lives in a register whose value happens to be calculated inside a hot loop. In this way, in the native compilation, this calculation is discarded as soon as the compiler infers that it is not used anymore – for instance, it is not used after the loop. In the OpenISA translation, the compiler can no longer discard this value because there is not enough information to realize that the value in the register is not used as a parameter, forcing the hot loop to calculate the value for this register in all paths because its use in printf dominates the loop and significantly affecting performance.

> **Design choice 3**: Change the call instruction to include one extra operand, the number of parameters passed to the callee.

This design choice allows the translator to infer important liveness information and

prevent unused values to be stored on memory at translation boundaries, allowing for higher precision translations.

In the next sections, this thesis will also explore the case where all functions in a program are translated into the same translation region. It is an expensive approach that may require a large amount of time to translate the program, but works for all tested programs and serves the purpose of measuring the overheads imposed by the register synchronization problem tackled by this design choice.

Enforcing function granularity

This design choice is necessary to allow the translation engine to easily identify the region of functions.

> **Design choice 4**: Include *begin function* and *end function* marker instructions that delimit the function region and allows the translator to employ *method-based compilation*.

Double and float storage

In order to implement the original MIPS register bank, it is necessary to allocate space for 32 integers and 32 floating-point numbers. These 32 floating-point numbers share the same space with 16 aliased doubles, which use, each one, 2 floating-point registers. When translating code that uses doubles, this register configuration requires casts to store the lower and the upper part of the double as two different 32-bit floating-point numbers.

A problem arises because aliased registers creates uncertainty about the type stored at a given memory location. Aliased registers imply that the same memory position can hold two different types – the exact one which you will only know when an operation uses the value. Until it is used, you must store it as an unknown type. The unknown type precludes this memory position from being promoted to use host registers, since the host machine often requires the type to be known (whether it is double, float or a variant of integer) to allow efficient use of host registers.

> **Design choice 5**: To segregate double and float register banks.

Therefore, if we assume a fixed type for each register, the translation avoid repetitive casts and provides an improved translation that compensates the lack of a high-level typing system.

Jump table identification

An important aspect of a translator is its ability to accurately recover the original CFG of the program. When programs have indirect jumps, it becomes challenging to recover the block successors with no hints. The translator may assume that all code pointers[2] pointing to addresses in the current function reveal potential successors (or targets of the

[2]See the discussion on metadata in Section 6.2.4

indirect jump). However, this conservative approach may create unnecessary edges in the CFG, hindering optimizations.

Another strategy is to read the operands of an indirect jump to try to figure out the base address of a jump table, read the jump table scanning for valid code addresses and assume all of those are block successors. This may not always be possible if the operands of the indirect jump are not in a canonical form. Even if they are, if the instructions building the final jump address are scheduled far apart, the translator will have a hard time finding those.

> **Design choice 6**: To create the IJMP instruction with the following operands: the jump table base address as an immediate, the index as a register and the table size as an immediate.

To make jump table identification trivial, the IJMP instruction forces the program to follow a canonical form of performing indirect jumps, exposing the jump table base address and index separately. This information can also be useful for security reasons, since indirect jumps may be used to divert control flow to malicious code. If the hardware enforces a size in the jump table, it is harder to tamper with the jump target by changing the index.

6.2.2 Rejected design choice

This subsection discusses a valid design decision that was discarded.

Guest code spills

In a past experiment, a design decision on OpenISA focused on reducing the overhead due to spilled values. A spilled value in OpenISA code always impacts the translation quality because the translated code is bound to use this memory location. If the value lives in registers, the translator can run dataflow optimizations on them and take allocation decisions that is on par with the original native compilation. If the value is spilled to memory, the translator can no longer differentiate between a local and a global value. Since a memory position can be externally updated, this value will never be promoted to a register and dataflow analyses will conservatively assume that this position can hold any value.

If this memory access was generated by a spill of a local variable that did not fit into the register bank, the backend can annotate it to inform the translator that this is not a global and this memory position can assume the same rules as a regular local variable. This motivated the following design choice:

> **Rejected design choice**: Include *spill load* and *spill store* as annotated load/store instructions that access a non-volatile memory position.

When the translator finds a `spilllw` or `spillsw` instruction, it may decide whether it really should be allocated in memory, i.e., if an instruction uses its address, or if it can be promoted to a register.

This design decision was later removed (rejected) because this problem is better mitigated by the use of an expanded register bank. The semantics of `spilllw` and `spillsw` would be identical to the regular load/store instructions `ldw` and `stw`, except for the implicit metadata stating that this instruction accesses a local value. However, this creates a difficult problem if loads and stores are not paired to use the same address. The metadata should tag memory addresses instead of the load/store operations. Since OpenISA avoids abusing metadata, this solution was discarded in favor of design decision 1.

6.2.3 Secondary design choices

This subsection discusses design choices made as a necessary compromise to allow OpenISA instructions to be encoded in 32 bits as a result of primary design decisions. For example, since OpenISA uses 64 directly addressable registers, the instruction encoding needs to allocate 6 bits to encode each register operand and consequently reduce the bitwidth available to some immediates.

Instruction operands using two registers and one immediate

Two common encodings for data processing instructions in RISC architectures is to use (1) three registers, two sources and one destination, and (2) two registers and one immediate, where one register and one immediate are source operands and the other register stores the operation result (destination).

The size of this immediate is different for each ISA and often reflects the compromise between other design decisions. For example, since ARM uses 4 bits in every data processing instruction to store a conditional code, enabling predicated execution of every instruction, it must shorten its immediate fields. ARM also allows instructions to encode the number of bits the user wants to shift left, shift right or rotate one of the operands because its original microarchitectural design featured a barrel shifter in the datapath [61]. For this reason, the immediate loading mechanism in ARM is quite complicated, involving 8 bits plus a possible shift of these 8 bits to any bit offset. For example, it is possible to encode the immediate `0xFF000000` in ARM, which is equivalent to the 8-bit immediate `FF` shifted left by 24 bits, but it is not possible to encode the immediate `1FF`, which requires 9 bits. The MIPS ISA, on the other hand, has plenty of room and encodes 16-bit immediates together with two other register operands. Sparc uses 13 bits to immediates and the IBM POWER, 16 bits.

Since OpenISA expands the register bank to 64 registers, it makes available 14 bits for such immediates.

Jump and call offset sizes

Since OpenISA encodes the number of parameters passed in a function call, the call instruction loses 6 bits of call offset in relation to MIPS. Therefore, OpenISA is open to call any function in a 22-bit range (encoding uses 20 bits since lower 2 bits are always zeroed). For unconditional jumps, it is able to jump anywhere in a 28-bit range (encoding uses 26 bits). For comparison, MIPS is able to call any function in a 28-bit range, SPARC

any function in the full 32-bit range and POWER, in a 26-bit range. Therefore, in OpenISA and other ISAs that do not allow a call to the full 32-bit address space, if two functions are layout far apart and they have a caller-callee relationship, the linker needs to insert glue code to connect both functions.

6.2.4 Metadata design

A separate category of information conveyed by the binary is metadata. This is not necessarily part of the ISA definition, since it does not specify the format of any new instruction, but it is part of the ABI and offers extra information that piggybacks on the binary and that is useful to improve the static translation performance. This is important for the low-level program distribution format and to allow static translation but it is unnecessary for dynamic binary translation. Therefore, this information may be absent if the intent is to run OpenISA on a virtual machine. The metadata, thus, is the extra information we define in the distribution format based on OpenISA studied in this thesis.

Certain metadata do not need to be present and they can be inferred. Others are harder to infer if they are missing and will incur a significant overhead in the statically translated binary performance.

OpenISA works with a minimal amount of metadata. Some metadata may be transferred from the ABI to the ISA. For example, function begin and end instructions are metadata, acting as markers to guide the translator to easily recognize a translation region, but are part of the ISA nonetheless.

In this thesis, OpenISA utilizes the following binary metadata:

- Function canaries, as discussed earlier with *begin function* instructions in design choice 4;
- Function call parameter count, as discussed earlier with CALL instructions in design choice 3;
- Jump table size information, as discussed earlier with IJMP instructions in design in design choice 6;
- Code pointer markers in the form of linker relocations revealing all data section addresses that contain pointers to the code region.

The last item is crucial to avoid expensive hashtable lookups to convert target to host addresses on the fly. It could be replaced by mandatory LCP (load code pointer) instructions with operands as either immediates or a memory addresses, but relying on linker annotations to flag code pointers is simpler. By easily identifying code pointers, the translator can convert such addresses to host addresses, since they will not be the same after the binary gets translated.

IJMP and code pointers alone are responsible for cutting the runtime of 458.sjeng by half in x86 and bitcount by half in ARM, just by avoiding hashtable lookups and efficiently recovering jump tables.

Future work on metadata design

There are a few metadata possibilities that were not incorporated in this thesis, but may be subject of further investigation in future work.

Indirect calls could be changed to convey the same information as IJMPs in a new instruction named ICALL. This would help reduce the translation times of programs that abuse indirect calls because the translator would know exactly all possible targets of an indirect call and make more accurate translation decisions.

LCP (load pointer instructions) could replace linker annotations to identify code pointers.

Finally, all instructions could carry a KILL bit for input operands stating whether a given register is dead after this instruction. Liveness analysis can be tricky to calculate during runtime, but it is readily available during compilation. Although this could help OpenISA to eliminate store instructions that save unused values, improving design choice 3, OpenISA instructions currently lack enough bits to encode this.

6.3 Register mapping techniques

As discussed in the previous section, data access is the source of the largest overhead when translating OpenISA or any other ISA with no challenging operations. This thesis analyzes three different techniques to understand how this overhead can be mitigated or eliminated.

Figure 5.5 back from Section 5.4 showed a simple example where all guest registers were mapped to memory on the target architecture. This means whenever a guest register is used by any guest instruction, the translator issues a load from memory to retrieve such value, does the computation and then issues a store to memory. The net result is code with short-lived values, low host register usage and difficulty in extracting the data-flow graph, which now involves performing alias analysis.

One way to address this problem is to assume that guest registers are mapped to a host memory array (called global register bank) but their values are loaded to local variables at the beginning of the function and stored back to the memory array when the function transfers the execution to other functions. Since the register allocation step tries to map local variables to host registers, these values are likely to reside on host registers during the function execution. This approach enables the use of optimized code inside regions, but suffers with the synchronization overhead when code frequently alternates between two regions, e.g. functions, as discussed in Section 6.2.1

Figure 6.2 shows a straightforward illustrative example. Here, the guest register bank is represented by a single value stored in the global `my_global_reg1` (line 1). Whenever starting a region, it is possible to load this value to a local variable (line 3), allowing the compiler to assign fast host registers to handle computations involving this value. However, the original global value must be updated (line 5) before exiting the region to allow other regions to use it, incurring additional overhead.

The three different design options explored are as follows:

```
1 int my_global_reg1;
2 void function() {
3    int reg1 = my_global_reg1;
4    // ... do computation
5    my_global_reg1 = reg1;
6    return;
7 }
```

Figure 6.2: Local/global register synchronization overhead example

1. **All globals**: All register accesses are translated to global variables accesses in the same way as Figure 5.5 shows. This is called *Global registers* throughout this thesis.

2. **Local registers**: Some values at the global register file are promoted to local variables at the LLVM IR. A liveness analysis decide which values must be loaded/-stored. The region granularity is the function. This is called `F-BT` throughout this thesis because it is the *function-based binary translation*.

3. **Whole program binary translation**: There is no register synchronization overhead because the entire program is issued into a single region, which is compiled as a single host function. All values at the global register file are mapped to local variables. This mode not only removes register synchronization issues, but also exposes more optimization opportunities as all of them are now intraprocedural. This is called `WP-BT` throughout this thesis because it is the *whole program binary translation*, detailed next.

6.3.1 Whole program binary translation

This technique works by emitting the entire program control flow in a single LLVM function, which, in turn, gets compiled to a single host function. This does not mean that the program does not call functions, but instead that we do not use the host machine capability of calling functions.

Figure 6.3 shows the example of the translation of a fibonacci function using a C-based pseudocode to be easier to read. In reality, the static translator generates LLVM IR instead of C. In (a), the code illustrates the result of the fibonacci translation in a strictly global register configuration where the translator does not cache the registers in locals. Line 1 shows the OpenISA register bank declaration, but this example uses only register 4, which is used as the first argument for functions and to return a value, register 29, the stack pointer, and register 30, the return address register. Line 3 declares the fibonacci function but, since the translator does not know its name, it generates it with the hexadecimal address where the function resides in the binary memory layout, e.g. 0xA4. Lines 4-18 show the translated fibonacci function: notice that the shadow memory is constantly accessed to mimic the OpenISA stack operations. If the translator uses `F-BT`, the code will also have host stack operations to store the local registers, which effectively will run code that deals with two stacks, the OpenISA stack and the host stack. When a call is necessary, the translator encodes it as a call to a host function with the

```
1  int reg4, reg29, reg30;
2
3  int a4() {  // aka fib
4      shadowmemory[reg29--] = reg4;
5      shadowmemory[reg29--] = reg30;
6      if (reg4 == 1 || reg4 == 2) {
7          reg4 = 1;
8          reg29 += 8;
9          return;
10     }
11     --reg4;
12     reg30 = returnaddress;
13     a4();
14     reg4 = shadowmemory[reg29+4] - 2;
15     reg30 = returnaddress;
16     a4();
17     reg29 += 8;
18     return;
19 }
20
21 void main() {
22    (...)
23    reg4 = 21;
24    reg30 = returnaddress;
25    a4();
26    return;
27 }
```

(a) Using the *Global registers* configuration

```
1  int main() {  // only one host function
2      int local_r4, local_r29, local_r30;
3      local_r4 = 21;
4      local_r30 = ret1addr;
5      goto a4;
6  ret1:
7      return; // finish execution
8  a4:
9      shadowmemory[local_r29--] = local_r4 -
           1;
10     shadowmemory[local_r29--] = local_r30;
11     if (local_r4 == 1) {
12         local_r4 = 1;
13         local_r30 == shadowmemory[local_r29]
14         reg29 += 8;
15         if (local_r30 == ret1addr)
16             goto ret1;
17         if (local_r30 == ret2addr)
18             goto ret2;
19         goto ret3;
20     }
21     (...)
22     local_r4 = shadowmemory[local_r29+4] -
           1;
23     local_r30 = ret2addr;
24     goto a4; // backedge forms an inner loop
25 ret2:
26     local_r4 = shadowmemory[local_r29+4] -
           2;
27     local_r30 = ret3addr;
28     goto a4; // backedge forms the outer
           loop
29 ret3:
30     (...)
```

(b) Using WP-BT

Figure 6.3: Whole program binary translation fibonacci example in a C-based pseudocode

name corresponding to its address (line 13, line 16, line 25). Before making the actual call, notice that register 30 is updated with the return address to mimic the OpenISA behavior, but this value is never actually used. Instead, the translator relies on the host machine return mechanism.

In `WP-BT`, case (b) in Figure 6.3, when calling functions, the translator simply branches to the function label and save its return address (lines 4-5). At return points, a chain of if-elses tests for the return address and branches back to one of the possible return points (lines 15-19). In order to support indirect calls, it must know all targets, which is obtained from the ELF file by analyzing the code relocations. `WP-BT` also necessarily uses local registers instead of global registers, since the function scope is already the entire program.

The key insight in `WP-BT` for static translation is that, instead of maintaining two stacks, one for OpenISA and other for the host machine, it sticks with only one, OpenISA's. Consider that in the traditional translation, the host stack is used actively and at each function call, several load instructions fetch the register bank data from memory and stores in the host function frame. Simultaneously, OpenISA code is also setting up its own stack that resides in ShadowMemory, as well as saving the return address that will never be used. In WP-BT, the host stack is set only once, when the program begin execution, because there is only one host function. The overhead of calling OpenISA

functions is considerably smaller because there is no host frame setup and the LLVM target-independent optimizations may now crunch an entire program as it were a single function, easily eliminating unnecessary code at a global level.

6.4 A practical application of the static translation prototype

Previous sections presented the static binary translation prototype in the context of an argument to show that OpenISA conveys enough semantic information to allow translations, and mentioned metadata are important to use OpenISA for a secondary goal, the program distribution aspect of OpenISA. This section elaborates on a practical application of this secondary goal, illustrating the benefits of OpenISA in a scenario where program distribution, or deployment, can be tricky, such as in a heterogeneous cluster of Internet-of-Things (IoT) devices. The views discussed in this section will also be published[3] in an article of the journal Concurrency and Computation: Practice and Experience [26] as the result of the development of this thesis, in collaboration with Carlos Millani, Alisson Linhares and Alexandre Brisighello.

In the IoT scenario, the OpenISA virtual machine is called COISA VP, *Compact OpenISA virtual platform*. The VM emulation engine, which is responsible for executing the guest applications via guest ISA emulation, can make use of one of two different strategies: (1) a compact interpreter that fetches, decodes and executes guest instructions one by one or (2) directly execute native code downloaded from the cloud. We call the latter CATs, cloud-assisted translations, because IoT devices frequently do not have the resources to use the expensive algorithms required to perform static binary translation and optimization, but they can outsource this work to the cloud.

Figure 6.4.a discusses general deployment of programs to platforms running COISA. Program distribution happens as OpenISA binaries, but they can be translated to native binaries if necessary. Figure 6.4.b shows an interface of the virtual platform that can accept either OpenISA code, which will be executed by the virtual machine using interpretation, or native code, which will be directly executed.

Interpreters allow us to build compact and portable virtual machines because the code can be made quite simple and small [124]. However, they provide lower performance than binary translators or native code. Most IoT applications do not rely on performance because they are event driven – if events are handled without missing any deadlines, the application works fine.

If the performance of interpreters is a problem for a particular IoT application, it can leverage an external host outside the IoT device (either a development workstation or a cloud service) to translate the OpenISA binary to native code (assisted translation). This translation employs a static binary translator. In this case, the application cannot have Self Modifying Code (SMC), which is seldom the case for IoT applications.

Static binary translation is well-suited in the assisted translation scenario where it is beneficial to minimize the network traffic required to translate the program. While, in the-

[3]The paper was accepted to appear at the time of this writing

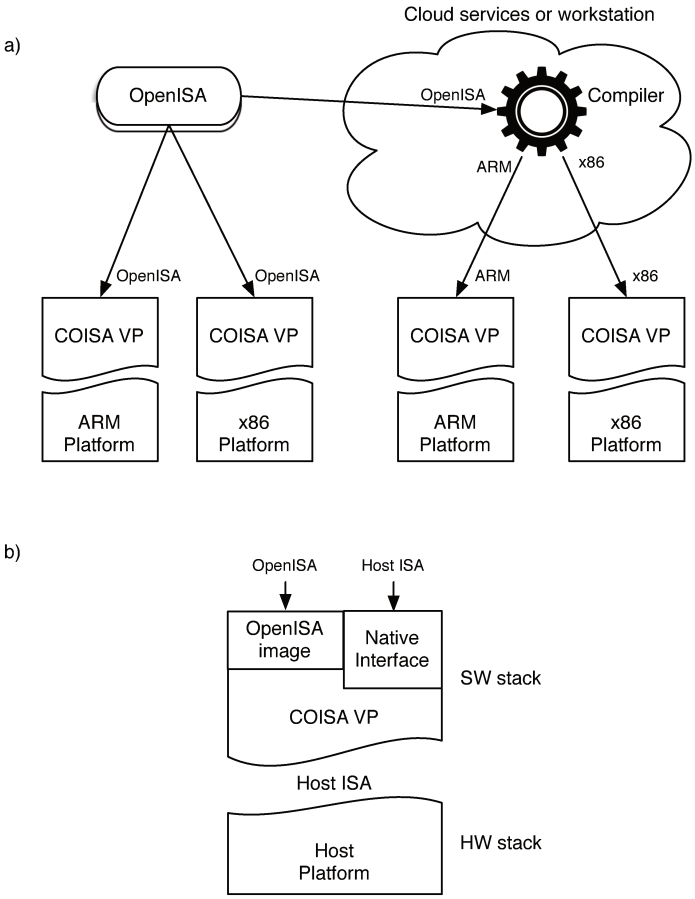


Figure 6.4: Diagram a) shows deployment of OpenISA binaries to simple platforms with limited resources, happening either directly to targets or assisted by an external compiler that translates OpenISA to the native ISA of the target. Diagram b) details the interface of the COISA Virtual Platform that can load and run OpenISA binaries directly or native code.

ory, it would be possible to perform dynamic translation and request a server to optimize a program fragment at a time, communication costs would likely be a significant overhead and the application would suffer with jitter, a sudden interruption of execution for a period of time, which conflicts with the real time requirements of some IoT applications. In static translation, communication costs are paid only once, during deployment.

Additionally, for medium or large platforms that require high performance, it is possible to replace the current emulation engine that uses interpretation by one that employs a high performing dynamic binary translator. In this sense, one of the applications of OpenISA is to achieve an IoT scenario with different flavors of the COISA virtual platform, ranging from compact to high performing systems, each one requiring different amounts of hardware resources, but providing transparent software compatibility across a wide variety of hardware platforms.

6.5 Summary

OpenISA began with a MIPS-based ISA as a starting point owing to strong evidence suggesting MIPS is a simple and easy ISA to be emulated. The experimental framework developed for this thesis was built to answer the question of whether it is possible to design an ISA that preserves enough semantic information to allow its translation to other ISAs without losing performance. It also allows the identification of common overhead sources when translating RISC ISAs, which was used to shape the design of OpenISA.

The experiments revealed that a common theme among the translation deficiencies concerns how data is accessed. OpenISA design decisions, therefore, are geared towards making data access straightforward or easy to be inferred. As such, an important consequence is that the translator design of how the OpenISA register accesses are translated has a major influence on performance. This thesis then proceeds to investigate different guest register access methods in binary translation. The prototype implements three different methods: accessing all registers as global variables, buffering some registers in local variables and translating the entire program at once, in a single region.

Chapter 7

Experimental Results

This chapter presents experimental results produced by the OpenISA static binary translator prototype. Its contents are divided into a brief description of the utilized host platforms, starting with the presentation and analysis of the results of the simple benchmarks that compose Shootout to give an overall sense of how a static translator impacts small kernels, and then proceeds to present and analyze the results of more complex and real-world benchmarks, part of Mibench and SPEC CPU2006.

7.1 Host Platform

All experiments utilized an Intel Xeon E5-2630v2 (Ivy Bridge-EP) at 2.6GHz as the x86 host, running Debian Jessie on a regular hard drive, but all experimental software running on a RAM disk to discard disk interference. For ARM, experiments used a stock Samsung Galaxy SIII, which has an ARM Cortex A9 1.4 GHz platform with a flash disk, running Android 4.1.2, Linux kernel version 3.0.31. We used the Android Debug Bridge (ADB) to run ARM native programs in its underlying Linux operating system and used a custom kernel that enabled hardware performance counters.

7.2 Simple benchmarks

The graph in Figure 7.1 shows the slowdowns relative to native execution when running the code produced by the static binary translator prototype to translate OpenISA programs to x86 and ARM hosts, respectively.

Errors: All measurements were repeated 10 times. The error in this graph is below 1% for all cases of x86, except for lists, where it is 3%. For ARM, the difference between measurements can reach up to 10% because they were made in an Android OS where interference can be high. There is no error characterization in ARM because of their nature. The ARM platform exhibited errors that did not obey a known probability density function. Therefore, this thesis conservatively works with a fixed error rate of 10% for all ARM measurements. It is also worth mentioning that the time of program runtime spent in libc is factored out because libc code is not translated, as mentioned in Section 5.5. To do this, an statistical profiler helps to determine when code is running in libc and when

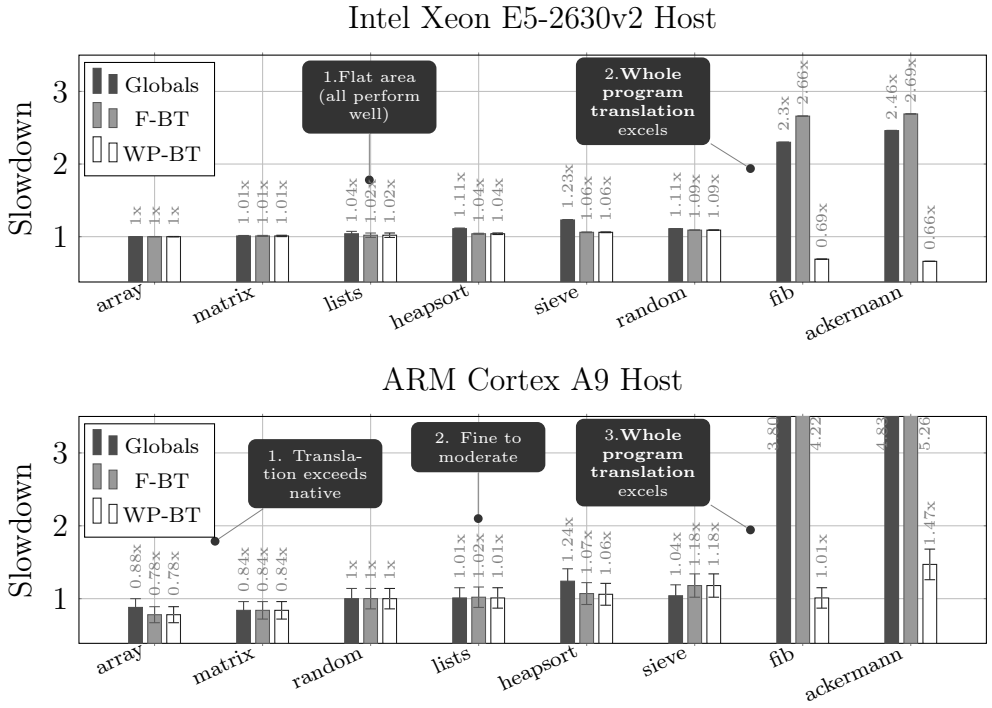


Figure 7.1: Ordered slowdows for OpenISA to x86 and ARM translations, relative to native performance - Shootout Programs (simple kernels)

it is not. Most part of the errors in the x86 platform comes from the inherent nature of the statistical profiler, which makes the normal distribution an accurate error model for x86. Programs that spend a significant amount of time in libc were removed from the analysis because they lead to large errors. The removed programs are all from Mibench: basicmath, patricia, blowfish, qsort and crc. Their results also did not show anything new, which justified their removal from results.

Analysis: The analysis starts with simple programs from the Shootout benchmark to analyze which effects contribute to a good binary translation. Later, analysis will cover more complex programs and potential second order effects.

Each bar in the graph of Figure 7.1 represents the translation using a different register mapping technique: *all globals*, *F-BT* and *WP-BT*. For example, the Fibonacci program, if translated using *Globals*, needs 2.3x the time that the same program, compiled directly to x86, takes to complete its task. This is a significant overhead and represents a poor translation example. However, the same program, if translated using *WP-BT*, may become, in fact, faster than the native version by using only 0.69x the time of the native counterpart. Fibonacci is an example where the translation scope has a large importance in code quality.

The first insight of this experiment comes from the order of magnitude of the overhead being discussed in the best approaches of each program. These numbers present low overheads for a cross-ISA binary translation, supporting this thesis' claim that it is possible to keep native performance even if the program was compiled to an entirely different ISA. The code quality alone, without discussing other overhead sources from a DBT engine such as code profiling, can be kept very good, which suggests that compiling a code to

OpenISA or a RISC-like variant preserves enough semantic information.

Another important observation about the results is that accessing the OpenISA registers with the *Globals* configuration does not always affect performance, despite being the simplest approach of the three. Sometimes it may be even more efficient than the *F-BT* configuration because it does not pay the price of synchronization, which is observable in benchmarks that make many function calls, e.g. fibonacci and ackermann.

Sieve, in x86, and Heapsort, in x86 and ARM, show that caching OpenISA registers in *F-BT* plays an important role in achieving near-native performance. The reason is that even in the absence of cache misses, using load/store instructions to access OpenISA registers increases the number of host instructions. Yet, the most infamous effect of using memory instead of registers is the difficulty of further optimizing the code owing to the aliasing problem. Since binaries are already optimized, this effect is not apparent in this experimental evaluation.

The superiority of whole program translation (WP-BT) is evident in this data, which only loses to other techniques by small differences in few cases. A significant advantage of WP-BT appears in recursive functions, such as fibonacci and ackermann. Since the program is compiled as a single function, LLVM optimizes the recursive calls as loops that gradually updates an array, which is, in reality, the return addresses being saved in the OpenISA stack. The result is a fibonacci that runs at almost half the original native speed, beating even the native x86 compiler with a full O3 optimization pipeline.

7.2.1 The Fibonacci Case

Fibonacci for ARM suffers a larger overhead (4.22x) than fibonacci for x86 (2.66x) for F-BT. For *Globals*, this number goes to 3.8x of overhead on ARM platforms against 2.3x on x86. Both platforms show worse results when using the Locals technique because frequent function calls force a high number of register bank synchronisation, if using Locals technique, as opposed to always directly accessing the global register bank in the *Globals* technique.

The larger overhead with the F-BT technique is a symptom of a bad boundary for the translation region, one in which the overhead of exiting and entering the region erodes the benefits of prefetching OpenISA registers in the host registers. In the case of excessive recursive calls such as in the Fibonacci benchmark, the call site and the recursive function should always be compiled as a single region to avoid the excessive overhead witnessed in this experiment.

In ARMv7, accessing the global register bank requires 3 instructions: load lower part of the memory address, load upper part and the load/store instruction per se. In x86, a single instruction performs this task. Owing to the recursive nature of the fib benchmark, most of the overhead of the translated version concentrates on OpenISA registers load/store, making it a fair benchmark for comparing synchronization overheads faced on ARM versus x86. Contrary to x86, the strategy seen in ARM code to handle a large number of global variable accesses (to access OpenISA registers) is to, instead of always loading the high and low part of the address, to try to load the addresses earlier and to hoist frequently used addresses, increasing register pressure. These results, in particular, show that x86 is

better than ARM at loading global variables.

Another issue the Fibonacci benchmark faces is lacking proper dead code removal because of the lack of a thorough liveness analysis pass. In the prototype, if an OpenISA register is used at any point in a function, this register is considered live and is synchronized in all checkpoints (function calls and returns) of this region. Even worse than the extra load/store instruction is the necessity to perform an unused computation. In Fibonacci, this issue appears with the use of the `slti` instruction in OpenISA code, which leverages a general purpose register to store the result of a comparison and then to jump or not based on the contents of this register. The x86 optimizes the comparison to use its `EFLAGS` register without the need to waste a general purpose register on this. However, the contents of the original OpenISA register used to store the result of the comparison is still calculated and stored in the global register bank in translated code because it is wrongly considered to be used.

7.2.2 Whole program translation beating native performance

After translation with WPT, fibonacci gets translated to a pair of loops: the first loop goes deeper into the stack, decrementing the argument by 1 at each iteration, emulating the first tail call found in the fibonacci algorithm. The second loop, at each iteration, unwinds the stack, adds the result of the last recursive call to a running total and, if necessary (in a given stack frame, if the value of n is greater than 2), goes back to the first loop to further emulate more calls. Thus, one loop emulates function calls while the other emulates function returns while updating the running total (the fibonacci calculation *per se*).

The translated code can still be improved. The first loop writes the value 248 at each iteration in the current stack frame, mimicking the return address being written. Later, this value is checked to see if the recursion should continue (if the return address is 248) or if it should end (if the return address is not 248). This could be replaced with a more efficient technique to detect the end of the recursion and increase the performance even further. This is a consequence of how fundamentally inefficient are recursive algorithms, despite elegant, and how they abuse the stack. Even suboptimal, this configuration alone is enough to beat native performance in x86. When rewriting the code as a loop, the compiler is able to optimize and properly schedule instructions. When running the native code as a recursion, this is not possible.

In the case of ARM, the code is similar to that of x86, except for the fact that it uses slightly more instructions when in need of accessing memory relative to the address of a global variable, as frequently is the case to access guest memory contents.

Other people observed the benefits of converting recursions to loops and this gave rise to an RFC in the LLVM developers mailing list about whether it would be good to have an LLVM pass to perform this kind of transformation. It is still unimplemented[1].

[1] See `http://lists.llvm.org/pipermail/llvm-dev/2015-February/081747.html`

	Native ARM	Translated to ARM
Cycles	635,238,444	533,801,950
Instructions	604,180,935	690,311,952
IPC	0.95	1.29
Cache accesses	246,196,048	248,865,483
Cache misses	6,680	6,680
Cache misses per K instructions	0.011	0.010
Branch instructions	81,072,587	81,070,395
Branch misses	2,821,766	2,820,700
Branch misses per K instructions	4.670	4.086

Table 7.1: Comparison of native and translated versions of the matrix benchmark

7.2.3 The Matrix Case

This program is remarkably well translated both for x86 and ARM. In the case of ARM, the performance differs between native and translated, in favor of the translated version. Table 7.1 presents the hardware performance counters for both versions and shows that the translated version, despite executing 14.3% more instructions, has better performance (uses less cycles). Table 7.1 also shows that the number of cache faults and branch mispredictions is similar and therefore it should not be the cause of the performance difference. Figure 7.2 compares the code of each version. The extra instruction of the translated version is necessary to add the base offset of the guest memory.

Native ARM	OpenISA	Translated to ARM

`.LBB0_10:`
`ldr r6, [r5, -r0, lsl #2]`
`ldr r1, [r2, -r0, lsl #2]`
`sub r0, r0, #1`
`cmn r0, #30`
`ldr r6, [r6, r7, lsl #2]`
`mla r4, r6, r1, r4`
`bne .LBB0_10`
	`$BB0_10:`
	`add $1, $17, $10`
	`ldw $1, 0($1)`
	`add $11, $6, $10`
	`ldw $12, 0($11)`
	`shl $11, $8, 2`
	`add $1, $1, $11`
	`ldw $1, 0($1)`
	`addi $10, $10, 4`
	`mul $0, $$1, $1, $12`
	`addu $9, $1, $9`
	`jne $10, $2, $BB0_10`
		`.LBB0_12:`
		`ldr r4, [r5, -r2]`
		`ldr r12, [r3, -r2]`
		`sub r2, r2, #4`
		`add r4, r4, r6`
		`cmn r2, #120`
		`ldr r4, [r7, r4]`
		`mla r0, r4, r12, r0`
		`bne .LBB0_12`

Figure 7.2: Comparison of three versions of matrix at the hottest basic block

In general, the extra instructions generated when translating from OpenISA are always related to adding the extra guest memory base offset. Curiously, this sequence of 8 instructions run faster than the original sequence of 7 instructions. The extra shift used in the operands of LDR in the native version of the code does not impact code performance [22]. Aside from the shift, since the only difference comes from the extra

	Native ARM	Translated to ARM
Cycles	8,721,356,316	13,054,281,502
Instructions	6,450,379,244	7,881,305,755
IPC	0.74	0.60
Cache accesses	833,958,460	1,017,119,071
Cache misses	172,863,164	488,720,524
Cache misses per K instructions	26.799	62.010
Branch instructions	1,074,343,445	1,790,460,187
Branch misses	475,337	716,791
Branch misses per K instructions	0.074	0.091

Table 7.2: Comparison of native and translated versions of the ackermann benchmark

ADD instruction, we conjecture that the root cause is a microarchitecture issue. Since Cortex A9 is out of order and is capable of dynamic scheduling, the native version may be suffering with a scheduling problem.

7.2.4 The Ackermann Case

In x86, Ackermann has performance similar to Fibonacci because both are recursive benchmarks that explore the same difficulties in code translation. In ARM, however, there is a large discrepancy between Fibonacci with whole program translation and Ackermann with whole program translation. The Fibonacci version has the same performance of the native code, whereas Ackermann has almost 50% of overhead. Table 7.2 presents the hardware performance counters.

What happens in this case is that native Ackermann is a recursive benchmark that fully occupies the data cache with stack frames, and the native edition uses 16B frames, while the OpenISA version uses 24B frames, 50% larger. The LLVM Mips backend conservatively allocates 16B in frames of functions that are not leaf, in case it needs to pass extra parameters via stack. Thus, this is a deficiency of the original MIPS backend modified to work with OpenISA.

Upon manually changing the translated code to allocate only 16B, the cycle count is reduced from 13B to 10B, consequently reducing the overhead from 50% to 14%.

7.2.5 The Sieve Case

Sieve is an array accessing benchmark that achieves a perfect (no overhead) translation in x86, except for *Globals*, which entails a memory access for every OpenISA register and is expected to impose overhead on CPU-intensive benchmarks. In ARM we observe the opposite behavior: *Globals* suffer almost no overhead, while WP-BT suffer 16% overhead.

For ARM, the native version runs in 1.8B cycles and 2B instructions (IPC of 1.11). The whole program translated version, on the other hand, runs in 2.1B cycles and 3B instructions (IPC of 1.42). Even though the out-of-order architecture absorbs good part of the impact of the extra instructions seen in the translated version, it is still a workload

with 50% more instructions. Both versions have almost zero cache misses and identical branch prediction footprint. Thus, it is possible to attribute the performance discrepancy to the considerably larger number of instructions seen in the translated version.

Native ARM	OpenISA	Translated to ARM

`.LBB0_5: strb r7, [r5, r2]`
`add r2, r2, r0`
`cmp r2, #8192`
`ble .LBB0_5`
	`$BB0_5: add $1, $17, $3`
	`add $3, $3, $2`
	`stb $zero, 0($1)`
	`slti $1, $3, 8193`
	`jnez $1, $BB0_5`
		`.LBB0_7: add r4, r6, r3`
		`add r3, r3, r2`
		`strb r5, [r4, r7]`
		`add r4, r3, #4`
		`cmp r4, #8192`
		`ble .LBB0_7`

R5 contains the vector address, R2, the induction variable and R0, the step size.	$17 contains the vector address, $3, the induction variable and $2, the step size.	R6 holds the guest memory base position, R3 the induction variable, R2 the step size and R7, the address of the vector relative to the guest memory base.

Figure 7.3: Comparison of three versions of sieve at the hottest basic block

The inner loop for this benchmark is simple. In the native version, it consists of 4 ARM instructions : store byte to zero the array positions that corresponds to non-prime indexes, add a constant to the index in order to continue the arithmetic progression, a compare instruction to check if we are out of bounds and a corresponding branch instruction. Unfortunately, the translated version features 6 ARM instructions (50% more instructions): the first extra instruction is used to add the guest memory offset, which is unnecessary in the x86 ISA because it is able to do this with a single instruction. The second adds a fixed offset to the induction variable before it is compared for loop bounds in the loop header. This offset could be deducted from the bound value, but then the immediate would not fit into ARM's limited immediate encoding mechanism. This value is a multiple of 2 and fits in ARM's immediate encoding mechanism, but it was transformed to an odd number in the OpenISA backend to avoid the usage of a "less than or equal" comparison and emit an `slt` instruction. This could be optimized back to a comparison against the correct immediate.

The *Globals* version of this program exposes calculations that are easily hoisted off the inner loop and results in only 4 ARM instructions, equivalent to the native version, and this is the reason why *Globals* outperforms all others.

In the x86 version, the native version uses 4 x86 instruction whereas the translated version (whole program) uses 5 instructions, but this extra instruction is not on the critical path and the processor scheduler is able to avoid the latency.

7.2.6 The Array Case

Similarly to sieve, this benchmark iterates over a large array. Different than sieve, it not only writes to array positions, but reads the contents of two different arrays, adds them and writes back the result to the same position of one of the input arrays. On ARM,

the translated version outperforms the native version by 20%. This happens because the translated version uses 6 instructions in the innermost loop, while the native version, 7 instructions. Both versions uses 2 load instructions to load operands and one store instruction, but the difference occurs on the compare instruction, which is discarded on the translated version due to a simple ARM induction variable optimization that uses an arithmetic instruction to set the necessary flags to check loop bounds. It is unclear, however, why the compiler missed this optimization in native compilation.

The performance difference between the two versions is entirely due to this extra instruction, since this is a hot loop that gets repeated throughout the entire benchmark duration. The native version achieves an IPC of 1.8 while the translated one, 1.66, since it is a more compact version that uses 1 less instruction.

The previous analysis leads to the conclusion that the binary translation of small kernels is effective. Thus, OpenISA does not lack any information that is important to allow its retranslation in these cases. Next, larger benchmarks are analyzed.

7.3 Complex Benchmarks

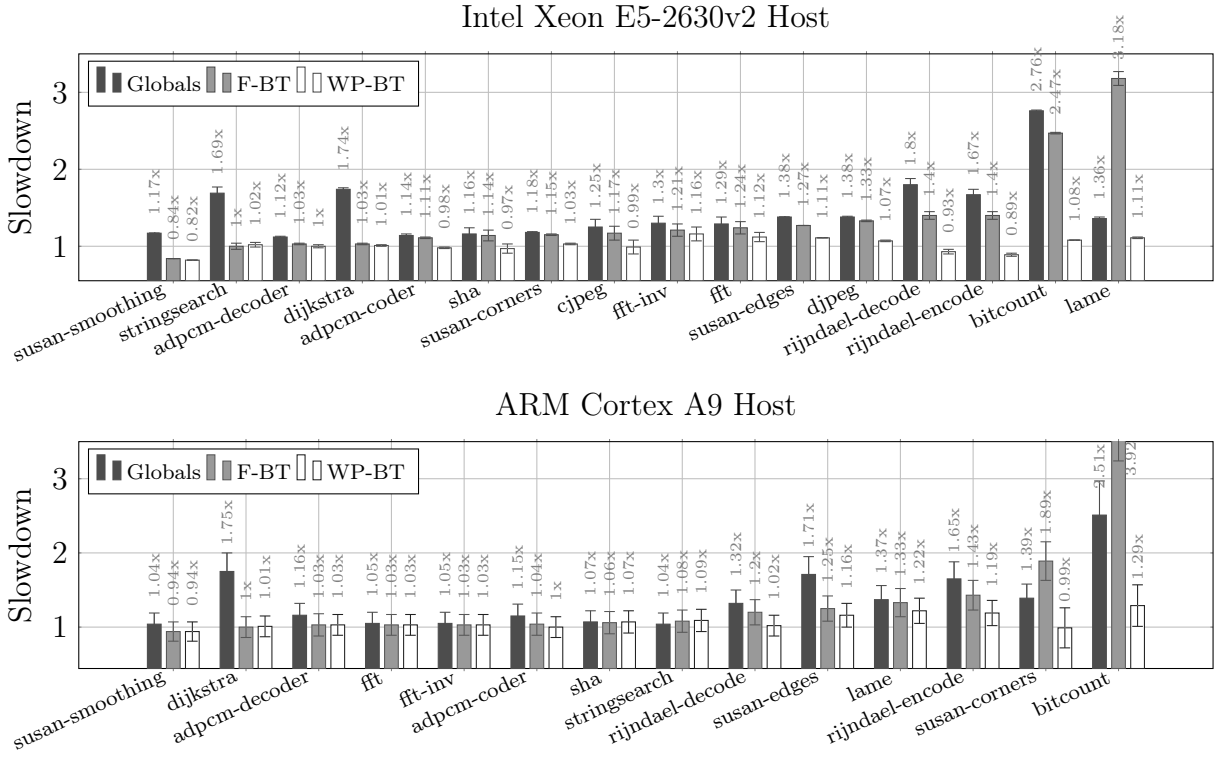


Figure 7.4: Ordered slowdows for OpenISA to x86 and ARM translations, relative to native performance - Mibench Programs (Complex programs)

Figure 7.4 shows the runtimes for more complex programs from Mibench using three different register mapping strategies again, for ARM and x86. As programs grow larger, it may become increasingly difficult to apply the WP-BT technique in some cases where the program uses many indirect calls and has many functions. For `cjpeg` and `djpeg`, for

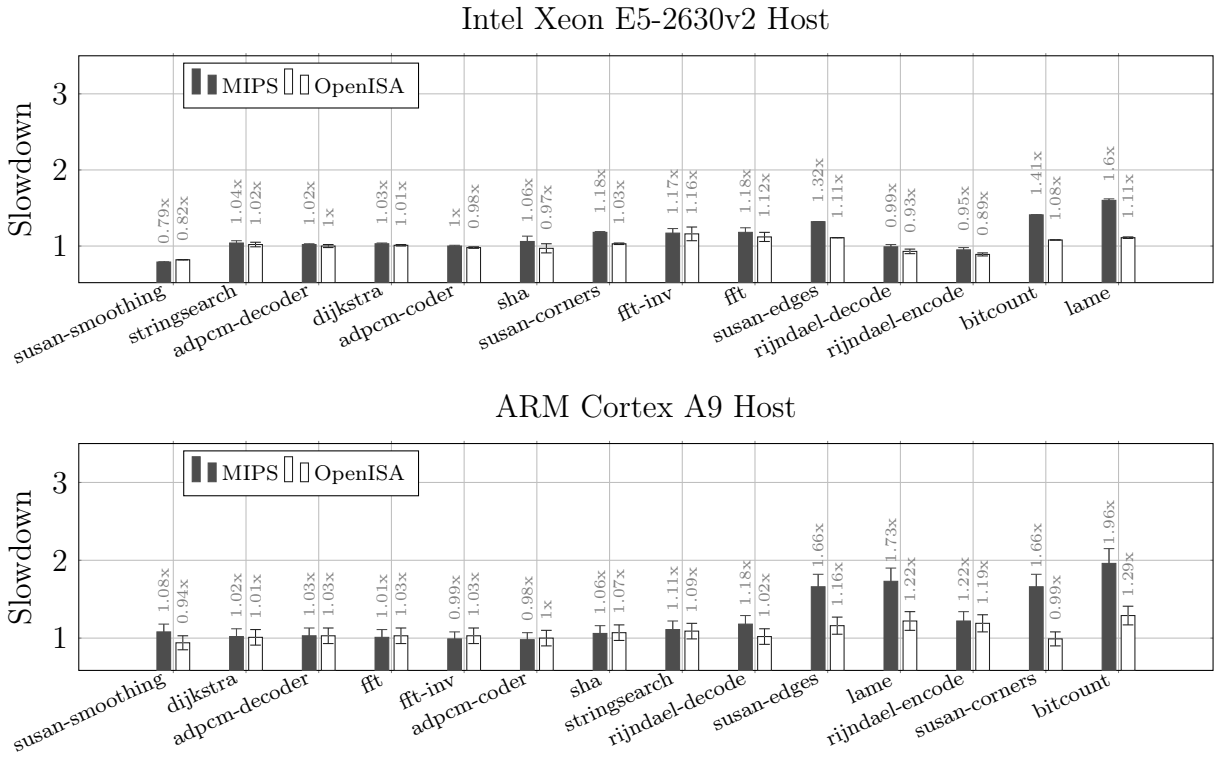


Figure 7.5: Comparison of MIPS and OpenISA translation performance in Mibench (lower is better), showing only WP-BT translations

example, there are 985 indirect calls and 181 possible call targets. In this case, WP-BT will translate each indirect call to an indirect jump with 181 successors accounting for all potential targets, causing the optimization pipeline for this case to take up to 5 hours to complete. This is an exceptionally slow outlier, since optimization typically takes a few minutest at most. We will discuss how to overcome the shortcomings of WP-BT in the next section.

To understand how OpenISA design decisions discussed in Chapter 6 affect emulation performance, Figure 7.5 shows the runtime relative to native execution in each Mibench program for the WP-BT register mapping technique, comparing the original MIPS and OpenISA. The largest gains come from programs that use many registers in hot regions and may spill registers. Avoiding these spills by increasing the register bank size is important for translated code performance, but it also leads to a higher overhead in *Locals* owing to the larger number of registers to be synchronized. Boosts in performance seen in lame and bitcount are partly due to the introduction of metadata (see Section 6.2.4) to aid in indirect calls. The introduction of the `IJMP` instruction, on the other hand, not only boosts performance, but is also essential to recover an accurate CFG for programs that abuse indirect jumps. The absence of such instruction by using the standard MIPS ISA, for example, may force us to create an overly conservative CFG that blocks optimizations. Mibench, however, does not have any such program and this case will be presented with SPEC benchmarks later. Next, we discuss individual cases in Mibench that stand out from the others.

7.3.1 The SUSAN Case

SUSAN is an image processing algorithm that traverses all pixels, computes convolutions and performs edges detection. It is a single program that can run three different algorithms, leading to three separate benchmarks: `susan-smoothing`, `susan-edges` and `susan-corners`. It suffers with poor branch predictor performance (circa 40% miss ratio), but this happens in both native and translated. The important insight from this benchmark comes from the high overhead of the translated version when the guest ISA has a small register bank.

	Native ARM	OpenISA 16 regs	OpenISA 32 regs	OpenISA 64 regs	OpenISA 78 regs
Cycles	9,815,980,596	24,063,114,815	17,453,761,518	11,441,313,479	10,893,395,296
Overhead	1.00x	2.45x	1.78x	1.16x	1.11x
Instructions	9,508,655,463	20,993,259,504	15,631,231,041	11,226,165,587	10,633,445,949
IPC	0.97	0.87	0.90	0.98	0.98
Cache accesses	3,755,362,901	13,829,071,252	9,032,216,020	4,827,352,288	4,196,754,312
Cache misses	7,343,401	8,130,340	7,959,849	7,467,843	8,042,481
Cache MPKI	0.772	0.387	0.509	0.665	0.756
Branch instructions	363,447,733	391,265,639	349,981,747	366,633,181	358,033,342
Branch misses	54,267,182	53,888,027	55,316,464	56,223,689	55,213,890
Branch MPKI	5.707	2.567	3.539	5.008	5.192

Table 7.3: Comparison of native and translated versions of the SUSAN-edges benchmark

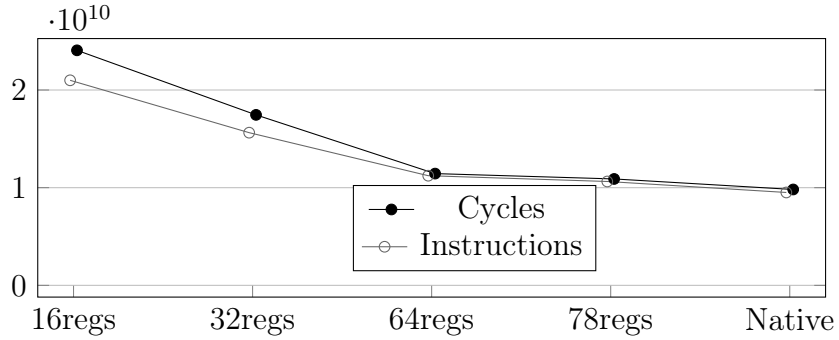


Figure 7.6: Instruction and cycle count of native and translated SUSAN using different sizes for the register bank of the guest ISA

Table 7.3 contains hardware performance counters for native and the translated code from different OpenISA versions, each one with a different size for the general purpose register bank. As the number of registers goes down, the OpenISA code will have more spills. When translated to ARM, accesses to the OpenISA stack requires adding the guest memory base address and, thus, more overhead. As the OpenISA register bank increases, spills in OpenISA code suffer a significant reduction. However, the number of host (ARM) registers is the same, and eventually this higher register pressure is observed as an increase in the number of spills in the translated ARM code. But in this case concerns *native* spills, or spills to the native ARM stack, which can be more easily addressed via the ARM stack pointer. Therefore, even though the number of required registers for this program is the same, the performance increases, peaking at 78 registers, which is in line with the observations made by Alipour et al [19]. Figure 7.6 shows the relationship of instruction and cycle count of the translated version and the size of the register bank of the guest ISA.

7.3.2 The Rijndael Case

Rijndael also benefits with improved translation quality owing to a larger register bank (it can use up to 74 registers, if available). However, the translation to ARM still has overhead and the remaining performance gap is a consequence of too many memory accesses that needs to index the guest memory. In the `encrypt` loop body, Rijndael performs several loads that will feed a chain of shifts, adds and xors that summarizes the computation performed in a given iteration. The typical load address in this program consists of three components: the guest memory base address, one of a range of possible constant (expressing the address of a global symbol) and a value that is calculated on the fly in the loop body.

The ARM instruction set can only add two components to form the address of a single load instruction. The compiler is unable to fuse the two constant values guest memory base address plus known location of a global variable because the guest memory base is an unknown value before link time. On the other hand, in the x86, the sum `GuestImage+Global symbol address` is attributed to a single `mov` instruction. When performing the final link, the linker simply patches the `mov` instruction with this sum.

The x86 strategy (symbol value added to a known offset) cannot be employed in ARM when modern ARM code is used to load large immediates. In this situation, the backend uses a pair of `movw`/`movt` instructions to load, respectively, the lower and the upper parts of a 32-bit immediate. If the immediate is a global symbol, such as the guest memory array, it emits a relocation [88] against both instructions, telling the linker to patch those two instructions with the lower and upper parts of the symbol `GuestMemory`. The trick used for the x86 of specifying an addend equal to the constant value does not work here because the two relocations are considered independently, making it impossible to propagate carry from the lower part to the upper part.

	Native ARM	Translated from OpenISA to an old ARM ISA version	Native ARM (using an old ARM ISA version)
Cycles	2,056,055,696	2,088,018,020	2,226,210,151
Instructions	1,726,135,418	1,850,956,091	1,786,338,762
IPC	0.84	0.89	0.80
Cache accesses	721,549,318	804,470,358	785,429,691
Cache misses	2,395,956	2,025,802	2,425,601
Cache misses per K instructions	1.388	1.094	1.358
Branch instructions	78,711,294	83,399,125	80,447,388
Branch misses	9,050,198	12,195,771	10,527,923
Branch misses per K instructions	5.243	6.589	5.894

Table 7.4: Comparison of native and translated versions of the rijndael benchmark

An alternative would be to create a constant island and issue relocations against `GuestMemory` plus a constant offset. Since each relocation covers 32 bits of data, the carry

problem vanishes and the linker works just as it would work for x86. However, loading immediates from a constant island is costly: it increases the number of data memory accesses and thus pressure on the data cache. For some applications, however, this may not be a problem. For Rijndael, for example, when generating the translated code with an ARM backend for an old ARM version that does not have `movw`/`movt` instructions, the results come much closer to native. See Table 7.4 for the hardware performance counters of the native version, the translated version used an old ARM backend and the translated version using the regular ARM backend.

Using the old ARM target, even though the backend could amalgamate `GuestImage` plus constant offset, it does not, which is a problem in the ARM backend. However, even without this feature, it can come much closer to native performance because of exchanging `movw`/`movt` with load values from constant islands.

This explains why, for ARM, even though the translated version may not increase cache misses or branch mispredictions, it increases the number of instructions executed. The higher the number of global variable accesses, the higher is the overhead.

7.3.3 The LAME Case

In LAME, the main overhead comes from LLVM failing to eliminate dead code in important loops. For example, when converting from floating point numbers to integers, the code not only stores the integer form, which is used in the program, but also the original floating point value, unused. While it may make sense to store it to mimic the register state of the guest machine, it is still dead code because it is not used anywhere in the program. Therefore, this comes from a compiler deficiency that is unrelated with the guest ISA, the subject of study of this thesis.

7.4 A closer look on WP-BT

Even though WP-BT frequently exhibits the best possible translation quality results in these experiments, it may not be an adequate technique if the translator needs to finish the translation as soon as possible. It requires extra resources because it involves compiling an entire program in a single host function. Thus, all optimizations that operate at the intraprocedure level are now also interprocedural, and compilation becomes more thorough but more expensive. It is clear that the super large region size (encompassing the whole program) is beneficial, but it is unclear why. One important research question is whether the difference between *F-BT* and WP-BT comes from the lack of synchronization overhead in WP-BT or if the extra optimization opportunities exposed by WP-BT play a key role in guaranteeing near-native performance.

To clarify this, a new experiment with another register mapping technique is presented next. Its goal is to reduce the synchronization overhead of *F-BT* and verify if it achieves code quality on par with WP-BT, thereby showing that most of the performance seen in WP-BT comes from the absence of register synchronization. To accomplish this, this technique restricted the number of registers synchronized between function calls to those that are predicted by the ABI to pass argument information. It also uses the information

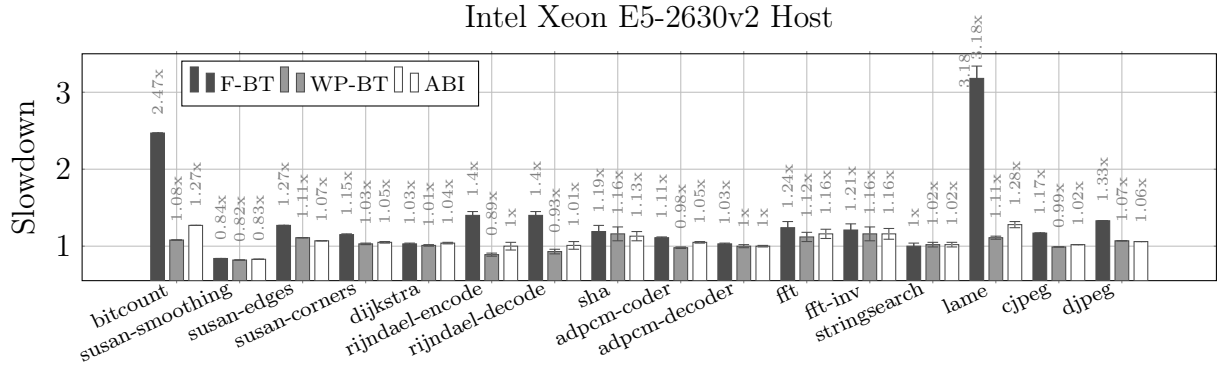


Figure 7.7: Ordered slowdows for OpenISA to x86 translation, relative to native performance, comparing F-BT augmented with ABI information - Mibench Programs (Complex programs)

of number of parameters passed in a call, an OpenISA extension, to further increase the accuracy of exactly which registers must be synchronized and carry useful information from one region to another. With this approach, the translator loses the capacity to emulate code that does not strictly follow a specific ABI, but it fits the purposes of this experiment.

Figure 7.7 presents the results collected along with the ratios over native time for *F-BT* and WP-BT, for easier comparison. For example, the graph shows that the runtime of `lame F-BT` is 3.18x slower than native and `lame WP-BT` is 1.11x, but *F-BT* augmented with ABI information achieves 1.28x, indeed showing that most of the speedup of WP-BT comes from solving a register synchronization issue. A similar pattern appears for all programs that previously had a large gap between *F-BT* and WP-BT, with the special case of Bitcount, which had one of the worst performance for *F-BT* augmented with ABI information, circa 1.5x slower than native. However, the introduction of design choice 3 (see Section 6.2) increases the information about which registers need to be synchronized, improving the performance of bitcount to 1.27x slower than native.

The Bitcount benchmark is a tight loop where the bit counting algorithms *per-se* are fast, but they depend on an indirect call to select which algorithm to run. Since the call mechanism is in the critical path for this benchmark, the synchronization overheads imposed by the function stack frame creation are more pronounced.

7.5 SPEC CPU2006 programs

SPEC CPU2006 benchmarks results are presented by Figure 7.8 for x86. Most SPEC programs behave in a similar way to Mibench ones, but there are cases demanding a deeper analysis: 458.sjeng exhibiting bad performance in WP-BT with 42% of overhead but surprisingly better performance in ABI-augmented F-BT and 464.h264ref showing a bad performance for ABI-augmented F-BT with 82% overhead. Another difference with SPEC is that its programs use indirect calls and indirect jumps more frequently, making it important to have the IJMP instruction to ease the task of translating such programs. 458.sjeng, for instance, has 41 indirect jumps. 401.bzip2, on the other hand, has 69

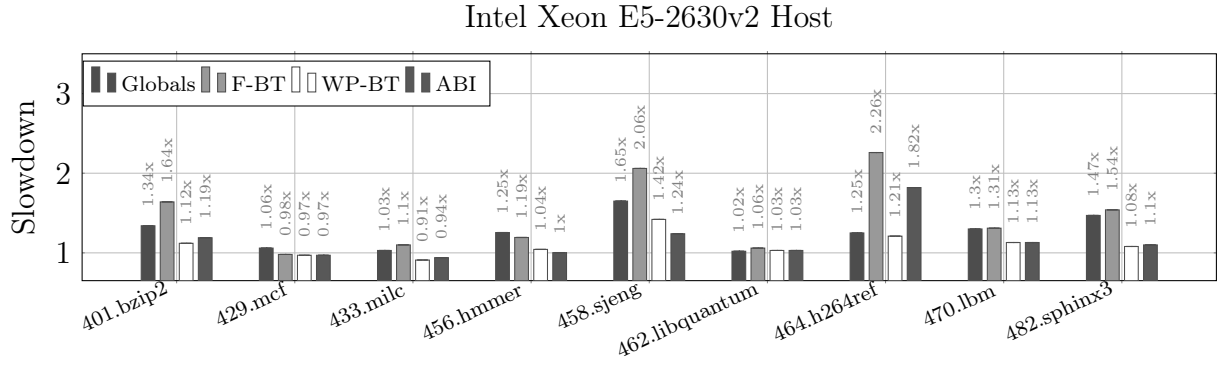


Figure 7.8: Ordered slowdows for OpenISA to x86 translation, relative to native performance - SPEC CPU2006

indirect call sites and 464.h264ref, 391 indirect call sites.

In 458.sjeng, ABI-augmented F-BT beating WP-BT is an indication that converting return instructions to indirect branches, as performed by WP-BT, is quite demanding on the branch predictor unit, which is failing to predict return addresses. Further evidence of this hypothesis is the number of branch mispredictions: in the WP-BT version of sjeng it is 22B versus 18B in ABI-augmented F-BT. This happens if there are many call sites for the same function, making the indirect jump, which models the return behavior, to be non-trivial to predict.

The 464.h264ref benchmark shows another noteworthy result. This time, ABI-augmented F-BT does not achieve the same performance as ABI or even globals, the simplest technique. In programs with few inlined functions and many function calls, the synchronization overhead can play an important role in performance. When the globals technique outperform others, we have clear evidence that register synchronization issues are dragging performance. This can be improved if call instructions are changed to carry information about how many arguments of each register bank, double, float or integer is being used. To fit the call instruction into a 32-bit encoding, its encoding has the *total* number of arguments, making F-BT less precise than it could be. However, by encoding calls into a 64-bit encoding, it is possible to improve this information.

7.6 Summary

The crux of OpenISA is the idea of easy translation to other machines. Our experiments show that OpenISA preserves enough semantic information to allow it to be emulated on other hosts with low overhead due to code quality. When collecting data, we developed a static translation technique to remove the overhead when entering or leaving the translation region called WP-BT and other technique to mitigate this overhead, the ABI-augmented F-BT. In these techniques, when targeting x86, the overhead purely due to the difficulty in translating the guest ISA is no more than 24%, which happens in 458.sjeng, and under 10% for the majority of programs (22 out of 32 program tested). When targeting ARM, this overhead is no more than 29%, which happens in bitcount, and under 10% for the majority of programs (16 of 22 programs tested). In contrast, the

best static translation framework we found in literature, LLBT [121], which translates ARM to x86, suffers 66% of overhead on average in the EEMBC benchmark, and can be up to 4x slower than native.

To achieve this, OpenISA is based on a simple RISC architecture with 6 important design changes and, specifically targeting a better static translation framework, preserves 4 metadata categories with information about the binary and was tested with 4 different translation techniques. The combination of changes in the guest ISA, metadata and the control of the translation region size or the register synchronization overhead leads to the best static translation results.

Finally, it is worth remembering that the OpenISA binaries that were translated were already optimized. This is an important assumption because if a binary that is not optimized is employed, all values are kept on memory rather than on registers. When values go to memory, it is difficult to reconstruct dataflow information that is invaluable to enable optimizations. Therefore, an unoptimized OpenISA binary forces the translated OpenISA program to also be unoptimized and it is no longer possible to recover the lost performance. These experiments assume that if the user is worried with performance, she works with optimized versions of the binaries and OpenISA is committed with preserving this performance during translation.

Chapter 8

Other exploratory work and contributions

The ISA interface is a specific layer of the software-hardware stack, but studying its changes requires extensive support of many tools ranging from simulators to compilers. This thesis is not limited to the ISA, but it also explores other levels and tools before supporting ISA changes.

This chapter presents other works developed in the context of this thesis, and while are not directly focused on OpenISA, they are important to evolve the state of the art in the areas of computer architecture and compilers, both areas involved in the OpenISA project. It finishes with a complete list of all literature published as a result of the investigations that led to this thesis.

8.1 The C language level

The portability problem may be addressed directly by programmers if, instead of writing code in a native machine language, they write code in a high-level language such as C. Programmers should be able to recompile their code to run on another hardware platform by switching compilers, as long as the compiler understands the same input language. In this way, the effort to reprogram an entire pool of software is reduced to that of *porting* the compiler. The task of porting software involves using a compiler that translates source code to the machine language of a different platform, by use of the original source code in a high-level language, and then testing and adapting the software to run on this new platform if this process introduced new bugs.

However, this classic approach constrains the creation of new ISAs since writing an efficient compiler that targets a new ISA is a challenging software engineering task. This is specially true for small research groups with limited access to programmers. Ceng et al. [47], Brandner et al. [42,56], Dias and Ramsey [54] and Ramsey and Dias [113] all have worked on easy ways to retarget a compiler. In order to do this, the usual approach is to start with a retargetable compiler project [18,79], which uses an independent intermediary representation (IR) to represent the result of the translation down to a certain level. Then, the retargeting effort is limited to write a component that translates the IR to machine

language, the backend. Since it is still challenging enough, these authors all proposed ways of automating the generation of compiler backends.

To solve the portability problem with automated compiler generation is quite challenging because the quality of the compiler is compromised. If the machine language translation of a program is poor, the program will suffer with poor performance, which means the new ISA will have an unfair evaluation. Old and deprecated ISAs, owing to its better-quality compilers, will still have an advantage over new technology, subverting innovation.

We address these issues by extending the ArchC [31] language, which allows a designer to easily describe a new ISA as well as the entire processor architecture. By building upon the work developed in my Master thesis [27], we finished the development of an ArchC compiler generator, which retargets the LLVM [86] compiler infrastructure. The difference of this compiler generator is that it is targeted at providing high-quality compilers instead of focusing solely in compiler retargeting. This motivated us to choose LLVM, which is already recognized as an industrial-strength compiler and adopted by many companies [90] including Apple, Adobe, Ageia, Electronic Arts, Intel and Nvidia. We published the results [30] in the Brazilian Symposium of Computer Architecture (SBAC-PAD 2012) conference, held in New York.

A fundamental issue with enhancing portability at the level of the C compiler is that C code still has many features that depend on the target machine. This means that the programmer still has to change the source code to work on other platforms, debug and check for correct behavior of the program when running on other machines. On the other hand, there are languages designed to be portable from scratch, whose goal is specific to run on several different machines. We will discuss this special category of software in the next section.

8.2 The browser level

As the computing world advanced to be internet-centric, the browser started to play a key role in the client software environment. By providing the capability to run small scripts, it created a powerful, easy to program virtual machine that runs in many platforms. The access to the web cannot be neglected and, as such, every device intended for consumer direct usage provides a browser with the capability to run JavaScript [59].

Owing to the capacity of running the same code on different machines, it is not surprising that many compilers target JavaScript as an intermediate language, such as the Google Web Toolkit [7] by Google, TypeScript [14] by Microsoft, Dart [11] by Google or the Emscripten [132] used in the LLVM [17, 86] community.

By using compilers that emit JavaScript, it is possible to provide a solution to the deployment problem by translating languages to JavaScript. We analyzed a compiler that translates a general purpose script language to assess how easy it is to port programs to JavaScript. This compiler is used in the TouchDevelop [76, 127, 128] platform that allows users to write cellphone apps that runs on any phone, Google Android [62], Apple iPhone or Windows Phone, by means of JavaScript.

However, the ability to run on many different environments also brings new challenges when it comes to ensure good performance of the scripts. Since clients have different browsers to choose from and each browser implements its own JavaScript engine (e.g. SpiderMonkey [9], V8 [16], JavaScriptCore (aka. Nitro) [15] or Chakra [13]), optimizing the JavaScript code becomes a guessing game because each engine has its own optimizations and limitations. For example, we show that applying JavaScript code optimizations in a tablet with Windows 8 and Internet Explorer 11 increased performance by, on average, 5 times, while running in a desktop with Windows 7 and Firefox decreased performance by 20%.

To overcome these problems, we developed a crowdsourced approach to drive the JavaScript compiler optimizations. We use a benchmark of scripts to exercise common performance bottlenecks and compile these scripts with different optimizations in different clients, storing the results of each client in the cloud. This enabled me to characterize how each system responds to the optimizations and this information gets uploaded to the cloud. When another user that uses the same platform compiles the script to JavaScript, the system queries the cloud to know the best set of flags, or optimizations to apply, that best suits her system. This work was published [29] in the International Conference of Compiler Construction (CC 2014).

8.3 List of published papers, contributions and acknowledgements

The list below shows all publications produced during the development of this PhD thesis.

- Rafael Auler, Carlos Eduardo Millani, Alexandre Brisighello, Alisson Linhares, Edson Borin. “Handing IoT platform heterogeneity with COISA, a compact OpenISA virtual platform”. Concurrency and Computation: Practice and Experience. (To appear)
- Carlos Eduardo Millani, Alisson Linhares, Rafael Auler, and Edson Borin. “COISA: A Compact OpenISA virtual platform for IoT devices”. In WSCAD 15, October 2015, Florianopolis, Brazil.
- Bruno Cardoso Lopes, Rafael Auler, Luiz Ramos, Edson Borin, and Rodolfo Azevedo. “SHRINK: Reducing the ISA Complexity via Instruction Recycling”. In The 42nd International Symposium on Computer Architecture (ISCA 42), June 2015, Portland, OR, USA.
- Rafael Auler, and Edson Borin. “OpenISA, freedom powered by efficient binary translation”. In The 8th Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT 2015, co-located with CGO, HPCA and PPoPP 2015), February 2015, San Francisco, USA.
- Gabriel Bertazi, Rafael Auler and Edson Borin. “Uma plataforma para o ensino de organização de computadores e linguagem de montagem”. International Journal of Computer Architecture Education, Vol. 3, p.13, 2014 (In Portuguese)

- Rafael Auler, Edson Borin, Peli de Halleux, Michal Moskal, and Nikolai Tillman. "Addressing JavaScript JIT engines performance quirks: A crowdsourced adaptive compiler". In The 23rd International Conference on Compiler Construction (CC 2014), April 2014, Grenoble, France.

- Divino Cesar, Rafael Auler, Rafael Dalibera, Sandro Rigo, Edson Borin, and Guido Araujo. "Modeling Virtual Machines Misprediction Overhead". In The 2013 IEEE International Symposium on Workload Characterization (IISWC 2013), September 2013, Portland, USA.

- Bruno Lopes, Rafael Auler, Rodolfo Azevedo and Edson Borin. "ISA Aging: A X86 Case Study". In Seventh Annual Workshop on the Interaction Amongst Virtualization, Operating Systems and Computer Architecture (WIVOSCA 2013), July 2013, Tel Aviv, Israel.

- Rafael Auler and Edson Borin. "A LLVM Just-in-Time Compilation Cost Analysis". Technical Report 13(2013) - Institute of Computing, University of Campinas, May 2013.

- Leonardo Piga, Gabriel F. T. Gomes, Rafael Auler, Bruno Rosa, Sandro Rigo, and Edson Borin. "Assessing Computer Performance with SToCS". In International Conference on Performance Engineering: ICPE 2013, April 2013, Prague, Czech Republic.

- Marcelo Guedes, Rafael Auler, Liana Duenha, Edson Borin, and Rodolfo Azevedo. "An Automatic Energy Consumption Characterization of Processors Using ArchC". Journal of Systems Architecture - Embedded Systems Design 59(8): 603-614(2013).

- Rafael Auler, Paulo Centoducatte, and Edson Borin. "ACCGen: An automatic ArchC compiler generator". In SBAC-PAD 12: The 24th International Symposium on Computer Architecture and High Performance Computing, October 2012, New York, USA.

- Marcelo Guedes, Rafael Auler, Edson Borin, and Rodolfo Azevedo. "An ArchC approach for automatic energy consumption characterization of processors". In The 23rd IEEE International Symposium on Rapid System Prototyping (RSP 2012), October 2012, Tampere, Finland.

- Divino Lucas, Rafael Dalibera, Rafael Auler, Guido Araujo, Sandro Rigo, and Edson Borin. "Hotness Misprediction Overhead in Virtual Machines". WISH 2012, San Jose, USA.

In the context of this thesis, the following patent application was also submitted:

- Bruno Cardoso Lopes, Rafael Auler, Edson Borin and Rodolfo Azevedo. Patent BR 10 2015 005838 (pending) for the mechanism described in the paper "SHRINK: Reducing the ISA Complexity via Instruction Recycling", ISCA 42.

And the following book published, helping students and LLVM enthusiasts to use the LLVM framework:

- Bruno Cardoso and Rafael Auler. Getting Started with LLVM Core Libraries, Aug. 26, 2014, Packt Publishing. Rated 4.7 out of 5 stars on Amazon.

We would like to thank the grant by FAPESP number 2011/09630-1 for the development of this PhD thesis. During the development of this PhD thesis, I was also a recipient of the Microsoft Research 2013 Graduate Research Fellowship Award and would like to thank Microsoft.

Chapter 9

Conclusion

This thesis presented the case for OpenISA, a hybrid ISA that aims to make the interface between the processor and its applications more flexible, bringing benefits to both hardware and software.

In the hardware aspect, OpenISA incorporates and extends the recycling mechanism [92], allowing processor implementations to realize only a subset of the full ISA while seamlessly supporting programs targeting any version of the ISA. The recycling mechanism decouples the ISA encoding from the instructions themselves, where any given ISA version maps encoding to instructions with an injective but non-surjective function. Subsequent ISA versions need not to map the same encodings to the same instructions, giving freedom to hardware developers to remove, change or add instructions to a new chip.

The recycling mechanism works by issuing traps when decoding instruction encodings for which hardware and software disagrees about its implementation. Since the hardware does not implement this instruction, its behavior is emulated in software by the exception handling function in kernel space.

OpenISA encoding formats were co-designed with its versioning and recycling mechanism. It is a fixed-length format that eases the implementation of a multi-issue decoding unit, but also allows larger instructions, with sizes multiples of 4 bytes, in case more information needs to be encoded in a single instruction. The versioning and recycling mechanism is, to the best of our knowledge, the first *hybrid* ISA, in which part of the specification may be emulated in software if needed. It also allows it to evolve while maintaining backwards and forwards compatibility at all times.

In the software aspect, OpenISA is designed to be easy to be emulated in other platforms, effectively acting as a program distribution format. Even though HLL frameworks such as Java can achieve good performance in this realm, it constrains programs. While binary translation translates any machine language program, it may struggle to achieve a good performance. The results presented here show that the problem lies in the fact that there are ISA features that are hard to emulate and that it is possible to overcome these problems and design an ISA that can be translated with high efficiency to other hosts.

This ISA began with a MIPS-based starting point. The experimental framework developed for this thesis was built to answer the question of whether it is possible to design an ISA that preserves enough semantic information to allow its translation to

other ISAs without losing performance. Experimental results showed that this is possible while testing OpenISA translation to two different host platforms, x86 and ARM.

It is important that the translation system is able to recover information about how data is accessed and build an accurate data-flow graph. Therefore, OpenISA design decisions are geared towards making data access straightforward or easy to be inferred. A further investigation is performed assessing three different guest register access methods in binary translation: accessing all registers as global variables, buffering some registers in local variables and translating the entire program at once, in a single region. A combination of changes in the guest ISA and the careful control of the translation region size or the register synchronization overhead leads to the best results.

OpenISA binaries tested in these experiments were already optimized. If a binary that is not optimized is employed, all values are kept on memory rather than on registers, harming the translator capacity of inferring an accurate dataflow graph. When values go to memory, it is more difficult to reconstruct dataflow information that is invaluable to enable optimizations because memory positions depend on an alias analysis.

This thesis focused on user-level programs and we leave as future work the efficient emulation of full systems. We also employed a static binary translation framework to show our results while factoring out other overheads faced in a dynamic binary translation virtual machine. Static binary translation cannot handle all classes of programs, specifically those with self-modifying code, although it can translate the majority of user-level programs and serves the purposes of this thesis.

Overall, this thesis presented an innovative, new interface for processors with focus on flexibility in both sides affected by the ISA: (1) flexibility on the hardware side, which is free to evolve without a hard commitment to past ISA decisions, tackling the ISA aging problem and (2) flexibility to the software, which is free to be translated to other ISAs, tackling the software deployment and portability problem.

Bibliography

[1] Ackermann's function. http://www-users.cs.york.ac.uk/%7Esusan/cyc/a/ackermnn.htm. Accessed: May 2016.

[2] Apple II Emulator. https://courses.cit.cornell.edu/ee476/FinalProjects/s2007/bcr22/\final%20webpage/final.html. Accessed: May 2016.

[3] AVR NESEMU. https://courses.cit.cornell.edu/ee476/FinalProjects/s2009/bhp7_teg25/bhp7_teg25/. Accessed: May 2016.

[4] Commodore VIC-20 AVR Emulator. http://www.belanger.pwp.blueyonder.co.uk/Projects/Vic%20Emu/vicemu.htm. Accessed: May 2016.

[5] Emulating a z80 computer. http://hackaday.com/2010/04/27/emulating-a-z80-computer-with-an-avr-chip/. Accessed: May 2016.

[6] GCC Front Ends. https://gcc.gnu.org/frontends.html. Accessed: May 2016.

[7] Google Web Toolkit Page. http://www.gwtproject.org/. Accessed: May 2016.

[8] Jikes RVM. http://www.jikesrvm.org. Accessed: May 2016.

[9] Mozilla SpiderMonkey JavaScript Engine. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. Accessed: May 2016.

[10] QEMU Tiny Code Generator (TCG). http://wiki.qemu.org/Documentation/TCG. Accessed: May 2016.

[11] The Dart Language Web Page. https://www.dartlang.org/. Accessed: May 2016.

[12] The Great Win32 Computer Language Shootout. http://dada.perl.it/shootout/. Accessed: May 2016.

[13] The New JavaScript Engine in Internet Explorer 9. http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx. Accessed: May 2016.

[14] The TypeScript Language Web Page. http://www.typescriptlang.org/. Accessed: May 2016.

[15] The WebKit Open Source Project. http://webkit.org/. Accessed: May 2016.

[16] V8 JavaScript Engine. `https://developers.google.com/v8/`. Accessed: May 2016.

[17] V Adve, C Lattner, M Brukman, A Shukla, and B Gaeke. LLVA: a low-level virtual instruction set architecture. In *MICRO36*, 2003.

[18] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools (2nd edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[19] Mehdi Alipour, Mostafa E. Salehi, and Hesamodin shojaei baghini. Design space exploration to find the optimum cache and register file size for embedded applications. In *The 2011 International Conference on Embedded Systems and Applications, Las Vegas, Nevada*, 2011.

[20] H. Peter Anvin, Alex Klaiber, Guillermo J. Rozas, and Parag Gupta. Method and apparatus for improving segmented memory addressing. U.S. Patent 6851040, 02 2005.

[21] Ehsan K Ardestani and Jose Renau. ESESC: A fast multicore simulator using time-based sampling. In *(HPCA 2013)*, pages 448–459. IEEE, 2013.

[22] ARM. *Cortex-A9 Technical Reference Manual, Revision r4p1, Appendix B "Instruction Cycle Timings", Section B.3 "Load and store instructions"*. ARM DDI 0388I, `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388e/Chdgjcci.html`, Accessed: May 2016.

[23] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA '00*, 2000.

[24] Krste Asanovic and David Patterson. The case for open instruction sets. Open ISA would enable free competition in processor design. In *Microprocessor Report*, 2014.

[25] R. Auler and E. Borin. OpenISA, freedom powered by efficient binary translation. In *Proceedings of the 8th AMAS-BT*, 2015.

[26] R. Auler, C. Millani, A. Brisighello, A. Linhares, and E. Borin. Handling IoT platform heterogeneity with COISA, a compact OpenISA virtual platform. In *Concurrency and Computation: Practice and Experience (to appear)*, 2016.

[27] Rafael Auler. ADL-Based Automatic Backend Generation. Master's thesis, In Portuguese, Institute of Computing, University of Campinas, Brazil, September 2011.

[28] Rafael Auler and Edson Borin. A LLVM Just-in-Time Compilation Cost Analysis. Technical Report 13-2013 IC-UNICAMP, May 2013.

[29] Rafael Auler, Edson Borin, Peli de Halleux, Michal Moskal, and Nikolai Tillmann. Addressing JavaScript JIT engines performance quirks: A crowdsourced adaptive compiler. In *CC 2014*.

[30] Rafael Auler, Paulo C. Centoducatte, and Edson Borin. ACCGen: An automatic ArchC compiler generator. In *SBAC-PAD '2012*.

[31] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, October 2005.

[32] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. PLDI'00, 2000.

[33] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *MICRO36*. IEEE Computer Society, 2003.

[34] Marcus Bartholomeu. *Compiled simulation for computer architectures described with ArchC*. PhD thesis, University of Campinas, 2005.

[35] Daniel Bartholomew. QEMU a Multihost Multitarget Emulator. *Linux Journal*, 2006(145):3, 2006.

[36] J. F. Bastien, Luke Wagner, Michael Holman, Seth Thompson, and Raphael Isemann. Development of webassembly and associated infrastructure. `https://github.com/webassembly`. Accessed: May 2016.

[37] Richard Belgard. Speculative address translation for processor using segmentation and optional paging. U.S. Patent 6813699, 11 2004.

[38] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[39] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. *ACM Trans. Comput. Syst.*, 33(1):3:1–3:34, March 2015.

[40] Carl Boettiger. An introduction to Docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, January 2015.

[41] E. Borin and Youfeng Wu. Characterization of DBT overhead. IISWC'09, 2009.

[42] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *CASES '07*, New York, NY, USA, 2007. ACM.

[43] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO 2003*. IEEE.

[44] Derek Lane Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[45] Bruno Cardoso and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[46] John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *OOPSLA '06*, 2006.

[47] Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. C compiler retargeting based on instruction semantics models. In *DATE '05*, 2005.

[48] Divino Cesar, Rafael Auler, Rafael Dalibera, Sandro Rigo, Edson Borin, and Guido Araujo. Modeling virtual machines misprediction overhead. In *IISWC 2013*.

[49] Wei Chen, Dan Chen, and Zhiying Wang. An approach to minimizing the interpretation overhead in dynamic binary translation. *The Journal of Supercomputing*, 61(3), 2012.

[50] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: Adaptive Compilation Made Efficient. In *LCTES '05*, 2005.

[51] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO 2003*.

[52] Joseph D'Errico and Wei Qin. Constructing portable compiled instruction-set simulators: An ADL-driven approach. In *DATE '06*, 2006.

[53] Stephen L Diamond and Gianluigi Castelli. Architecture Neutral Distribution Format (ANDF). *IEEE Micro*, 14(6):73–76, 1994.

[54] João Dias and Norman Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *POPL '10*, 2010.

[55] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In *ASPLOS IX*, 2000.

[56] Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. Effective compiler generation by architecture description. In *LCTES '06*, New York, NY, USA, 2006. ACM.

[57] A. Fauth and A. Knoll. Automated generation of DSP program development tools using a machine description formalism. In *ICASSP-93*, 1993.

[58] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings of the EDTC: The European Design and Test Conference*, pages 503–507. IEEE Computer Society, March 1995.

[59] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., 2002.

[60] Agner Fog. Proposal for an ideal extensible instruction set. `http://agner.org/optimize/blog/read.php?i=523#523`. Accessed: May 2016.

[61] Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[62] Nisarg Gandhewar and Rahila Sheikh. Google android: An emerging software platform for mobile devices. *International Journal on Computer Science & Engineering*, 2011.

[63] M. Garcia, R. Azevedo, and S. Rigo. Optimizing simulation in multiprocessor platforms using dynamic-compiled simulation. In *WSCAD-SSC 2012*.

[64] Maxiwell Salvador Garcia, Rodolfo Azevedo, and Sandro Rigo. Optimizing a retargetable compiled simulator to achieve near-native performance. In *WSCAD-SCC 2010*. IEEE.

[65] Google. Welcome to native client. `https://developer.chrome.com/native-client`. Accessed: May 2016.

[66] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[67] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: an instruction set description language for retargetability. In *DAC 34*. ACM Press, 1997.

[68] Masayo Haneda, Peter MW Knijnenburg, and Harry AG Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *PACT 2005*. IEEE.

[69] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[70] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[71] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO '07*, 2007.

[72] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture exploration for embedded processors with LISA*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[73] Greg Hoglund and Gary McGraw. *Exploiting software: How to break code*. Pearson Education India, 2004.

[74] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *CGO '12*. ACM, 2012.

[75] Raymond J. Hookway and Mark A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1), 1997.

[76] Nigel Horspool and Nikolai Tillmann. *TouchDevelop: Programming on the Go*. Apress, 2013.

[77] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew, and Wei-Chung Hsu. Lnq: Building high performance dynamic binary translators with existing compiler backends. In *ICPP 2011*. IEEE.

[78] Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu, Pen-Chung Yew, Ding-Yong Hong, Wei-Chung Hsu, and Chien-Min Wang. Improving dynamic binary optimization through early-exit guided code region formation. In *VEE '13*, 2013.

[79] Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. DisIRer: Converting a retargetable compiler into a multiplatform binary translator. *ACM Trans. Archit. Code Optim.*, 7(4):18:1–18:36, 2010.

[80] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, et al. The superblock: an effective technique for VLIW and superscalar compilation. *the Journal of Supercomputing*, 7(1-2):229–248, 1993.

[81] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. A trace-based java JIT compiler retrofitted from a method-based compiler. In *CGO '11*, 2011.

[82] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[83] Marco Kaufmann, Matthias Häsing, Thomas Preusser, and Rainer Spallek. The java virtual machine in retargetable, high-performance instruction set simulation. In *PPPJ '11*. ACM, 2011.

[84] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *PLDI '01*. ACM, 2001.

[85] DE Knuth. The art of computer programming, vol. III: Sorting and searching. 1973.

[86] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*. IEEE Computer Society, 2004.

[87] Kevin P Lawton. Bochs: A portable PC emulator for UNIX/X. *Linux Journal*, 1996(29es):7, 1996.

[88] John R. Levine. *Linkers and Loaders (The Morgan Kaufmann Series in Software Engineering and Programming)*. Morgan Kaufmann, January 2000.

[89] Imperas Software Limited. Open Virtual Platforms website. `http://www.ovpworld.org`. Accessed: May 2016.

[90] LLVM-community. List of LLVM Users. `http://www.llvm.org/Users.html`. Accessed: May 2016.

[91] LLVMdev. LLVM IR is a compiler IR. `http://lists.llvm.org/pipermail/llvm-dev/2011-October/043724.html`. Accessed: May 2016.

[92] B. C. Lopes, R. Auler, L. Ramos, E. Borin, and R. Azevedo. SHRINK: Reducing the ISA complexity via instruction recycling. ISCA 42. ACM, 2015.

[93] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, 2005.

[94] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[95] Carlos Millani, Alisson Linhares, Rafael Auler, and Edson Borin. COISA: A compact OpenISA virtual platform for IoT devices. In *Proceedings of WSCAD'15*, 2015.

[96] Prabhat Mishra and Nikil Dutt, editors. *Processor Description Languages*. Morgan Kaufmann Publishers Inc., Volume I. San Francisco, CA, USA, 2008.

[97] Joao Batista Correia Gomes Moreira, Divino Lucas, Guido Araujo, Edson Borin, and Sandro Rigo. Asynchronous program flow verification through binary instrumentation in QEMU. In *AMASBT 2012*. ACM.

[98] Mozilla. Asm.js working draft. `http://asmjs.org/spec/latest/`. Accessed: May 2016.

[99] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07*. ACM, 2007.

[100] Guilherme Ottoni, Thomas Hartin, Christopher Weaver, Jason Brandt, Belliappa Kuttanna, and Hong Wang. Harmonia: a transparent, efficient, and harmonious dynamic binary translator targeting the intel architecture. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, 2011.

[101] John Owens. GPU architecture overview. In *ACM SIGGRAPH*, volume 1, pages 5–9, 2007.

[102] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot server compiler. JVM'01, 2001.

[103] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargeting of compiled simulators for digital signal processors using a machine description language. In *DATE '00*. ACM, 2000.

[104] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *DAC 36*. ACM Press, 1999.

[105] Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, page 335, 2010.

[106] Jason A Poovey, Markus Levy, Shay Gal-On, and T Conte. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29, September 2009.

[107] Zdeněk Přikryl, Jakub Křoustek, Tomáš Hruška, and Dušan Kolář. Fast translated simulation of ASIPs. In *6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, Brno, CZ, MUNI*, pages 135–142, 2010.

[108] ZdenËĞek Prikryl. Advanced methods of microprocessor simulation. *Information Sciences and Technologies, Bulletin of the ACM Slovakia*, 3(3):1–13, 2011.

[109] Mark Probst. Dynamic binary translation. In *UKUUG Linux Developer's Conference*, 2002.

[110] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. MICRO'39, 2006.

[111] Wei Qin and Sharad Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *DATE 2003*. IEEE.

[112] R. Ramaswamy and T. Wolf. PacketBench: a tool for workload characterization of network processing. In *Workload Characterization, 2003. WWC-6. 2003 IEEE International Workshop on*, pages 42–50, Oct 2003.

[113] Norman Ramsey and João Dias. Resourceable, retargetable, modular instruction selection using a machine-independent, type-based tiling of low-level intermediate code. In *POPL '11*. ACM, 2011.

[114] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(3):20, 2009.

[115] Sandro Rigo, Rodolfo Azevedo, and Luiz Santos. *Electronic System Level Design: An Open-Source Approach*. Springer, 2011.

[116] Phil Rogers. Heterogeneous system architecture overview. In *Hot Chips*, 2013.

[117] Tiark Rompf, Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. Surgical precision JIT compilers. In *PLDI'14*, 2014.

[118] Guillermo Rozas, David Dunn, David Dobrikin, Alex Klaiber, and Daniel H. Nelsen. Method and apparatus for emulating a floating point stack in a translation process. U.S. Patent 6725361, 03 2004.

[119] Harsh Sharangpani and H Arora. Itanium processor microarchitecture. *Micro, IEEE*, 20(5):24–43, 2000.

[120] Bor-Yeh Shen, Wei-Chung Hsu, and Wuu Yang. A retargetable static binary translator for the ARM architecture. *ACM Trans. Archit. Code Optim.*, 11(2):18:1–18:25, June 2014.

[121] Bor-Yeh Shen, Wei-Chung Hsu, and Wuu Yang. A retargetable static binary translator for the arm architecture. *ACM Trans. Archit. Code Optim.*, 11(2):18:1–18:25, June 2014.

[122] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[123] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, February 1993.

[124] J.E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005.

[125] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for a Java Just-in-time Compiler. In *PLDI '03*. ACM, 2003.

[126] Gregory T Sullivan, Derek L Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 50–57. ACM.

[127] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *ONWARD 2011*. ACM.

[128] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, Manuel Fahndrich, and Sebastian Burckhardt. Touchdevelop: app development on mobile devices. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 39. ACM, 2012.

[129] Harry Wagstaff, Miles Gould, Björn Franke, and Nigel Topham. Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *50th DAC*, New York, NY, USA, 2013. ACM.

[130] Cheng Wang, Shiliang Hu, Ho-Seop Kim, Sreekumar R Nair, Mauricio Breternitz Jr, Zhiwei Ying, and Youfeng Wu. StarDBT: An efficient multi-platform dynamic binary translation system. In *Advances in Computer Systems Architecture*, pages 4–15. Springer, 2007.

[131] Zhonglei Wang and Jörg Henkel. HyCoS: Hybrid compiled simulation of embedded software with target dependent code. In *CODES+ISSS '12*. ACM, 2012.

[132] Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *SPLASH 2011*. ACM.

[133] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. Hermes: A fast cross-ISA binary translator with post-optimization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 246–256, Washington, DC, USA, 2015. IEEE Computer Society.

[134] Rafael Zinsly. Técnicas de formação de regiões para projetos de máquinas virtuais eficientes. Master's thesis, In Portuguese, Institute of Computing, University of Campinas, Brazil, October 2013.

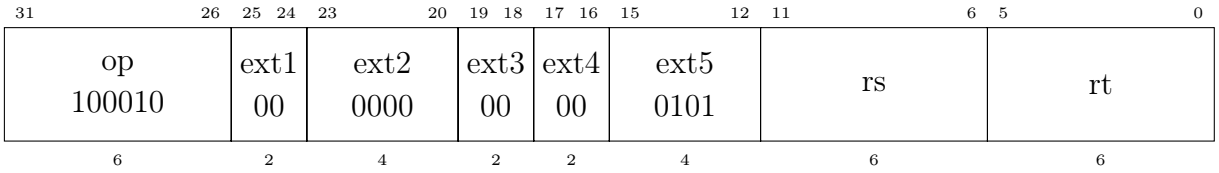
Appendix A

OpenISA instruction set reference

This appendix presents the ArchC-based, automatically generated OpenISA instruction set reference. The central repository for information about OpenISA is its ArchC model and the corresponding architecture description files. The next pages are the result of our ISA reference generator available on github[1].

[1] Available at http://github.com/rafaelauler/ISARefGenerator

A.1 abs.d



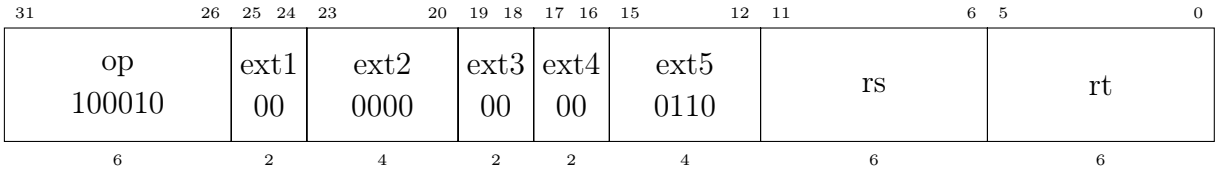
Encoding Format: PL12

Assembly Language Syntax: `abs.d %freg, %freg`

Operation (C Code):

```
double res = fabs(load_double(rt));
save_double(res, rs);
```

A.2 abs.s



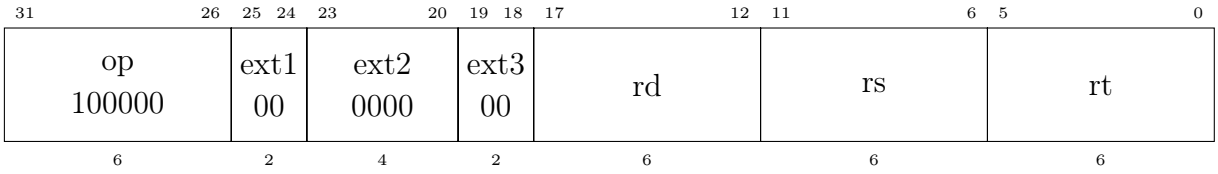
Encoding Format: PL12

Assembly Language Syntax: `abs.s %freg, %freg`

Operation (C Code):

```
float res = fabsf(load_float(rt));
save_float(res, rs);
```

A.3 add



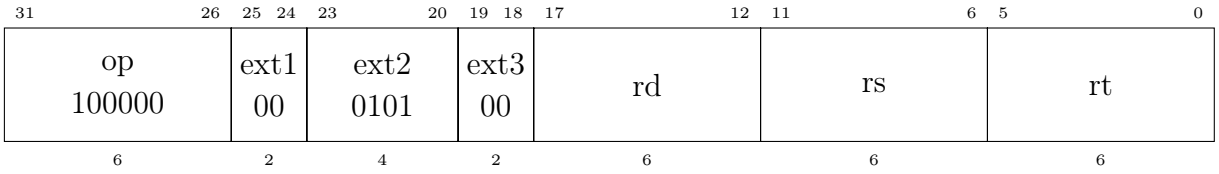
Encoding Format: PL18

Assembly Language Syntax: `add %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = RB[rs] + RB[rt];
```

A.4 add.d



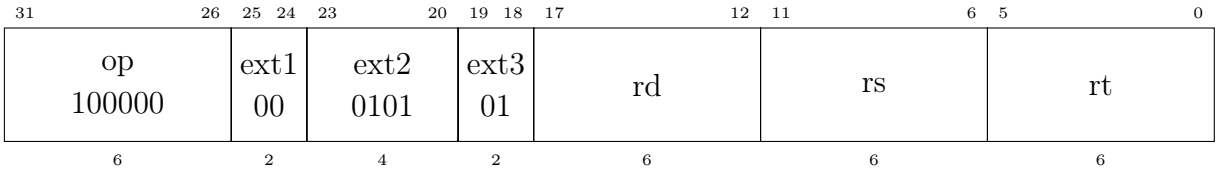
Encoding Format: PL18

Assembly Language Syntax: `add.d %freg, %freg, %freg`

Operation (C Code):

```
double res = load_double(rs) + load_double(rt);
save_double(res, rd);
```

A.5 add.s



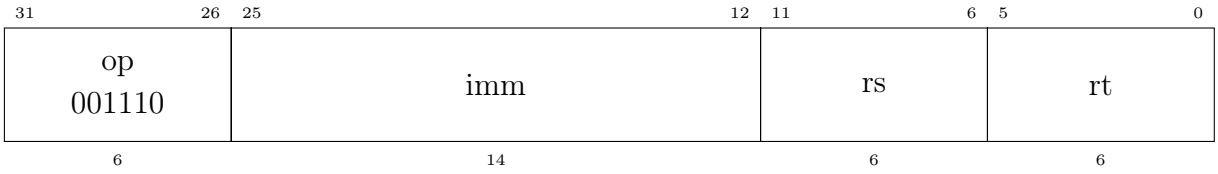
Encoding Format: PL18

Assembly Language Syntax: `add.s %freg, %freg, %freg`

Operation (C Code):

```
float res = load_float(rs) + load_float(rt);
save_float(res, rd);
```

A.6 addi



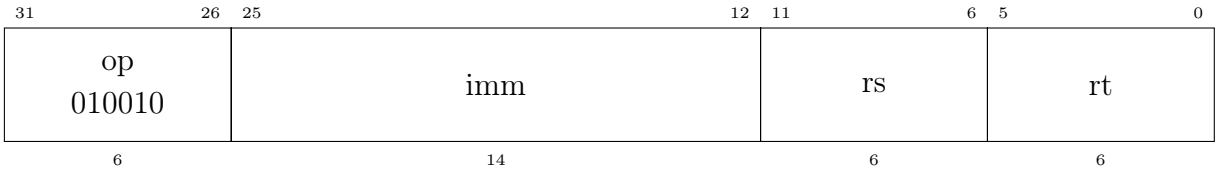
Encoding Format: PL26i

Assembly Language Syntax: `addi %reg, %reg, %exp`

Operation (C Code):

```
RB[rt] = RB[rs] + imm;
```

A.7 andi



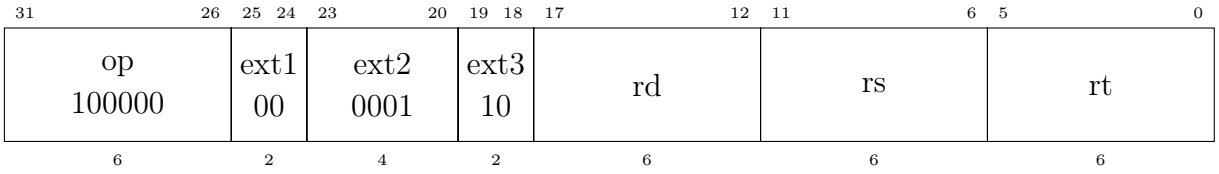
Encoding Format: PL26i

Assembly Language Syntax: `andi %reg, %reg, %imm`

Operation (C Code):

```
RB[rt] = RB[rs] & (imm & 0x3FFF);
```

A.8 and



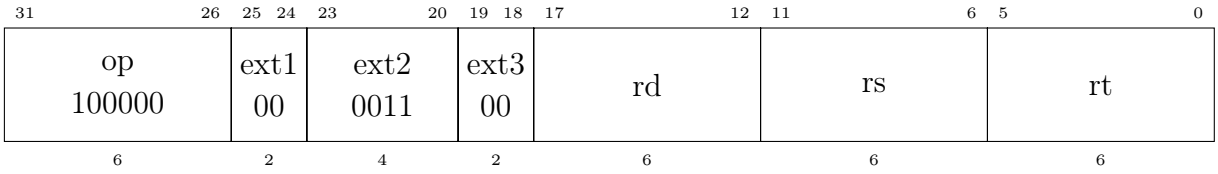
Encoding Format: PL18

Assembly Language Syntax: `and %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = RB[rs] & RB[rt];
```

A.9 asr



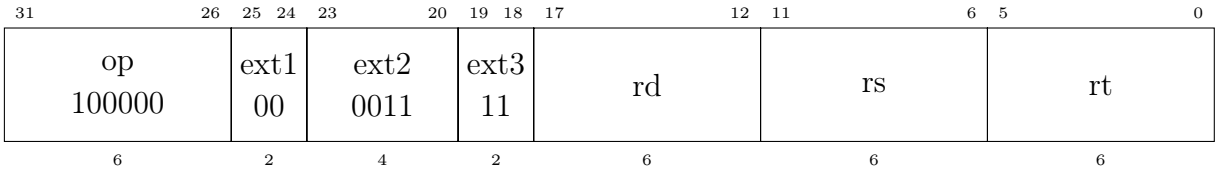
Encoding Format: PL18

Assembly Language Syntax: `asr %reg, %reg, %imm`

Operation (C Code):

```
RB[rd] = (ac_Sword)RB[rt] >> rs;
```

A.10 asrr



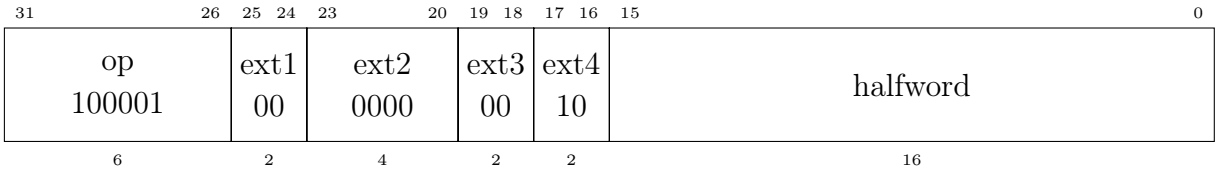
Encoding Format: PL18

Assembly Language Syntax: `asrr %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = (ac_Sword)RB[rt] >> (RB[rs] & 0x1F);
```

A.11 bc1f



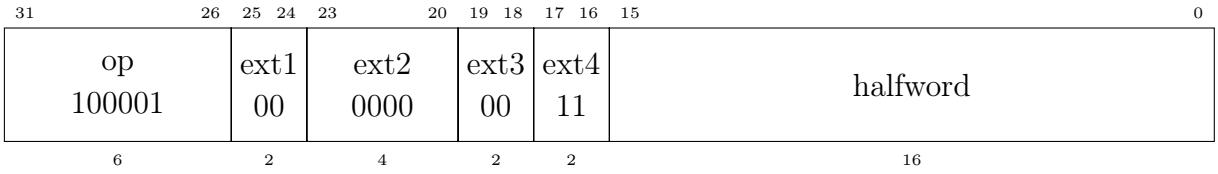
Encoding Format: PL16

Assembly Language Syntax: `bc1f %exp(pcrel)`

Operation (C Code):

```
if (cc == 0) {
  ac_pc = ac_pc + (halfword << 2);
}
```

A.12 bc1fl



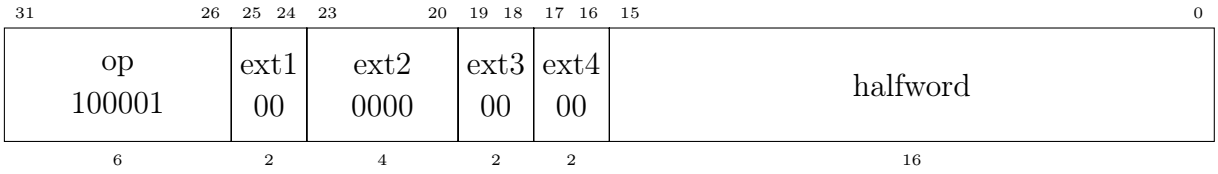
Encoding Format: PL16

Assembly Language Syntax: `bc1fl %exp(pcrel)`

Operation (C Code):

```
if (cc == 0) {
  ac_pc = ac_pc + (halfword << 2);
}
```

A.13 bc1t



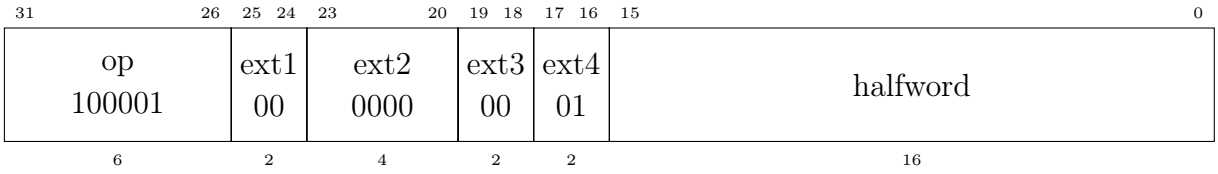
Encoding Format: PL16

Assembly Language Syntax: `bc1t %exp(pcrel)`

Operation (C Code):

```
if (cc == 1) {
  ac_pc = ac_pc + (halfword << 2);
}
```

A.14 bc1t



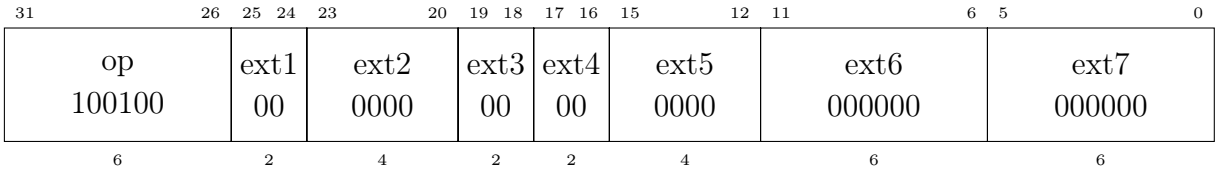
Encoding Format: PL16

Assembly Language Syntax: `bc1t %exp(pcrel)`

Operation (C Code):

```
if (cc == 1) {
  ac_pc = ac_pc + (halfword << 2);
}
```

A.15 break



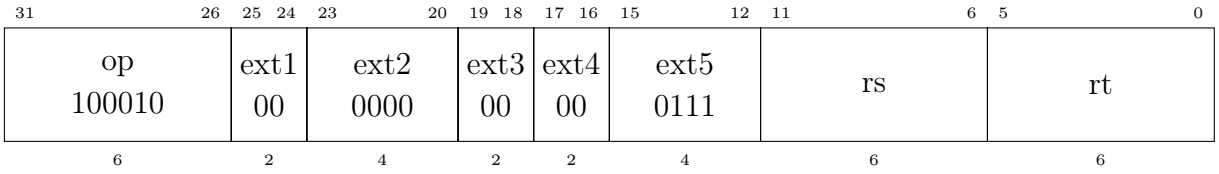
Encoding Format: PL0

Assembly Language Syntax: `break`

Operation (C Code):

```
fprintf(stderr, "instr_break behavior not implemented.\n");
exit(EXIT_FAILURE);
```

A.16 c.eq.d



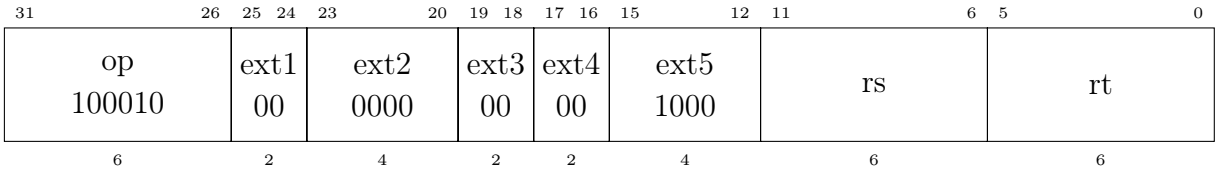
Encoding Format: PL12

Assembly Language Syntax: `c.eq.d %freg, %freg`

Operation (C Code):

```
double a = load_double(rs);
double b = load_double(rt);
cc = a == b ? (custom_isnan(a) || custom_isnan(b) ? 0 : 1) : 0;
```

A.17 c.eq.s



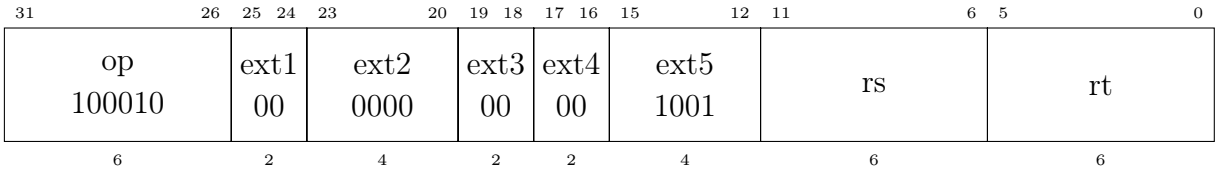
Encoding Format: PL12

Assembly Language Syntax: `c.eq.s %freg, %freg`

Operation (C Code):

```
float a = load_float(rs);
float b = load_float(rt);
cc = a == b ? (custom_isnanf(a) || custom_isnanf(b) ? 0 : 1) : 0;
```

A.18 c.ole.d



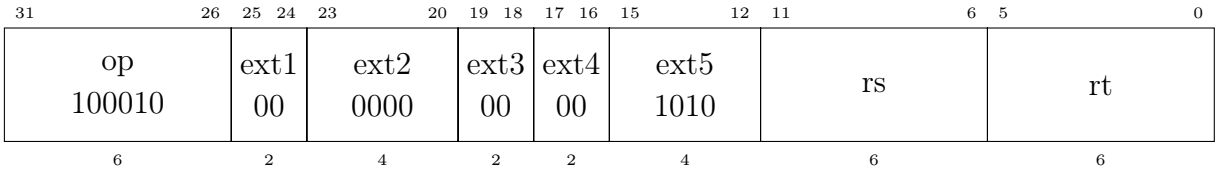
Encoding Format: PL12

Assembly Language Syntax: `c.ole.d %freg, %freg`

Operation (C Code):

```
double a = load_double(rs);
double b = load_double(rt);
cc = a <= b ? (custom_isnan(a) || custom_isnan(b) ? 0 : 1) : 0;
```

A.19 c.ole.s



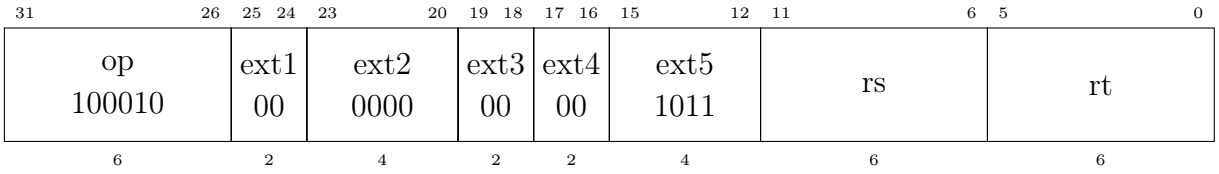
Encoding Format: PL12

Assembly Language Syntax: `c.ole.s %freg, %freg`

Operation (C Code):

```
float a = load_float(rs);
float b = load_float(rt);
cc = a <= b ? (custom_isnanf(a) || custom_isnanf(b) ? 0 : 1) : 0;
```

A.20 c.olt.d



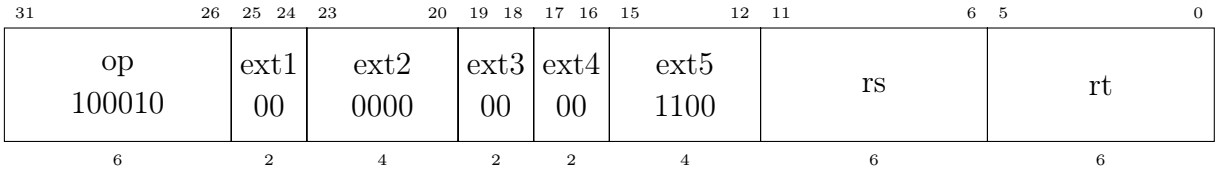
Encoding Format: PL12

Assembly Language Syntax: `c.olt.d %freg, %freg`

Operation (C Code):

```
double a = load_double(rs);
double b = load_double(rt);
cc = a < b ? (custom_isnan(a) || custom_isnan(b) ? 0 : 1) : 0;
```

A.21 c.olt.s



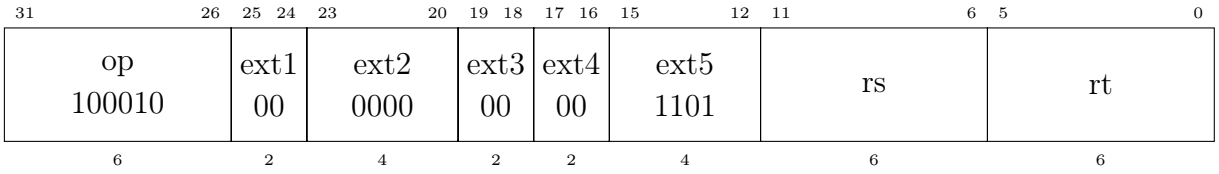
Encoding Format: PL12

Assembly Language Syntax: `c.olt.s %freg, %freg`

Operation (C Code):

```
float a = load_float(rs);
float b = load_float(rt);
cc = a < b ? (custom_isnanf(a) || custom_isnanf(b) ? 0 : 1) : 0;
```

A.22 c.ueq.d



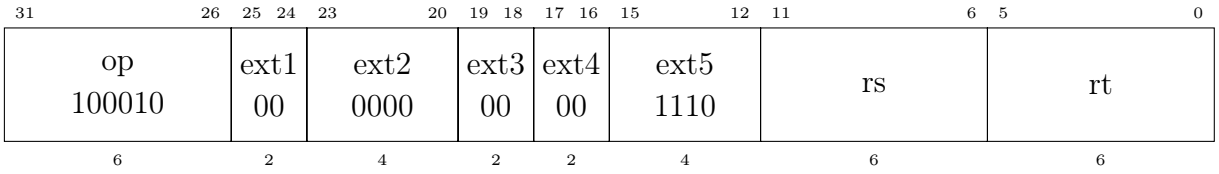
Encoding Format: PL12

Assembly Language Syntax: `c.ueq.d %freg, %freg`

Operation (C Code):

```
cc = (load_double(rs) == load_double(rt)) ? 1 : 0;
```

A.23 c.ueq.s



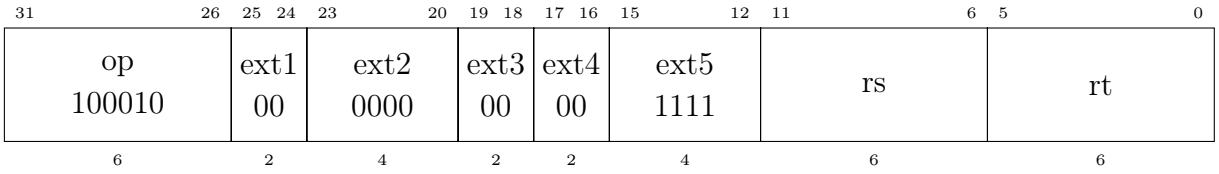
Encoding Format: PL12

Assembly Language Syntax: `c.ueq.s %freg, %freg`

Operation (C Code):

```
cc = (load_float(rs) == load_float(rt)) ? 1 : 0;
```

A.24 c.ule.d



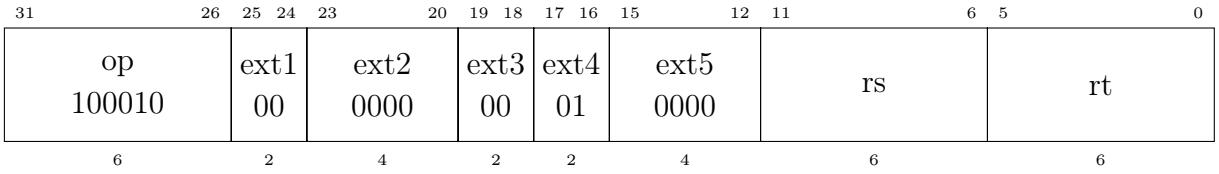
Encoding Format: PL12

Assembly Language Syntax: `c.ule.d %freg, %freg`

Operation (C Code):

```
cc = (load_double(rs) <= load_double(rt)) ? 1 : 0;
```

A.25 c.ule.s



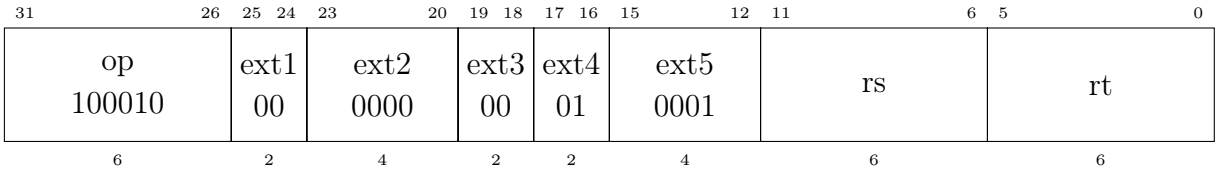
Encoding Format: PL12

Assembly Language Syntax: `c.ule.s %freg, %freg`

Operation (C Code):

```
cc = (load_float(rs) <= load_float(rt)) ? 1 : 0;
```

A.26 c.ult.d



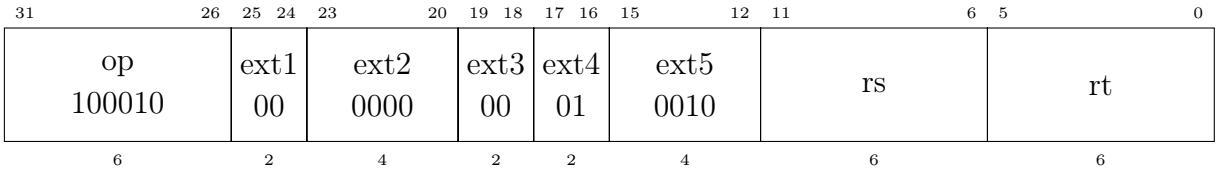
Encoding Format: PL12

Assembly Language Syntax: `c.ult.d %freg, %freg`

Operation (C Code):

```
cc = (load_double(rs) < load_double(rt)) ? 1 : 0;
```

A.27 c.ult.s



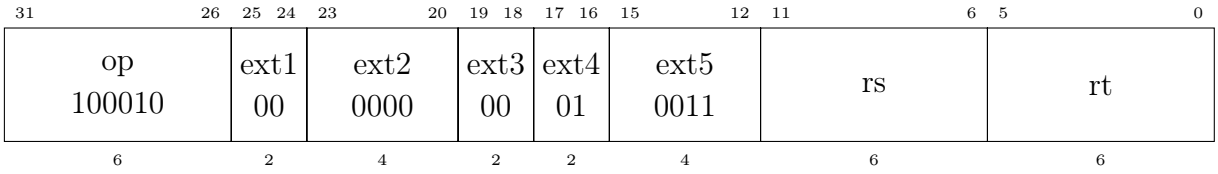
Encoding Format: PL12

Assembly Language Syntax: `c.ult.s %freg, %freg`

Operation (C Code):

```
cc = (load_float(rs) < load_float(rt)) ? 1 : 0;
```

A.28 c.un.d



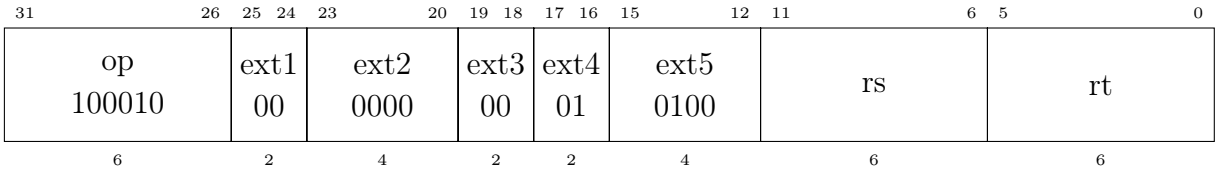
Encoding Format: PL12

Assembly Language Syntax: `c.un.d %freg, %freg`

Operation (C Code):

```
cc = (custom_isnan(load_double(rs))
   || custom_isnan(load_double(rt))) ? 1 : 0;
```

A.29 c.un.s



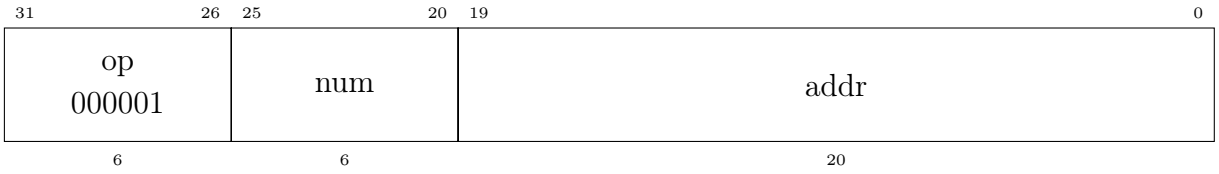
Encoding Format: PL12

Assembly Language Syntax: `c.un.s %freg, %freg`

Operation (C Code):

```
cc = (custom_isnanf(load_float(rs))
   || custom_isnanf(load_float(rt))) ? 1 : 0;
```

A.30 call



Encoding Format: PL26c

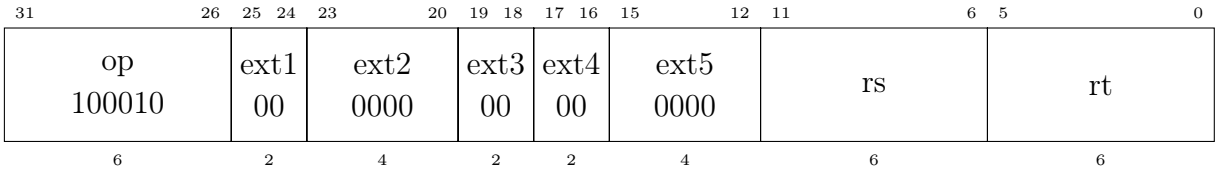
Assembly Language Syntax: `call %exp(align), %imm`

Operation (C Code):

```
RB[Ra] = ac_pc;

addr = addr << 2;
ac_pc = (ac_pc & 0xF0000000) | addr;
```

A.31 callr



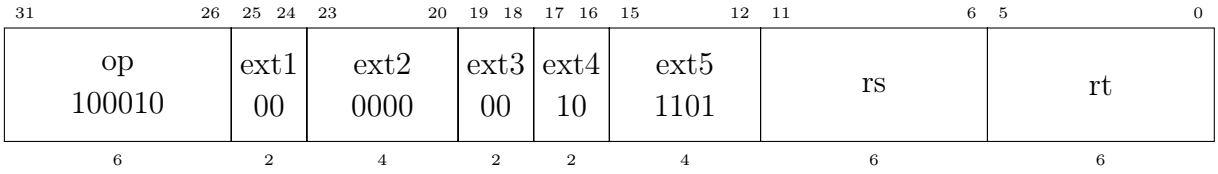
Encoding Format: PL12

Assembly Language Syntax: `callr %reg, %imm`

Operation (C Code):

```
RB[Ra] = ac_pc;
ac_pc = RB[rt];
```

A.32 ceil.w.d



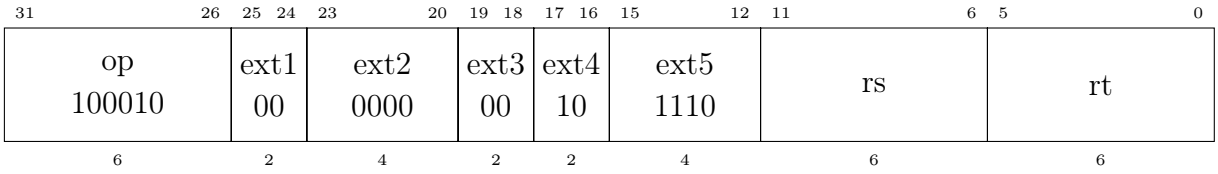
Encoding Format: PL12

Assembly Language Syntax: `ceil.w.d %reg, %freg`

Operation (C Code):

```
int32_t res = (int) ceil(load_double(rt));
RB[rt] = res;
```

A.33 ceil.w.s



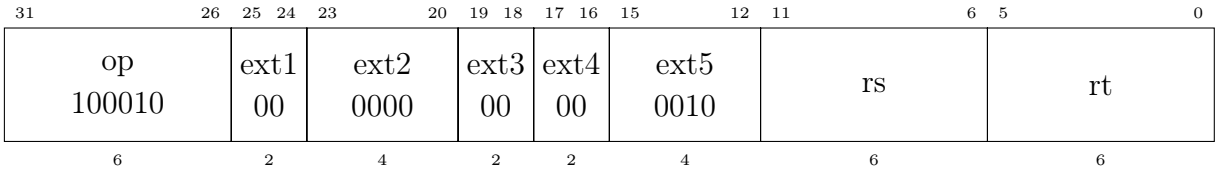
Encoding Format: PL12

Assembly Language Syntax: `ceil.w.s %reg, %freg`

Operation (C Code):

```
int32_t res = (int) ceilf(load_float(rt));
RB[rt] = res;
```

A.34 clz



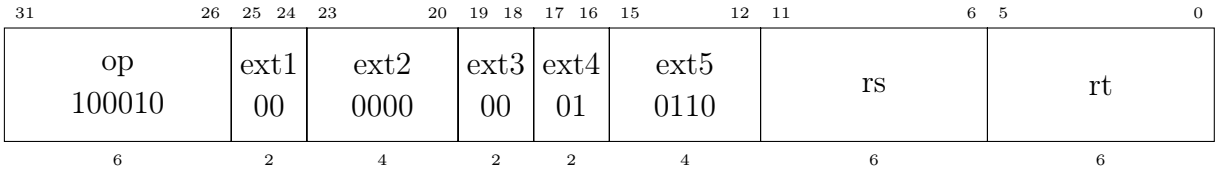
Encoding Format: PL12

Assembly Language Syntax: `clz %reg, %reg`

Operation (C Code):

```
uint32_t x = RB[rt];
x |= x >> 1;
x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;
x = ffs(x + 1);
if (x != 0) {
  x = 32 - x + 1;
}
RB[rs] = x;
```

A.35 cvt.d.s



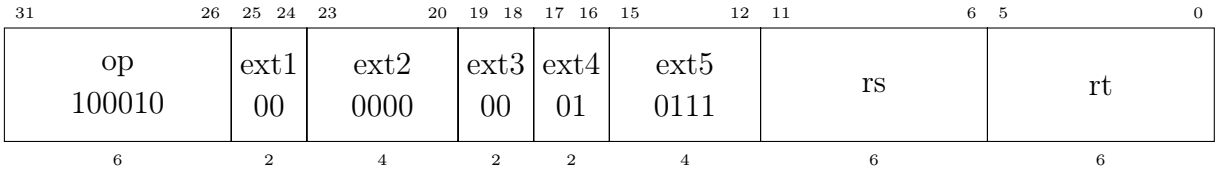
Encoding Format: PL12

Assembly Language Syntax: `cvt.d.s %freg, %freg`

Operation (C Code):

```
double temp = (double)load_float(rt);
save_double(temp, rs);
```

A.36 cvt.d.w



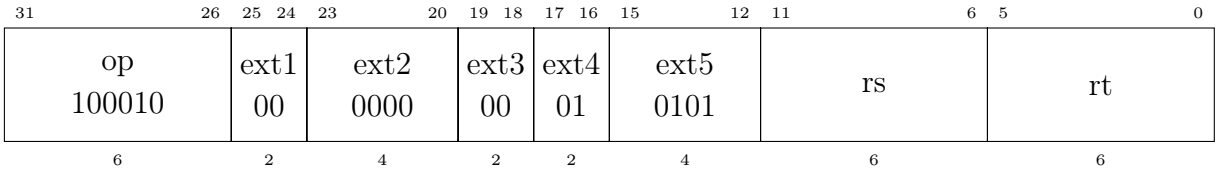
Encoding Format: PL12

Assembly Language Syntax: `cvt.d.w %freg, %freg`

Operation (C Code):

```
double temp = (double)(int)RBS[rt];
save_double(temp, rs);
```

A.37 cvt.s.d



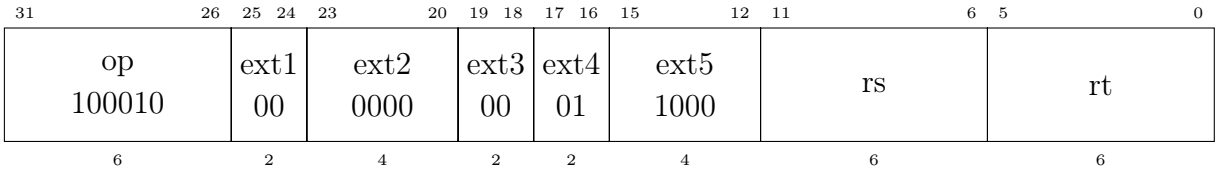
Encoding Format: PL12

Assembly Language Syntax: `cvt.s.d %freg, %freg`

Operation (C Code):

```
float temp = (float)load_double(rt);
save_float(temp, rs);
```

A.38 cvt.s.w



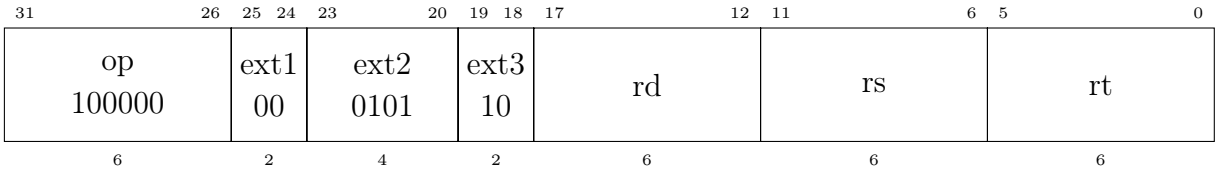
Encoding Format: PL12

Assembly Language Syntax: `cvt.s.w %freg, %freg`

Operation (C Code):

```
float temp = (float)(int)RBS[rt];
save_float(temp, rs);
```

A.39 div.d



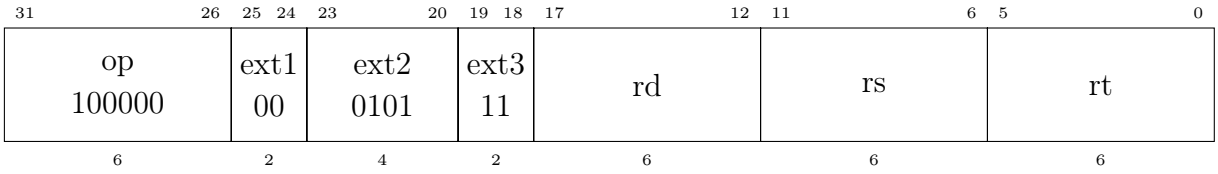
Encoding Format: PL18

Assembly Language Syntax: `div.d %freg, %freg, %freg`

Operation (C Code):

```
double res = load_double(rs) / load_double(rt);
save_double(res, rd);
```

A.40 div.s



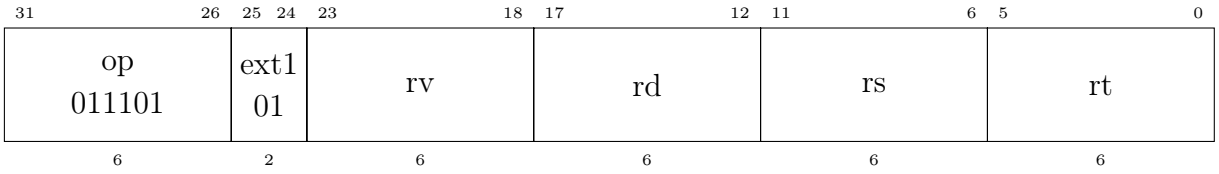
Encoding Format: PL18

Assembly Language Syntax: `div.s %freg, %freg, %freg`

Operation (C Code):

```
float res = load_float(rs) / load_float(rt);
save_float(res, rd);
```

A.41 div



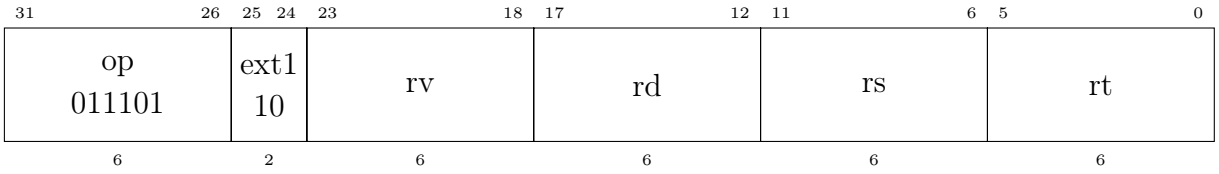
Encoding Format: PL24

Assembly Language Syntax: `div %reg, %reg, %reg, %reg`

Operation (C Code):

```
if (rd != 0)
  RB[rd] = (ac_Sword)RB[rs] / (ac_Sword)RB[rt];
if (rv != 0)
  RB[rv] = (ac_Sword)RB[rs] % (ac_Sword)RB[rt];
```

A.42 divu



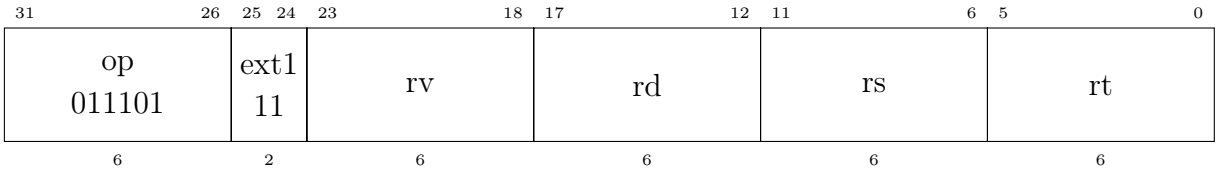
Encoding Format: PL24

Assembly Language Syntax: `divu %reg, %reg, %reg, %reg`

Operation (C Code):

```
if (rd != 0)
  RB[rd] = RB[rs] / RB[rt];
if (rv != 0)
  RB[rv] = RB[rs] % RB[rt];
```

A.43 ext



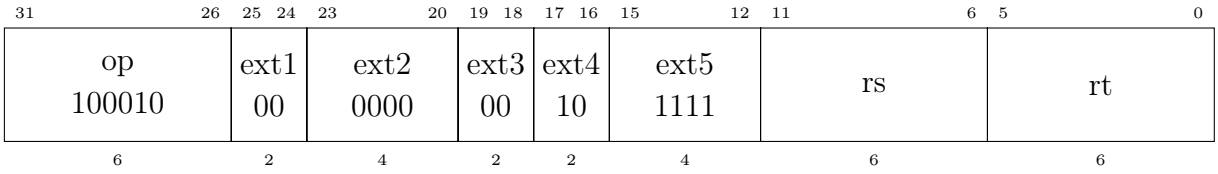
Encoding Format: PL24

Assembly Language Syntax: `ext %reg, %reg, %imm, %imm`

Operation (C Code):

```
uint32_t lsb = rv;
uint32_t size = rt + 1;
RB[rd] = (RB[rs] << (32 - size - lsb)) >> (32 - size);
```

A.44 floor.w.d



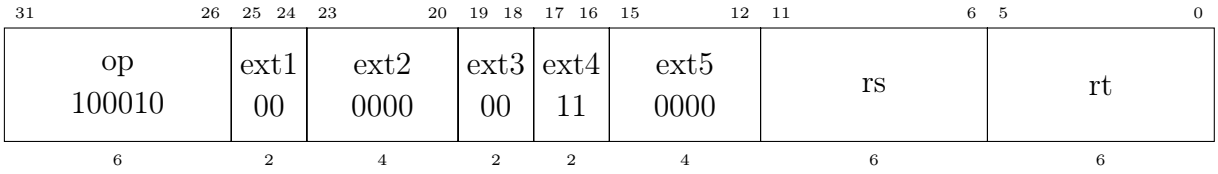
Encoding Format: PL12

Assembly Language Syntax: `floor.w.d %reg, %freg`

Operation (C Code):

```
int32_t res = (int) floor(load_double(rt));
RB[rt] = res;
```

A.45 floor.w.s



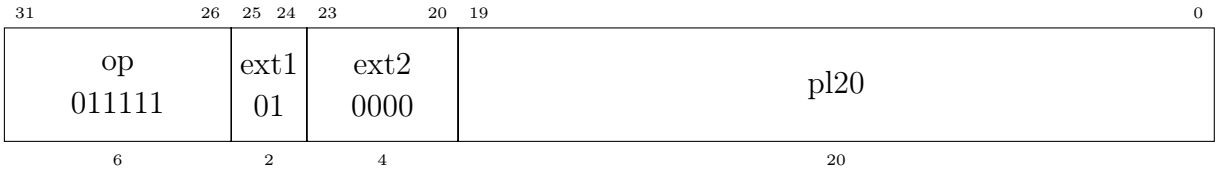
Encoding Format: PL12

Assembly Language Syntax: `floor.w.s %reg, %freg`

Operation (C Code):

```
int32_t res = (int) floorf(load_float(rt));
RB[rt] = res;
```

A.46 ijmphi



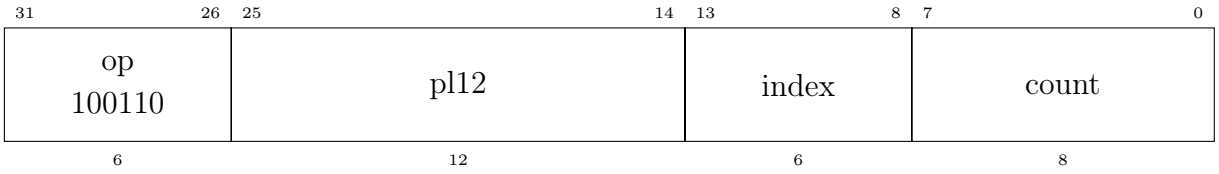
Encoding Format: PL20i

Assembly Language Syntax: `ijmphi %imm`

Operation (C Code):

```
ijmpreg = 0;
ijmpreg |= pl20 << 12;
```

A.47 ijmp



Encoding Format: PL26ij

Assembly Language Syntax: `ijmp %imm (%reg), %imm`

Operation (C Code):

```
ijmpreg &= 0xFFFFF000;
ijmpreg |= pl12 & 0xFFF;
uint32_t Target = DATA_PORT->read(ijmpreg + RB[index]);
ac_pc = Target;
```

A.48 jeq

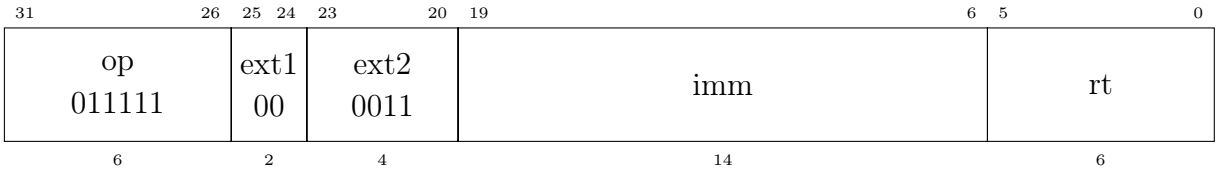
Encoding Format: PL26i

Assembly Language Syntax: `jeq %reg, %reg, %exp(pcrel)`

Operation (C Code):

```
if (RB[rs] == RB[rt]) {
  ac_pc = ac_pc + (imm << 2);
}
```

A.49 jgez



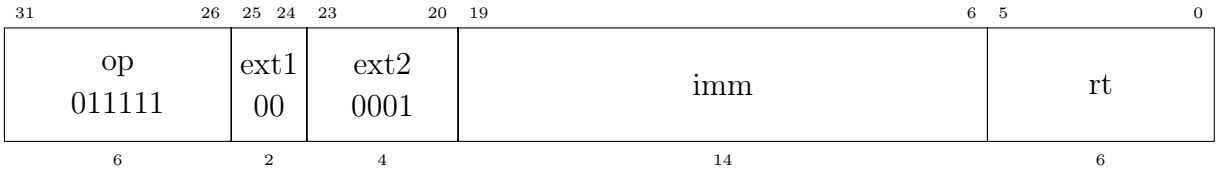
Encoding Format: PL20

Assembly Language Syntax: `jgez %reg, %exp(pcrel)`

Operation (C Code):

```
if (!(RB[rt] & 0x80000000)) {
  ac_pc = ac_pc + (imm << 2);
}
```

A.50 jgtz



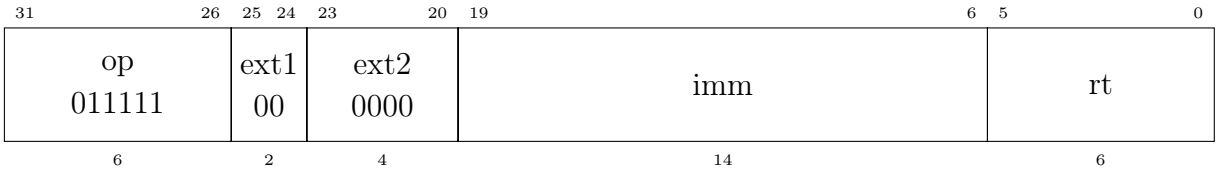
Encoding Format: PL20

Assembly Language Syntax: `jgtz %reg, %exp(pcrel)`

Operation (C Code):

```
if (!(RB[rt] & 0x80000000) && (RB[rt] != 0)) {
  ac_pc = ac_pc + (imm << 2);
}
```

A.51 jlez



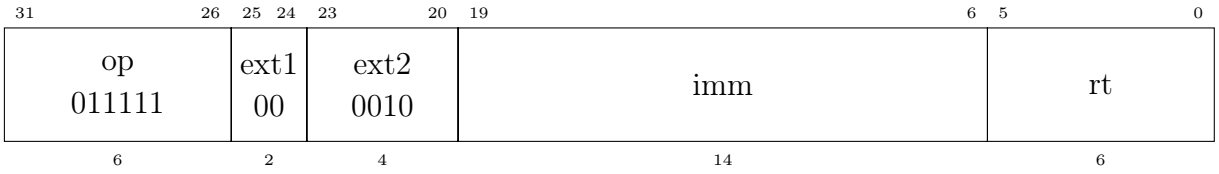
Encoding Format: PL20

Assembly Language Syntax: `jlez %reg, %exp(pcrel)`

Operation (C Code):

```
if ((RB[rt] == 0) || (RB[rt] & 0x80000000)) {
  ac_pc = ac_pc + (imm << 2), 1;
}
```

A.52 jltz



Encoding Format: PL20

Assembly Language Syntax: `jltz %reg, %exp(pcrel)`

Operation (C Code):

```
if (RB[rt] & 0x80000000) {
  ac_pc = ac_pc + (imm << 2);
}
```

A.53 jne

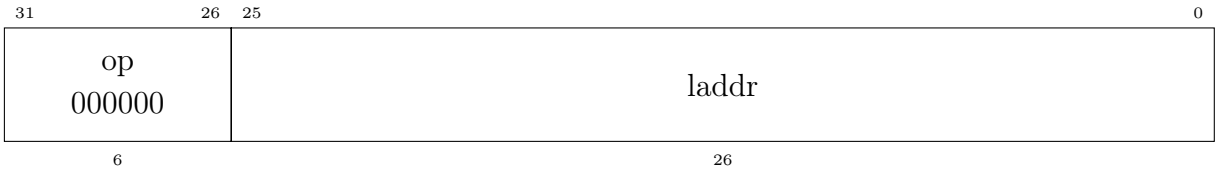
Encoding Format: PL26i

Assembly Language Syntax: `jne %reg, %reg, %exp(pcrel)`

Operation (C Code):

```
if (RB[rs] != RB[rt]) {
  ac_pc = ac_pc + (imm << 2);
}
```

A.54 jump



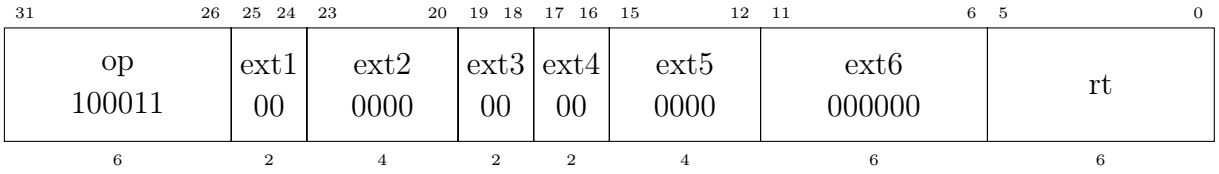
Encoding Format: PL26j

Assembly Language Syntax: `jump %exp(align)`

Operation (C Code):

```
if (laddr == 0) {
  printf("Jump to address zero\n");
  exit(-1);
}
laddr = laddr << 2;
ac_pc = (ac_pc & 0xF0000000) | laddr;
```

A.55 jumpr



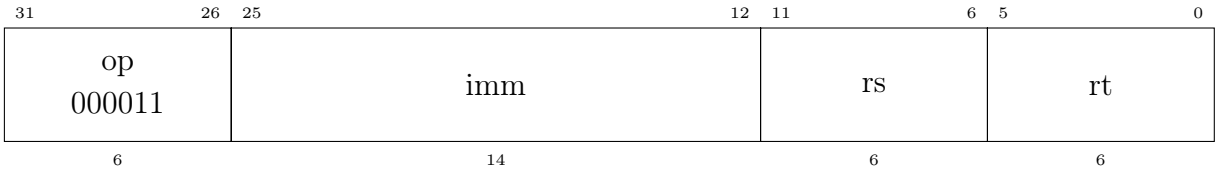
Encoding Format: PL6

Assembly Language Syntax: `jumpr %reg`

Operation (C Code):

```
ac_pc = RB[rt];
```

A.56 ldbu



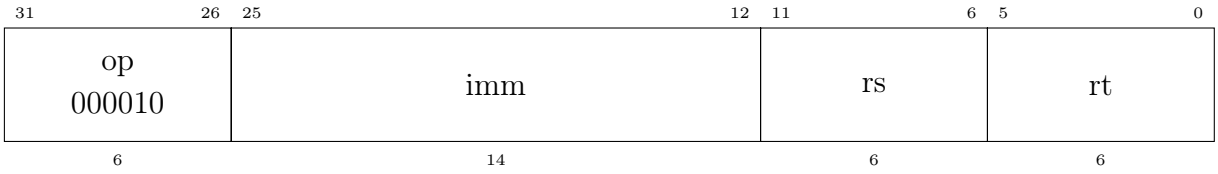
Encoding Format: PL26i

Assembly Language Syntax: `ldbu %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
unsigned char byte;
byte = DATA_PORT->read_byte(RB[rs] + imm);
RB[rt] = byte;
```

A.57 ldb



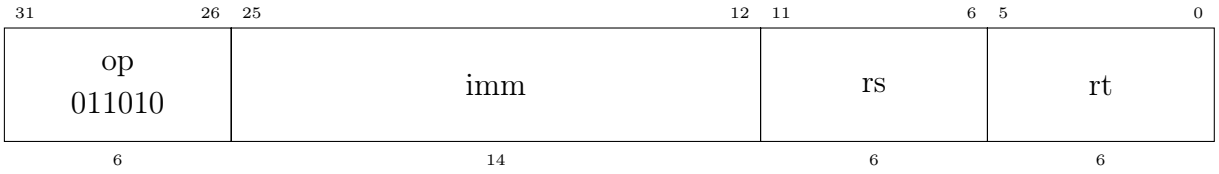
Encoding Format: PL26i

Assembly Language Syntax: `ldb %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
char byte;
byte = DATA_PORT->read_byte(RB[rs] + imm);
RB[rt] = (ac_Sword)byte;
```

A.58 ldc1



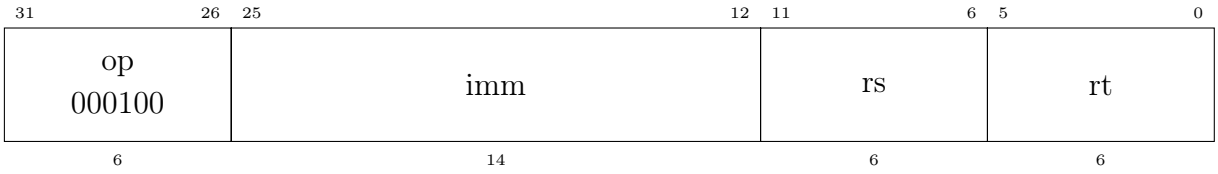
Encoding Format: PL26i

Assembly Language Syntax: `ldc1 %freg, %imm (%reg)`

Operation (C Code):

```
RBD[rt + 1] = DATA_PORT->read(RB[rs] + imm);
RBD[rt] = DATA_PORT->read(RB[rs] + imm + 4);
double temp = load_double(rt);
```

A.59 ldh



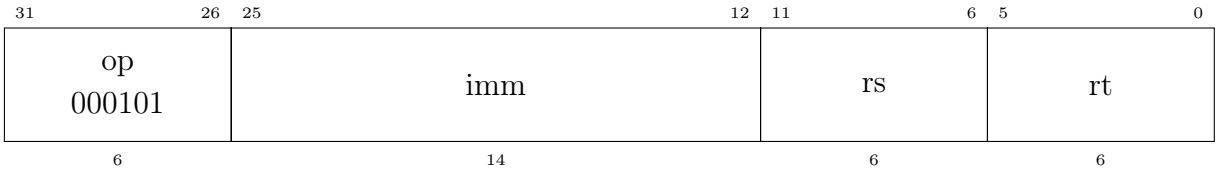
Encoding Format: PL26i

Assembly Language Syntax: `ldh %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
short int half;
half = DATA_PORT->read_half(RB[rs] + imm);
RB[rt] = (ac_Sword)half;
```

A.60 ldhu



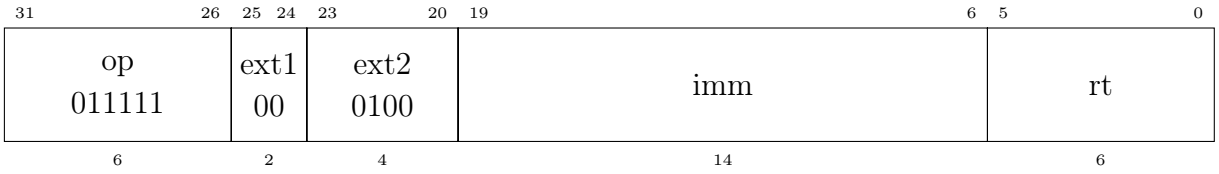
Encoding Format: PL26i

Assembly Language Syntax: `ldhu %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
unsigned short int half;
half = DATA_PORT->read_half(RB[rs] + imm);
RB[rt] = half;
```

A.61 ldi



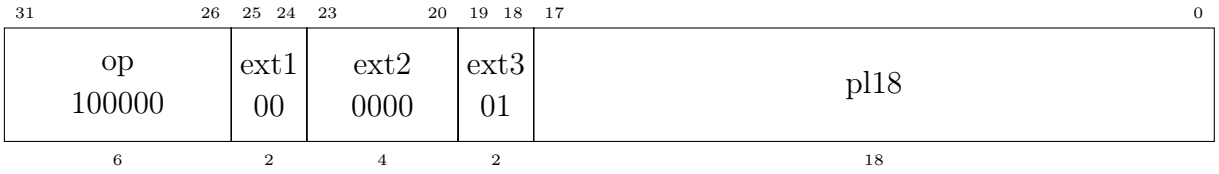
Encoding Format: PL20

Assembly Language Syntax: `ldi %reg, %imm`

Operation (C Code):

```
ldireg = rt;
RB[ldireg] &= 0xFFFFC000;
RB[ldireg] |= imm & 0x3FFF;
```

A.62 ldihi



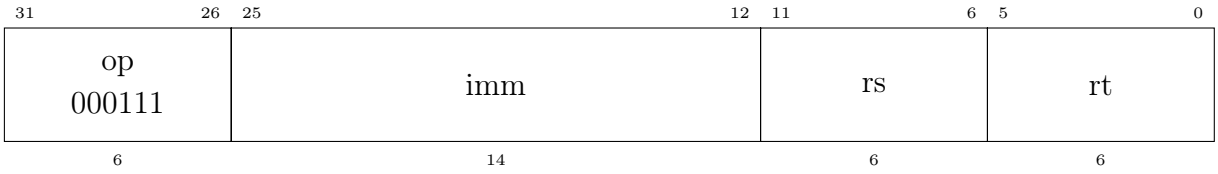
Encoding Format: PL18i

Assembly Language Syntax: `ldihi %imm`

Operation (C Code):

```
RB[ldireg] &= 0x3FFF;
RB[ldireg] |= pl18 << 14;
```

A.63 ldwl



Encoding Format: PL26i

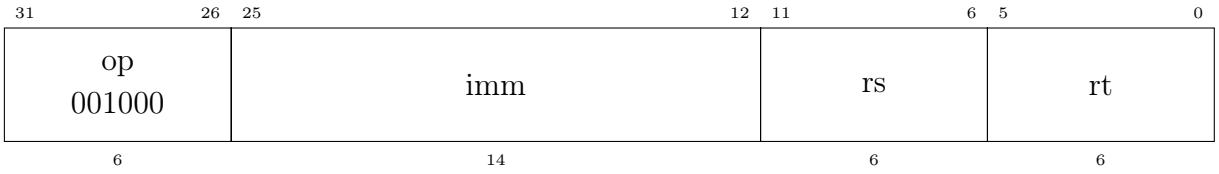
Assembly Language Syntax: `ldwl %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
unsigned int addr, offset;
ac_Uword data;

addr = RB[rs] + imm;
offset = (addr & 0x3) * 8;
data = DATA_PORT->read(addr & 0xFFFFFFFC);
data <<= offset;
data |= RB[rt] & ((1 << offset) - 1);
RB[rt] = data;
```

A.64 ldwr



Encoding Format: PL26i

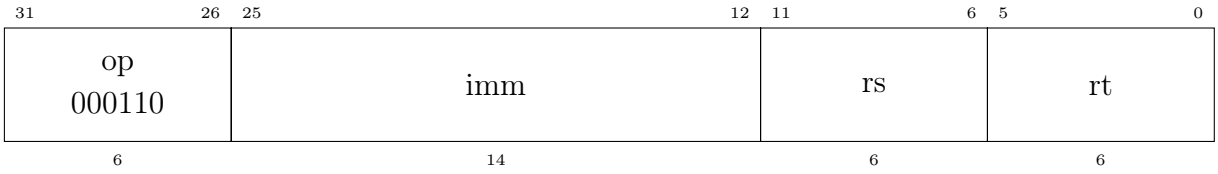
Assembly Language Syntax: `ldwr %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
unsigned int addr, offset;
ac_Uword data;

addr = RB[rs] + imm;
offset = (3 - (addr & 0x3)) * 8;
data = DATA_PORT->read(addr & 0xFFFFFFFC);
data >>= offset;
data |= RB[rt] & (0xFFFFFFFF << (32 - offset));
RB[rt] = data;
```

A.65 ldw



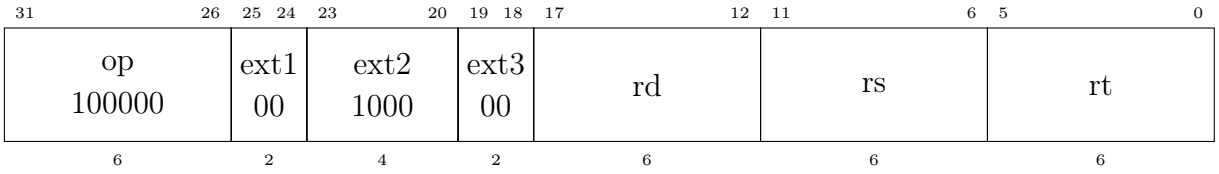
Encoding Format: PL26i

Assembly Language Syntax: `ldw %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
RB[rt] = DATA_PORT->read(RB[rs] + imm);
```

A.66 ldxc1



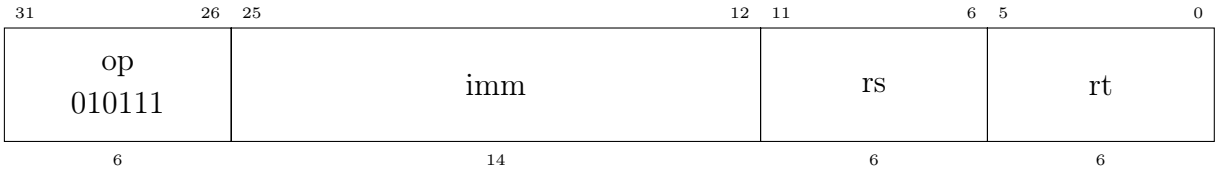
Encoding Format: PL18

Assembly Language Syntax: `ldxc1 %freg, %reg (%reg)`

Operation (C Code):

```
RBD[rd + 1] = DATA_PORT->read(RB[rt] + RB[rs]);
RBD[rd] = DATA_PORT->read(RB[rt] + RB[rs] + 4);
double temp = load_double(rd);
```

A.67 ll



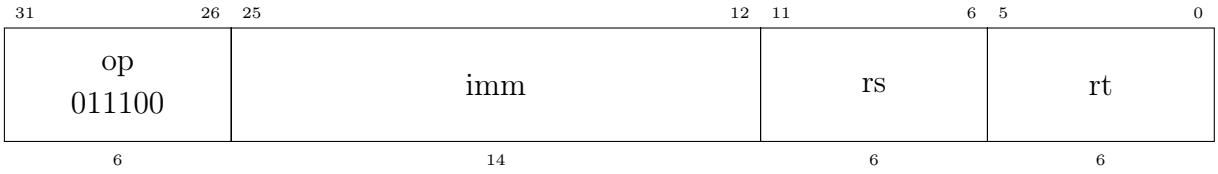
Encoding Format: PL26i

Assembly Language Syntax: `ll %reg, %imm (%reg)`

Operation (C Code):

```
RB[rt] = DATA_PORT->read(RB[rs] + imm);
```

A.68 lwc1



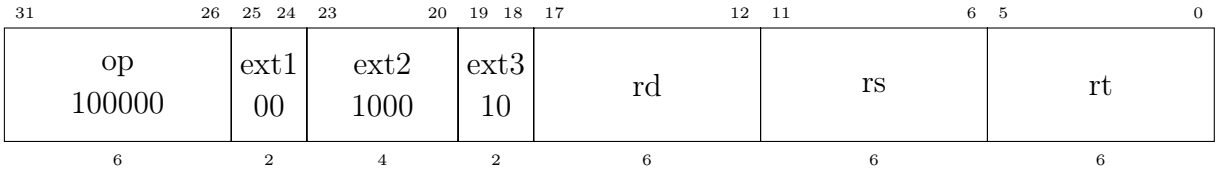
Encoding Format: PL26i

Assembly Language Syntax: `lwc1 %freg, %imm (%reg)`

Operation (C Code):

```
RBS[rt] = DATA_PORT->read(RB[rs] + imm);
float temp = load_float(rt);
```

A.69 lwxc1



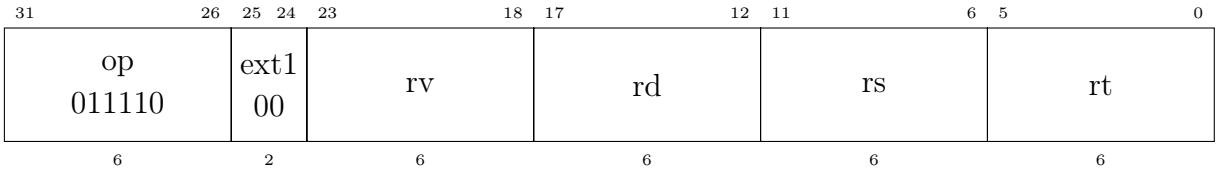
Encoding Format: PL18

Assembly Language Syntax: `lwxc1 %freg, %reg (%reg)`

Operation (C Code):

```
RBS[rd] = DATA_PORT->read(RB[rt] + RB[rs]);
float temp = load_float(rd);
```

A.70 madd.d



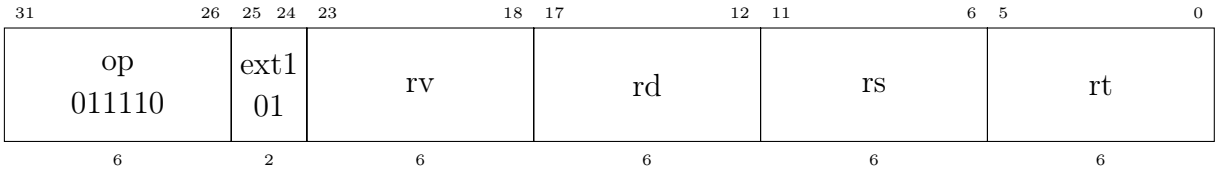
Encoding Format: PL24

Assembly Language Syntax: `madd.d %reg, %reg, %reg, %reg`

Operation (C Code):

```
double res = load_double(rs) * load_double(rt) + load_double(rv);
save_double(res, rd);
```

A.71 madd.s



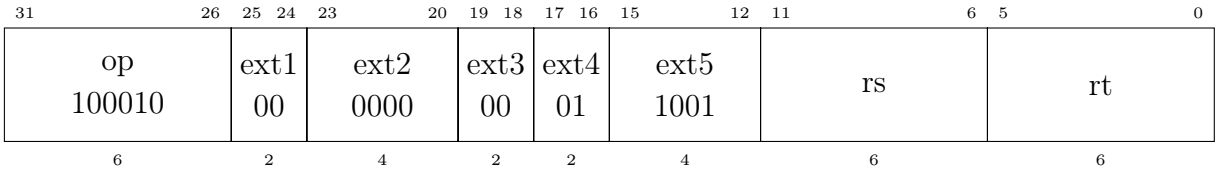
Encoding Format: PL24

Assembly Language Syntax: `madd.s %reg, %reg, %reg, %reg`

Operation (C Code):

```
float res = load_float(rs) * load_float(rt) + load_float(rv);
save_float(res, rd);
```

A.72 mfc1



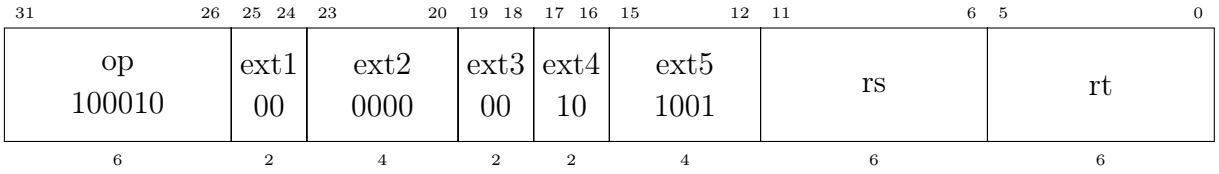
Encoding Format: PL12

Assembly Language Syntax: `mfc1 %reg, %freg`

Operation (C Code):

```
RB[rs] = RBS[rt];
```

A.73 mfhc1



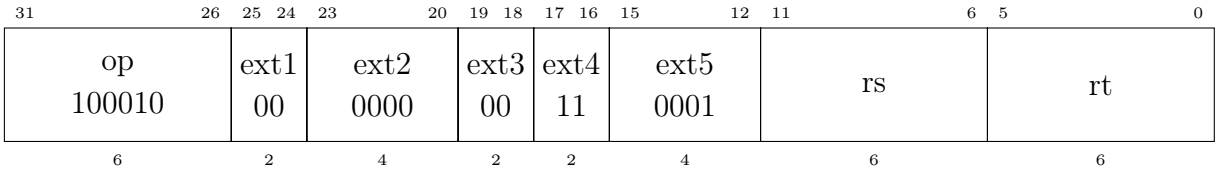
Encoding Format: PL12

Assembly Language Syntax: `mfhc1 %reg, %freg`

Operation (C Code):

```
uint64_t temp;
double input = load_double(rt);
memcpy(&temp, &input, sizeof(uint64_t));
RB[rs] = temp >> 32;
```

A.74 mflc1



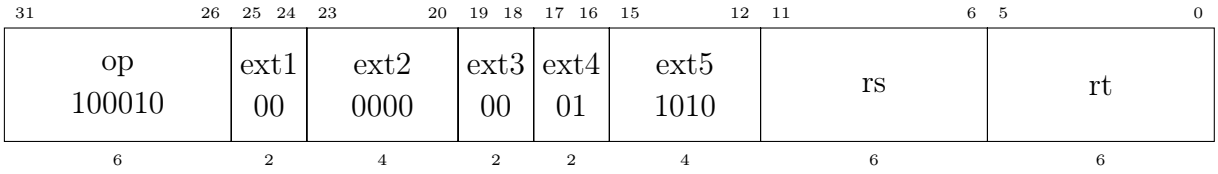
Encoding Format: PL12

Assembly Language Syntax: `mflc1 %reg, %freg`

Operation (C Code):

```
uint64_t temp;
double input = load_double(rt);
memcpy(&temp, &input, sizeof(uint64_t));
RB[rs] = temp & 0xFFFFFFFF;
```

A.75 mov.d



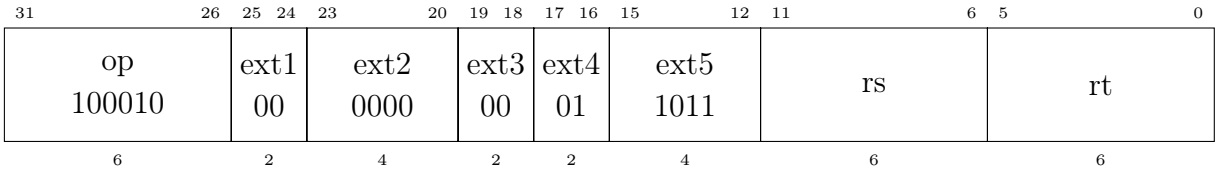
Encoding Format: PL12

Assembly Language Syntax: `mov.d %freg, %freg`

Operation (C Code):

```
double res = load_double(rt);
save_double(res, rs);
```

A.76 mov.s



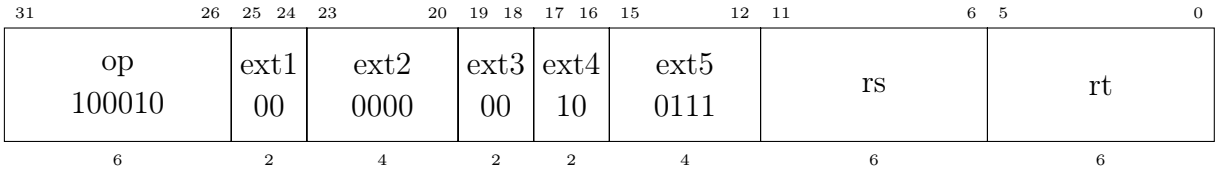
Encoding Format: PL12

Assembly Language Syntax: `mov.s %freg, %freg`

Operation (C Code):

```
float res = load_float(rt);
save_float(res, rs);
```

A.77 movf



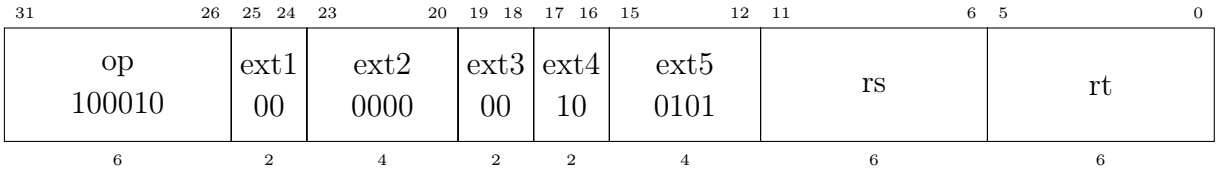
Encoding Format: PL12

Assembly Language Syntax: `movf %reg, %reg`

Operation (C Code):

```
if (cc == 0)
  RB[rs] = RB[rt];
```

A.78 movf.d



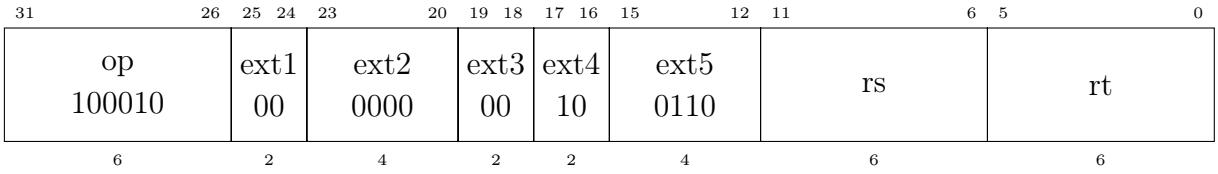
Encoding Format: PL12

Assembly Language Syntax: `movf.d %freg, %freg`

Operation (C Code):

```
if (cc == 0) {
  RBD[rs] = RBD[rt];
  RBD[rs + 1] = RBD[rt + 1];
}
```

A.79 movf.s



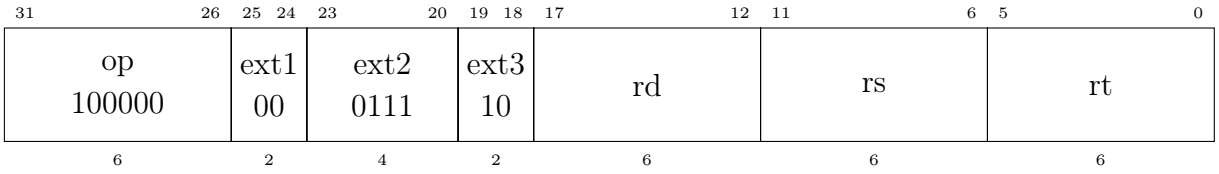
Encoding Format: PL12

Assembly Language Syntax: `movf.s %freg, %freg`

Operation (C Code):

```
if (cc == 0) {
  RBS[rs] = RBS[rt];
}
```

A.80 movn.d



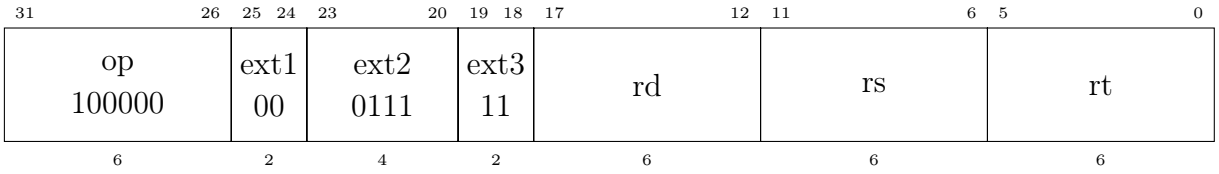
Encoding Format: PL18

Assembly Language Syntax: `movn.d %freg, %freg, %reg`

Operation (C Code):

```
if (RB[rt] != 0) {
  RBD[rd] = RBD[rs];
  RBD[rd + 1] = RBD[rs + 1];
}
```

A.81 movn.s



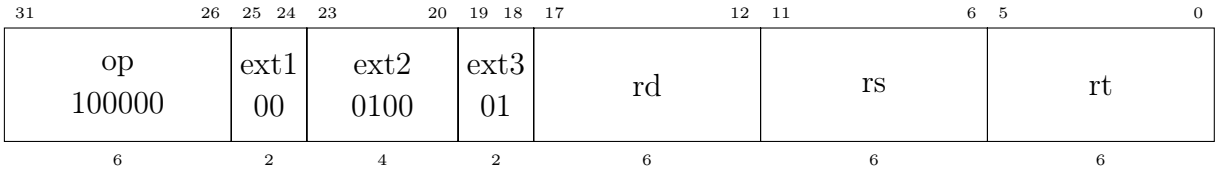
Encoding Format: PL18

Assembly Language Syntax: `movn.s %freg, %freg, %reg`

Operation (C Code):

```
if (RB[rt] != 0) {
  RBS[rd] = RBS[rs];
}
```

A.82 movn



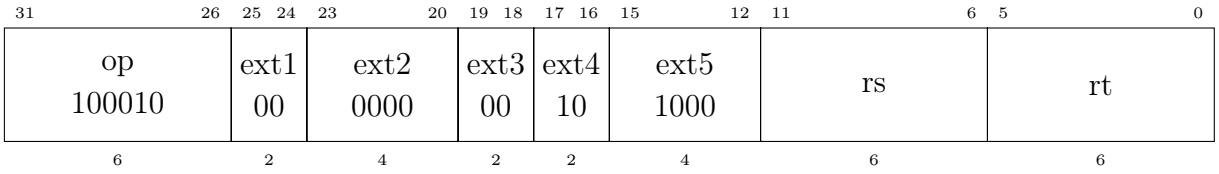
Encoding Format: PL18

Assembly Language Syntax: `movn %reg, %reg, %reg`

Operation (C Code):

```
if (RB[rt] != 0)
  RB[rd] = RB[rs];
```

A.83 movt



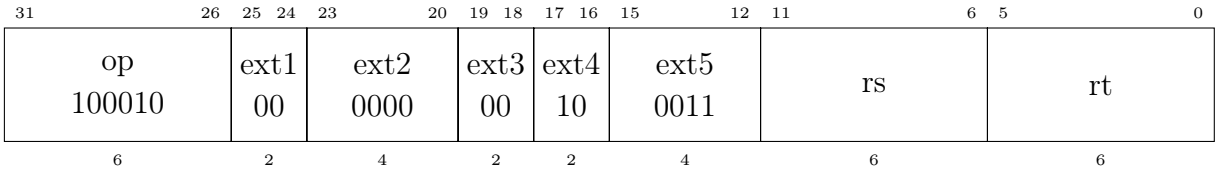
Encoding Format: PL12

Assembly Language Syntax: `movt %reg, %reg`

Operation (C Code):

```
if (cc != 0)
  RB[rs] = RB[rt];
```

A.84 movt.d



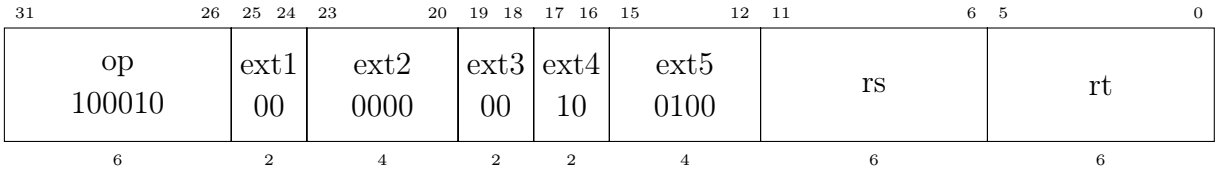
Encoding Format: PL12

Assembly Language Syntax: `movt.d %freg, %freg`

Operation (C Code):

```
if (cc != 0) {
  RBD[rs] = RBD[rt];
  RBD[rs + 1] = RBD[rt + 1];
}
```

A.85 movt.s



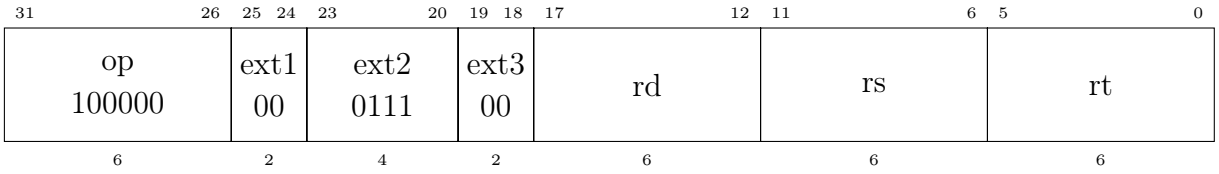
Encoding Format: PL12

Assembly Language Syntax: `movt.s %freg, %freg`

Operation (C Code):

```
if (cc != 0) {
  RBS[rs] = RBS[rt];
}
```

A.86 movz.d



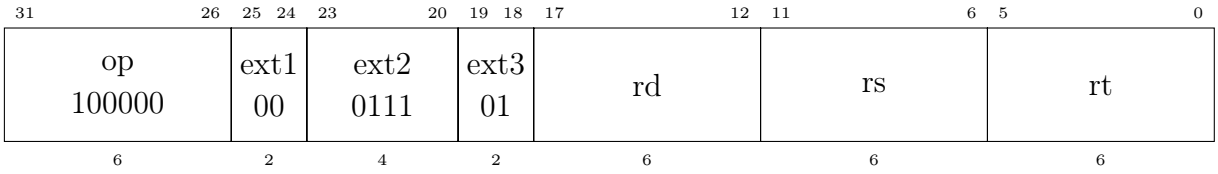
Encoding Format: PL18

Assembly Language Syntax: `movz.d %freg, %freg, %reg`

Operation (C Code):

```
if (RB[rt] != 0) {
  RBD[rd] = RBD[rs];
  RBD[rd + 1] = RBD[rs + 1];
}
```

A.87 movz.s



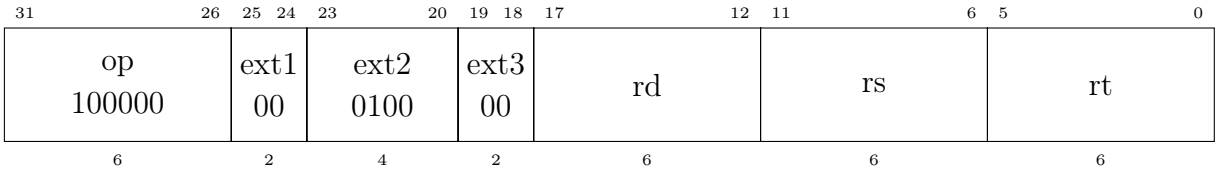
Encoding Format: PL18

Assembly Language Syntax: `movz.s %freg, %freg, %reg`

Operation (C Code):

```
if (RB[rt] != 0) {
  RBS[rd] = RBS[rs];
}
```

A.88 movz



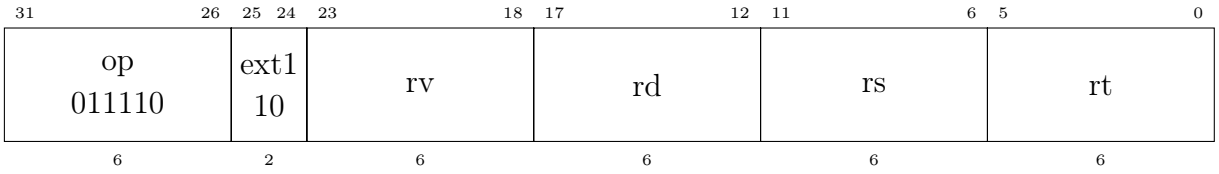
Encoding Format: PL18

Assembly Language Syntax: `movz %reg, %reg, %reg`

Operation (C Code):

```
if (RB[rt] == 0)
  RB[rd] = RB[rs];
```

A.89 msub.d



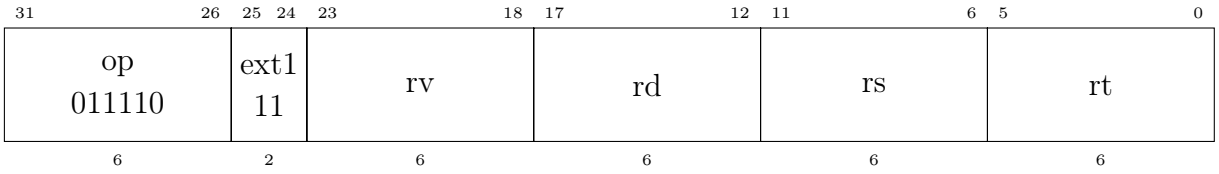
Encoding Format: PL24

Assembly Language Syntax: `msub.d %reg, %reg, %reg, %reg`

Operation (C Code):

```
double res = load_double(rs) * load_double(rt) - load_double(rv);
save_double(res, rd);
```

A.90 msub.s



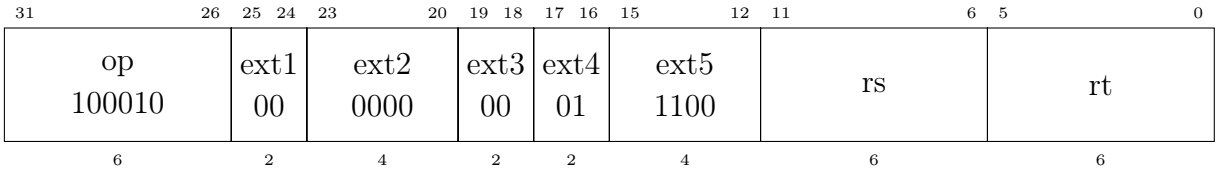
Encoding Format: PL24

Assembly Language Syntax: `msub.s %reg, %reg, %reg, %reg`

Operation (C Code):

```
float res = load_float(rs) * load_float(rt) - load_float(rv);
save_float(res, rd);
```

A.91 mtc1



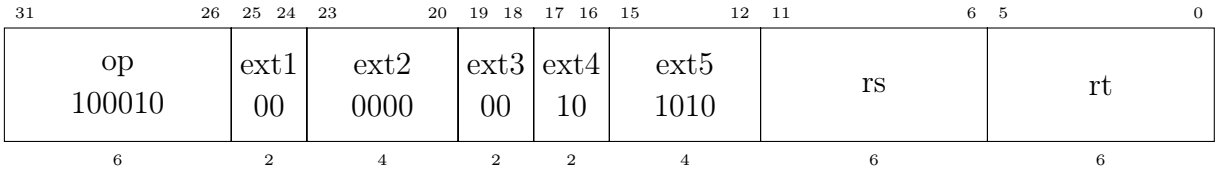
Encoding Format: PL12

Assembly Language Syntax: `mtc1 %reg, %freg`

Operation (C Code):

```
RBS[rt] = RB[rs];
```

A.92 mthc1



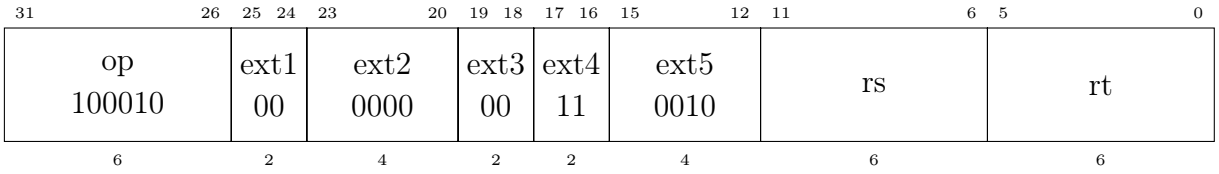
Encoding Format: PL12

Assembly Language Syntax: `mthc1 %reg, %freg`

Operation (C Code):

```
double temp = load_double(rt);
uint64_t to_int;
memcpy(&to_int, &temp, sizeof(uint64_t));
to_int = (to_int & 0xFFFFFFFFULL) + (((uint64_t)RB[rs]) << 32);
memcpy(&temp, &to_int, sizeof(uint64_t));
save_double(temp, rt);
```

A.93 mtlc1



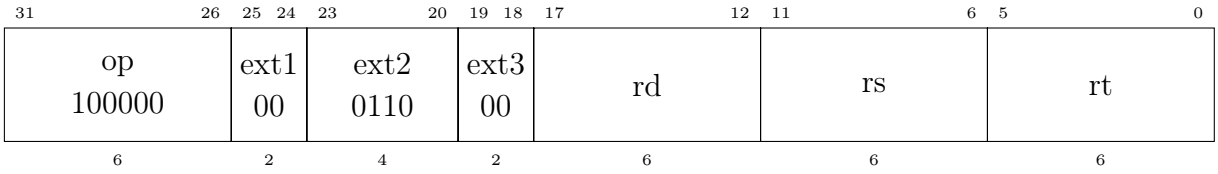
Encoding Format: PL12

Assembly Language Syntax: `mtlc1 %reg, %freg`

Operation (C Code):

```
double temp = load_double(rt);
uint64_t to_int;
memcpy(&to_int, &temp, sizeof(uint64_t));
to_int = (to_int & 0xFFFFFFFF00000000ULL) + (((uint64_t)RB[rs]));
memcpy(&temp, &to_int, sizeof(uint64_t));
save_double(temp, rt);
```

A.94 mul.d



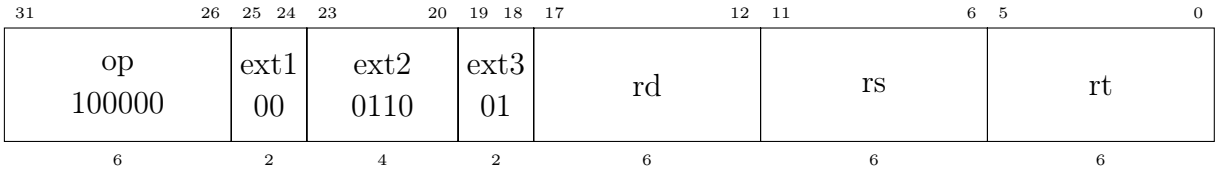
Encoding Format: PL18

Assembly Language Syntax: `mul.d %freg, %freg, %freg`

Operation (C Code):

```
double res = load_double(rs) * load_double(rt);
save_double(res, rd);
```

A.95 mul.s



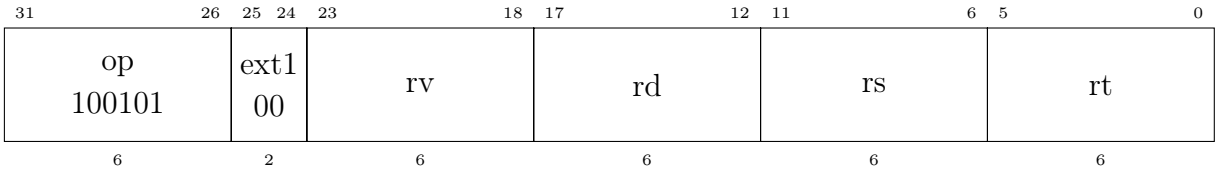
Encoding Format: PL18

Assembly Language Syntax: `mul.s %freg, %freg, %freg`

Operation (C Code):

```
float res = load_float(rs) * load_float(rt);
save_float(res, rd);
```

A.96 mulu



Encoding Format: PL24

Assembly Language Syntax: `mulu %reg, %reg, %reg, %reg`

Operation (C Code):

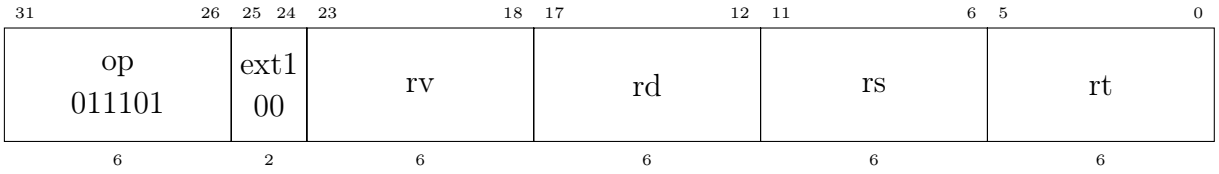
```
unsigned long long result;
int half_result;

result = (ac_Uword)RB[rs];
result *= (ac_Uword)RB[rt];

half_result = (result & 0xFFFFFFFF);
if (rd != 0)
  RB[rd] = half_result;

half_result = ((result >> 32) & 0xFFFFFFFF);
if (rv != 0)
  RB[rv] = half_result;
```

A.97 mul



Encoding Format: PL24

Assembly Language Syntax: `mul %reg, %reg, %reg, %reg`

Operation (C Code):

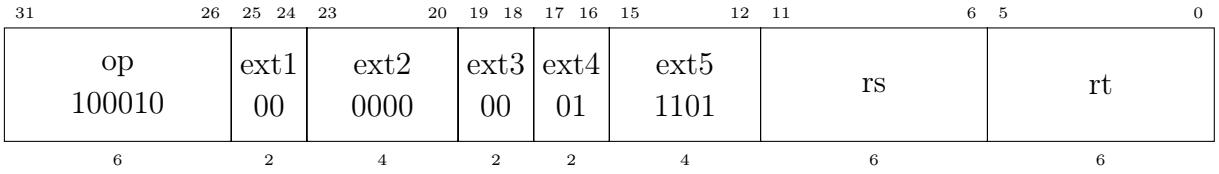
```
long long result;
int half_result;

result = (ac_Sword)RB[rs];
result *= (ac_Sword)RB[rt];

half_result = (result & 0xFFFFFFFF);
if (rd != 0)
  RB[rd] = half_result;

half_result = ((result >> 32) & 0xFFFFFFFF);
if (rv != 0)
  RB[rv] = half_result;
```

A.98 neg.d



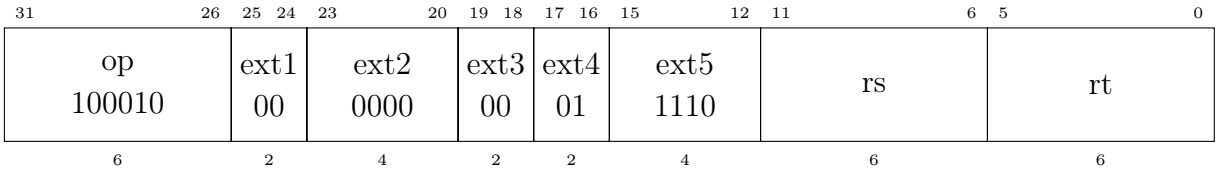
Encoding Format: PL12

Assembly Language Syntax: `neg.d %freg, %freg`

Operation (C Code):

```
double res = -load_double(rt);
save_double(res, rs);
```

A.99 neg.s



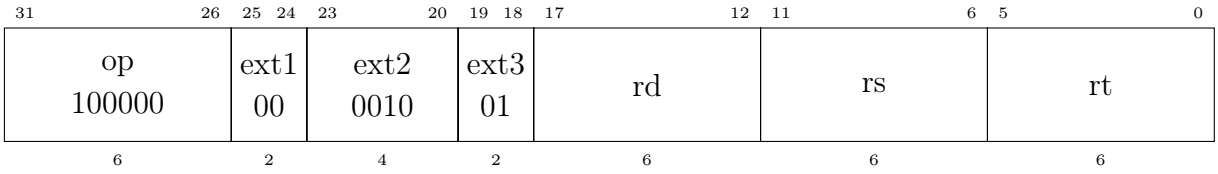
Encoding Format: PL12

Assembly Language Syntax: `neg.s %freg, %freg`

Operation (C Code):

```
float res = -load_float(rt);
save_float(res, rs);
```

A.100 nor



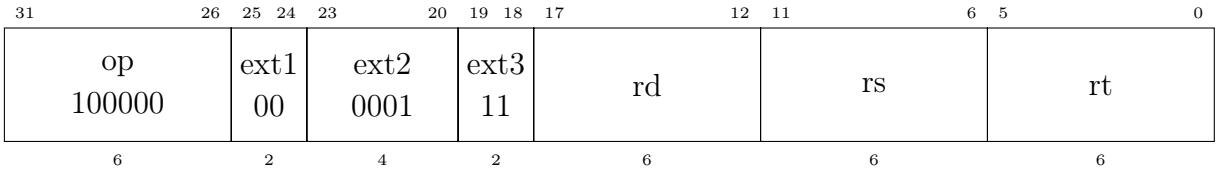
Encoding Format: PL18

Assembly Language Syntax: `nor %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = ~(RB[rs] | RB[rt]);
```

A.101 or



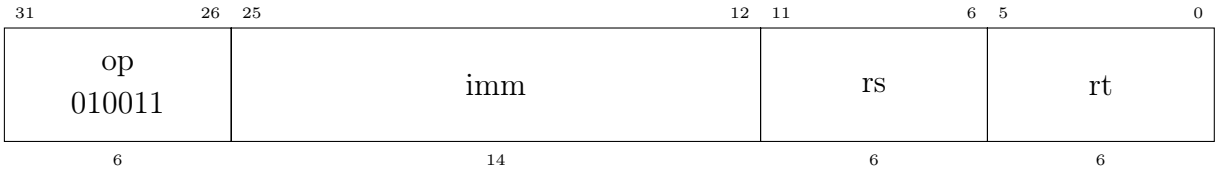
Encoding Format: PL18

Assembly Language Syntax: `or %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = RB[rs] | RB[rt];
```

A.102 ori



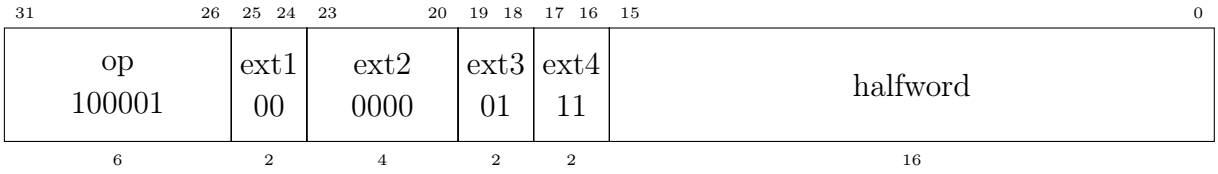
Encoding Format: PL26i

Assembly Language Syntax: `ori %reg, %reg, %imm`

Operation (C Code):

```
RB[rt] = RB[rs] | (imm & 0x3FFF);
```

A.103 rcall



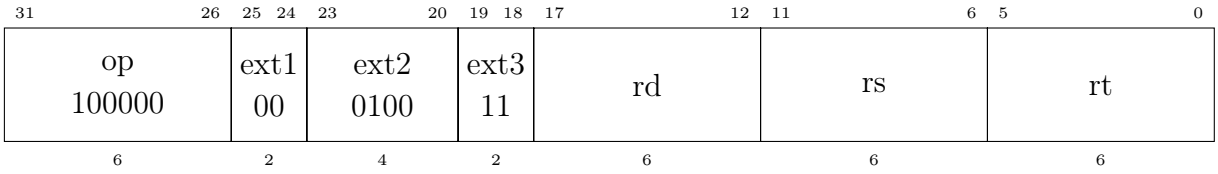
Encoding Format: PL16

Assembly Language Syntax: `rcall %exp(pcrel)`

Operation (C Code):

```
RB[Ra] = ac_pc;
ac_pc = ac_pc + (halfword << 2);
```

A.104 rorr



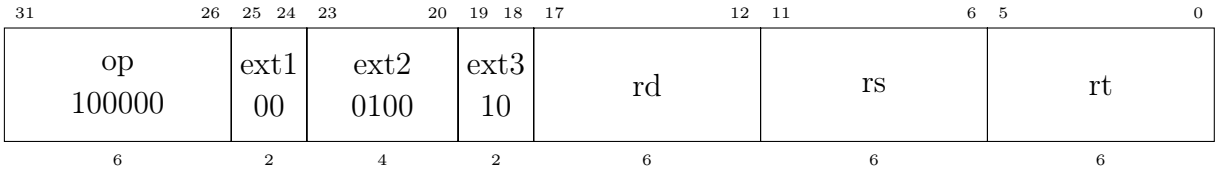
Encoding Format: PL18

Assembly Language Syntax: `rorr %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = rotate_right(RB[rt], RB[rs]);
```

A.105 ror



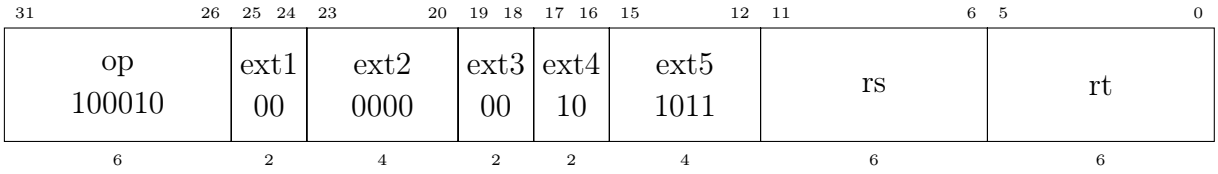
Encoding Format: PL18

Assembly Language Syntax: `ror %reg, %reg, %imm`

Operation (C Code):

```
RB[rd] = rotate_right(RB[rt], rs);
```

A.106 round.w.d



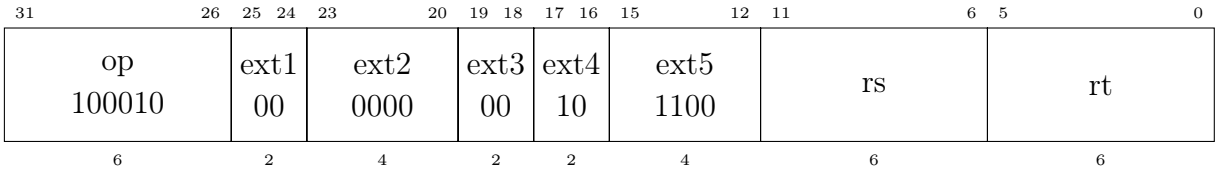
Encoding Format: PL12

Assembly Language Syntax: `round.w.d %reg, %freg`

Operation (C Code):

```
if (fesetround(FE_TONEAREST) == 0) {
  fprintf(stderr, "Failed to set rounding mode.\n");
  exit(EXIT_FAILURE);
}
int32_t res = (int) nearbyint(load_double(rt));
RB[rt] = res;
```

A.107 round.w.s



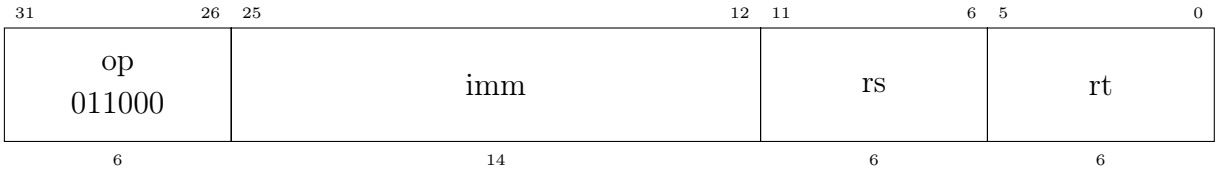
Encoding Format: PL12

Assembly Language Syntax: `round.w.s %reg, %freg`

Operation (C Code):

```
if (fesetround(FE_TONEAREST) == 0) {
  fprintf(stderr, "Failed to set rounding mode.\n");
  exit(EXIT_FAILURE);
}
int32_t res = (int) nearbyintf(load_float(rt));
RB[rt] = res;
```

A.108 sc



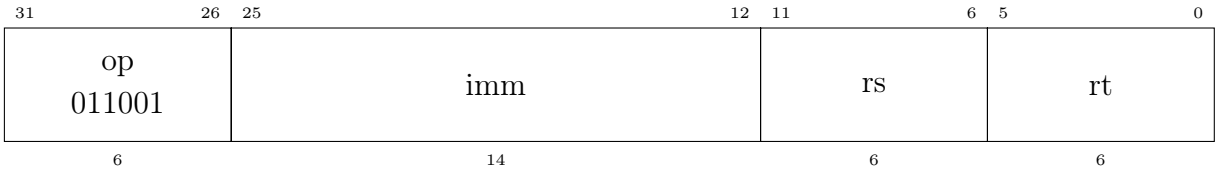
Encoding Format: PL26i

Assembly Language Syntax: `sc %reg, %imm (%reg)`

Operation (C Code):

```
DATA_PORT->write(RB[rs] + imm, RB[rt]);
RB[rt] = 1;
```

A.109 sdc1



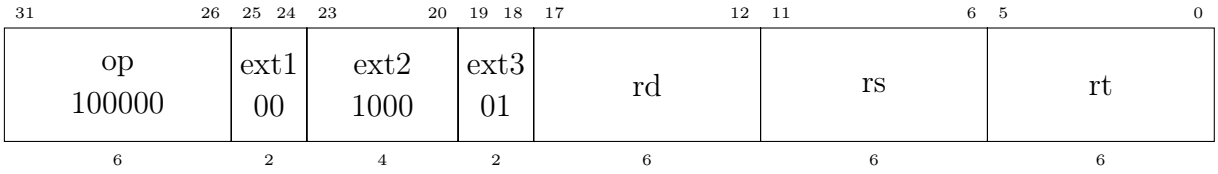
Encoding Format: PL26i

Assembly Language Syntax: `sdc1 %freg, %imm (%reg)`

Operation (C Code):

```
DATA_PORT->write(RB[rs] + imm + 4, RBD[rt]);
DATA_PORT->write(RB[rs] + imm, RBD[rt + 1]);
```

A.110 sdxc1



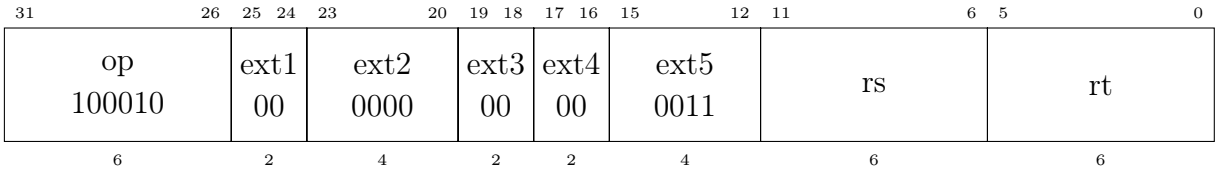
Encoding Format: PL18

Assembly Language Syntax: `sdxc1 %freg, %reg (%reg)`

Operation (C Code):

```
DATA_PORT->write(RB[rt] + RB[rs] + 4, RBD[rd]);
DATA_PORT->write(RB[rt] + RB[rs], RBD[rd + 1]);
```

A.111 seb



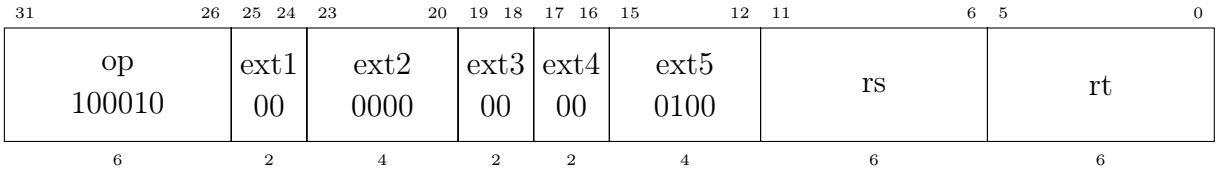
Encoding Format: PL12

Assembly Language Syntax: `seb %reg, %reg`

Operation (C Code):

```
RB[rs] = sign_extend(RB[rt], 8);
```

A.112 seh



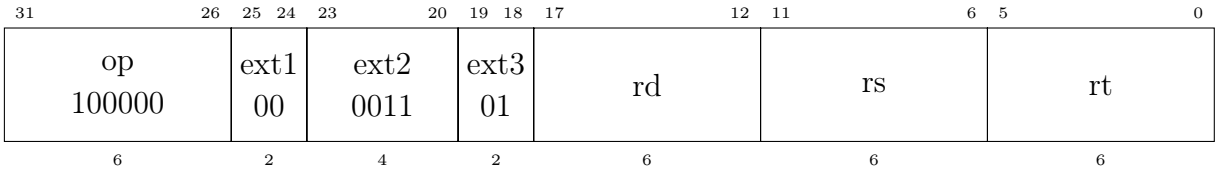
Encoding Format: PL12

Assembly Language Syntax: `seh %reg, %reg`

Operation (C Code):

```
RB[rs] = sign_extend(RB[rt], 16);
```

A.113 shlr



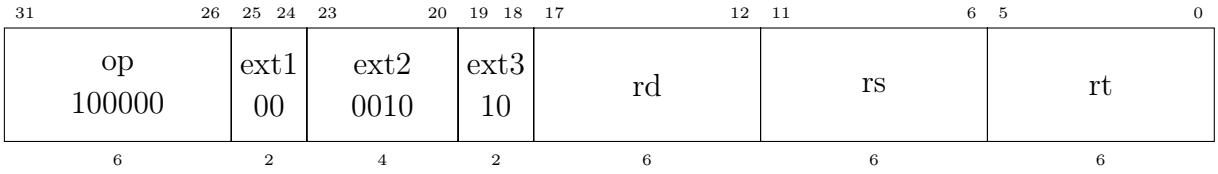
Encoding Format: PL18

Assembly Language Syntax: `shlr %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = RB[rt] << (RB[rs] & 0x1F);
```

A.114 shl



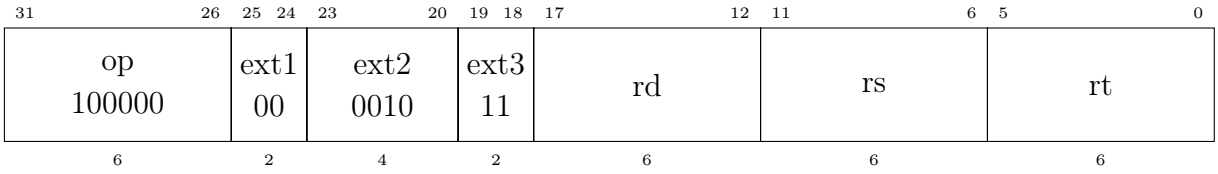
Encoding Format: PL18

Assembly Language Syntax: `shl %reg, %reg, %imm`

Operation (C Code):

```
RB[rd] = RB[rt] << rs;
```

A.115 shr



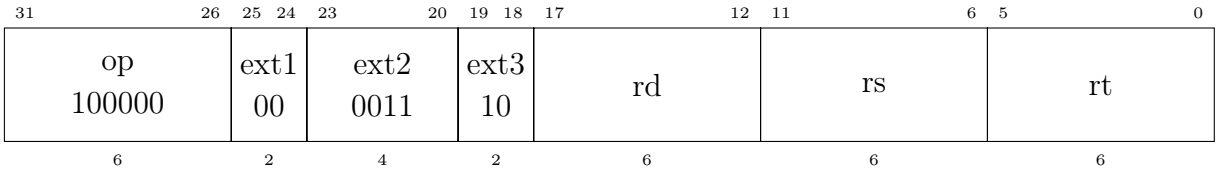
Encoding Format: PL18

Assembly Language Syntax: `shr %reg, %reg, %imm`

Operation (C Code):

```
RB[rd] = RB[rt] >> rs;
```

A.116 shrr



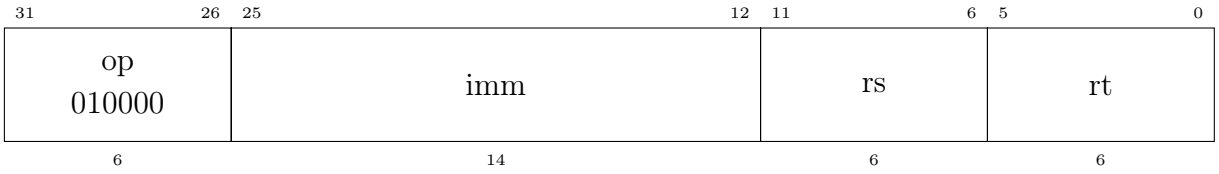
Encoding Format: PL18

Assembly Language Syntax: `shrr %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = RB[rt] >> (RB[rs] & 0x1F);
```

A.117 slti



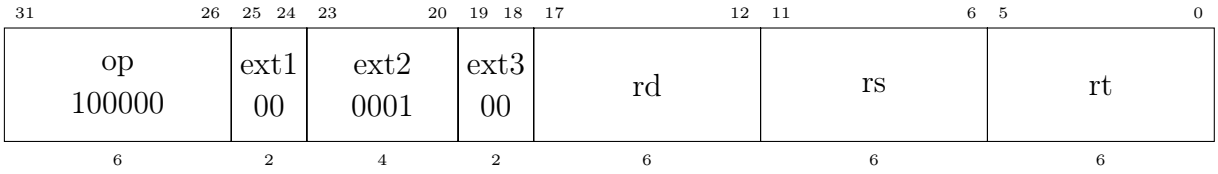
Encoding Format: PL26i

Assembly Language Syntax: `slti %reg, %reg, %exp`

Operation (C Code):

```
// Set the RD if RS< IMM
if ((ac_Sword)RB[rs] < (ac_Sword)imm)
  RB[rt] = 1;
// Else reset RD
else
  RB[rt] = 0;
```

A.118 slt



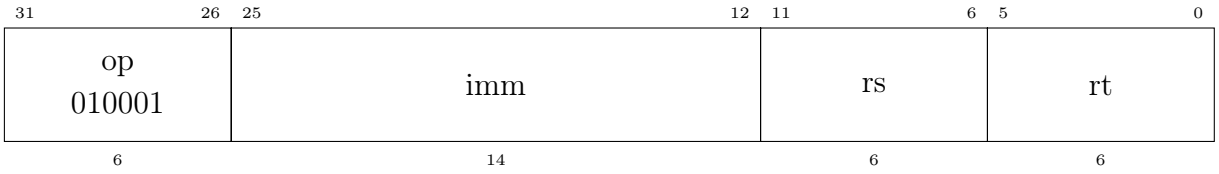
Encoding Format: PL18

Assembly Language Syntax: `slt %reg, %reg, %reg`

Operation (C Code):

```
// Set the RD if RS< RT
if ((ac_Sword)RB[rs] < (ac_Sword)RB[rt])
  RB[rd] = 1;
// Else reset RD
else
  RB[rd] = 0;
```

A.119 sltiu



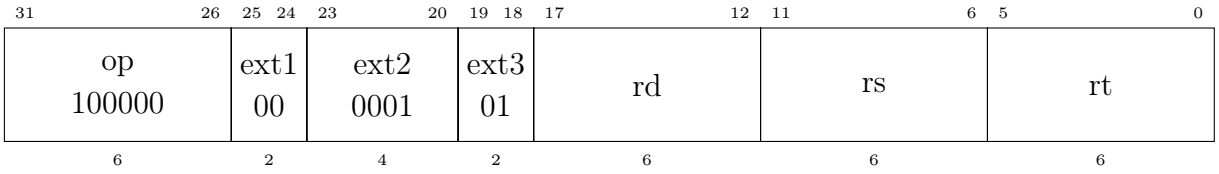
Encoding Format: PL26i

Assembly Language Syntax: `sltiu %reg, %reg, %exp`

Operation (C Code):

```
// Set the RD if RS< IMM
if ((ac_Uword)RB[rs] < (ac_Uword)(imm & 0x3FFF))
  RB[rt] = 1;
// Else reset RD
else
  RB[rt] = 0;
```

A.120 sltu



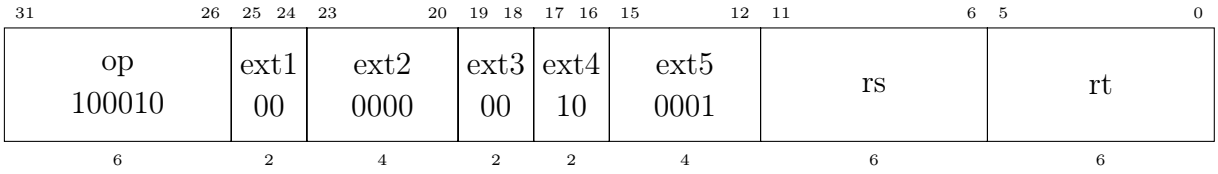
Encoding Format: PL18

Assembly Language Syntax: `sltu %reg, %reg, %reg`

Operation (C Code):

```
// Set the RD if RS < RT
if (RB[rs] < RB[rt])
  RB[rd] = 1;
// Else reset RD
else
  RB[rd] = 0;
```

A.121 sqrt.d



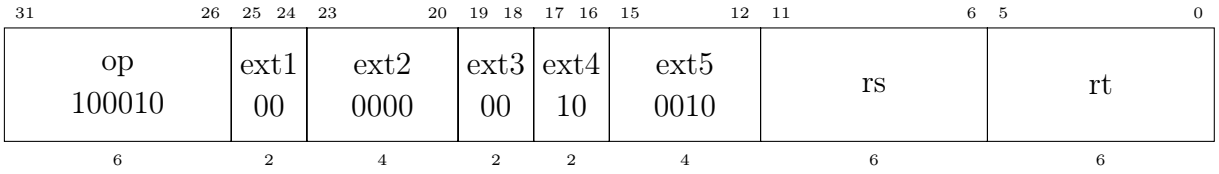
Encoding Format: PL12

Assembly Language Syntax: `sqrt.d %freg, %freg`

Operation (C Code):

```
double res = sqrt(load_double(rt));
save_double(res, rs);
```

A.122 sqrt.s



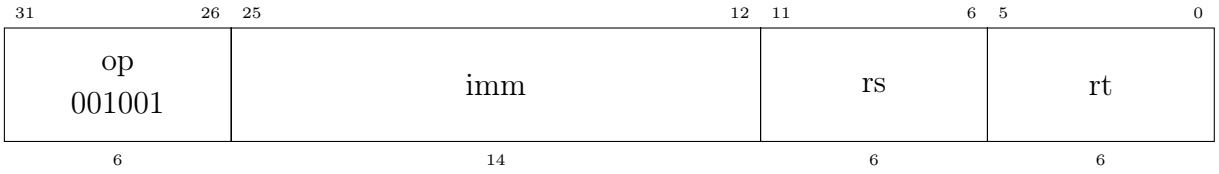
Encoding Format: PL12

Assembly Language Syntax: `sqrt.s %freg, %freg`

Operation (C Code):

```
float res = sqrtf(load_float(rt));
save_float(res, rs);
```

A.123 stb



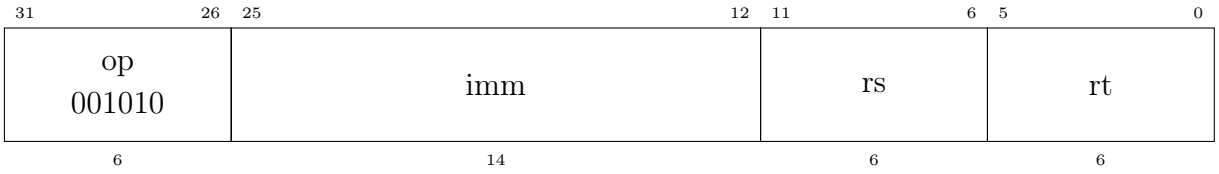
Encoding Format: PL26i

Assembly Language Syntax: `stb %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
unsigned char byte;
byte = RB[rt] & 0xFF;
DATA_PORT->write_byte(RB[rs] + imm, byte);
```

A.124 sth



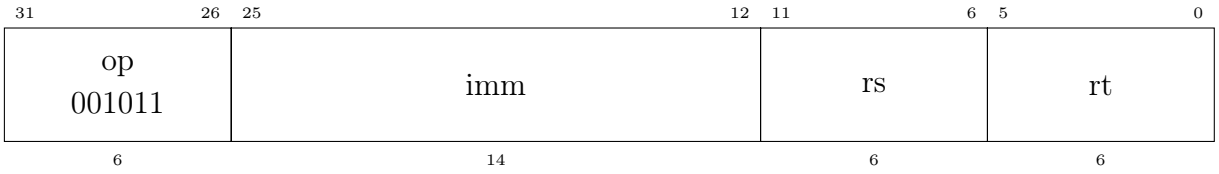
Encoding Format: PL26i

Assembly Language Syntax: `sth %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
unsigned short int half;
half = RB[rt] & 0xFFFF;
DATA_PORT->write_half(RB[rs] + imm, half);
```

A.125 stw



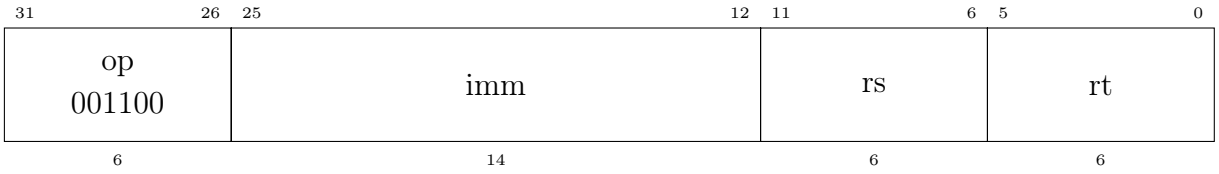
Encoding Format: PL26i

Assembly Language Syntax: `stw %reg, \%lo(%exp)(%reg)`

Operation (C Code):

```
DATA_PORT->write(RB[rs] + imm, RB[rt]);
```

A.126 stwl



Encoding Format: PL26i

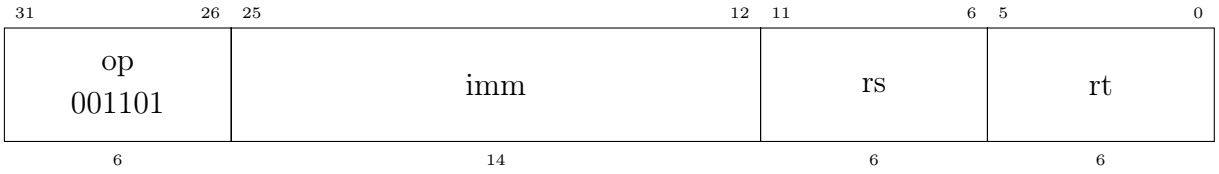
Assembly Language Syntax: `stwl %reg, (%reg)`

Operation (C Code):

```
unsigned int addr, offset;
ac_Uword data;

addr = RB[rs] + imm;
offset = (addr & 0x3) * 8;
data = RB[rt];
data >>= offset;
data |= DATA_PORT->read(addr & 0xFFFFFFFC)
        & (0xFFFFFFFF << (32 - offset));
DATA_PORT->write(addr & 0xFFFFFFFC, data);
```

A.127 stwr



Encoding Format: PL26i

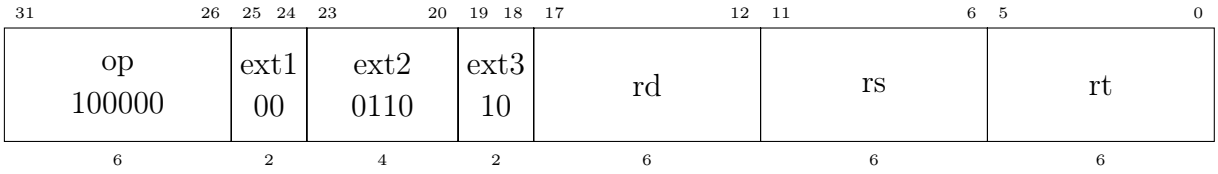
Assembly Language Syntax: `stwr %reg, (%reg)`

Operation (C Code):

```
unsigned int addr, offset;
ac_Uword data;

addr = RB[rs] + imm;
offset = (3 - (addr & 0x3)) * 8;
data = RB[rt];
data <<= offset;
data |= DATA_PORT->read(addr & 0xFFFFFFFC) & ((1 << offset) - 1);
DATA_PORT->write(addr & 0xFFFFFFFC, data);
```

A.128 sub.d



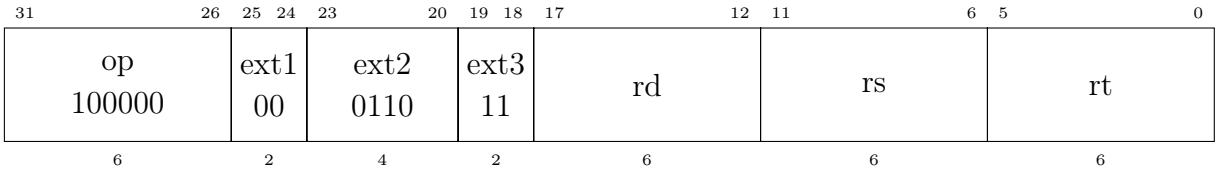
Encoding Format: PL18

Assembly Language Syntax: `sub.d %freg, %freg, %freg`

Operation (C Code):

```
double res = load_double(rs) - load_double(rt);
save_double(res, rd);
```

A.129 sub.s



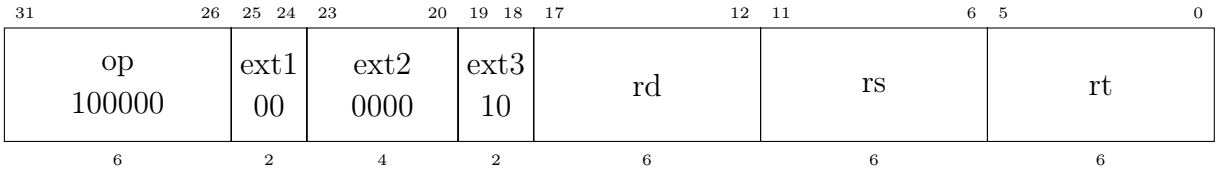
Encoding Format: PL18

Assembly Language Syntax: `sub.s %freg, %freg, %freg`

Operation (C Code):

```
float res = load_float(rs) - load_float(rt);
save_float(res, rd);
```

A.130 sub



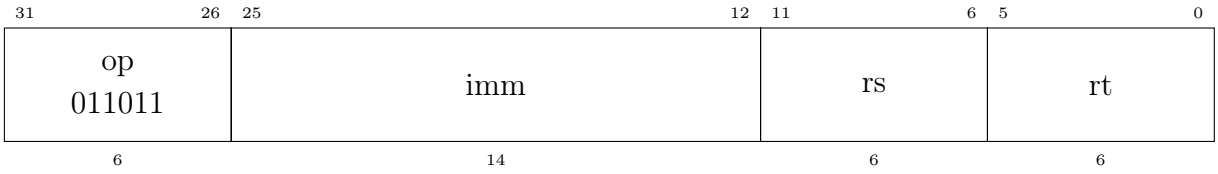
Encoding Format: PL18

Assembly Language Syntax: `sub %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = RB[rs] - RB[rt];
```

A.131 swc1



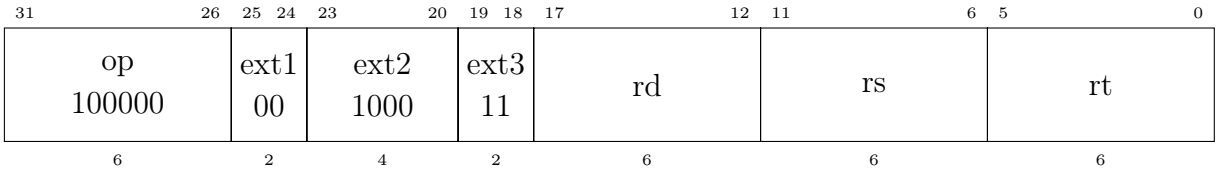
Encoding Format: PL26i

Assembly Language Syntax: `swc1 %freg, %imm (%reg)`

Operation (C Code):

```
DATA_PORT->write(RB[rs] + imm, RBS[rt]);
```

A.132 swxc1



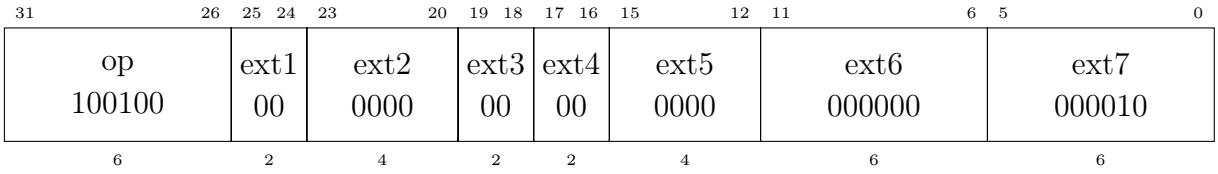
Encoding Format: PL18

Assembly Language Syntax: `swxc1 %freg, %reg (%reg)`

Operation (C Code):

```
DATA_PORT->write(RB[rt] + RB[rs], RBS[rd]);
```

A.133 sync



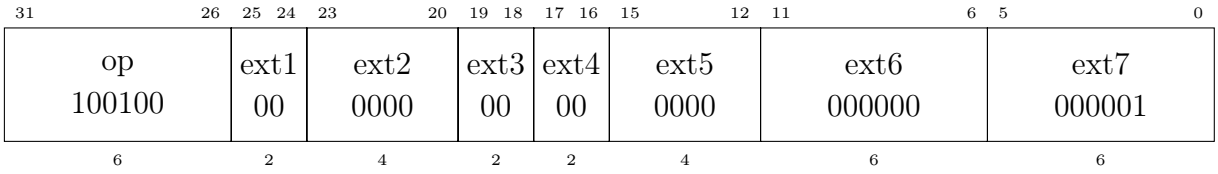
Encoding Format: PL0

Assembly Language Syntax: `sync`

Operation (C Code):

```
// Memory barrier
```

A.134 syscall



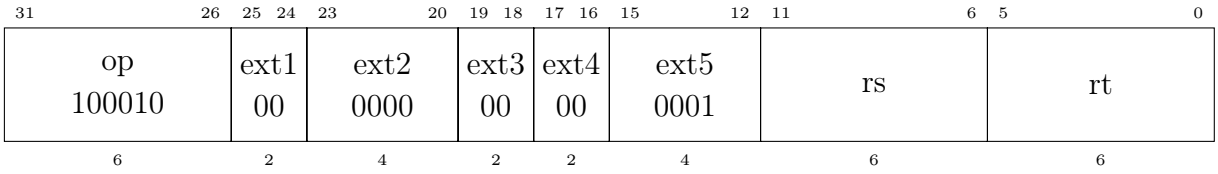
Encoding Format: PL0

Assembly Language Syntax: `syscall`

Operation (C Code):

```
uint32_t sysnum = RB[4];
if (sysnum == 0x100C) {
  fprintf(stderr, "Warning: fstat unimplemented.\n");
  RB[2] = -1;
  return;
}
// relocating regs
RB[4] = RB[5];
RB[5] = RB[6];
RB[6] = RB[7];
RB[7] = RB[8];
RB[8] = RB[9];
if (syscall.process_syscall(sysnum) == -1) {
  fprintf(stderr, "Warning: Unimplemented syscall.\n");
  RB[2] = -1;
}
// Sets a3 to 1 or 0 for error/success
if ((int)RB[2] < 0)
  RB[7] = 1;
else
  RB[7] = 0;
```

A.135 teq



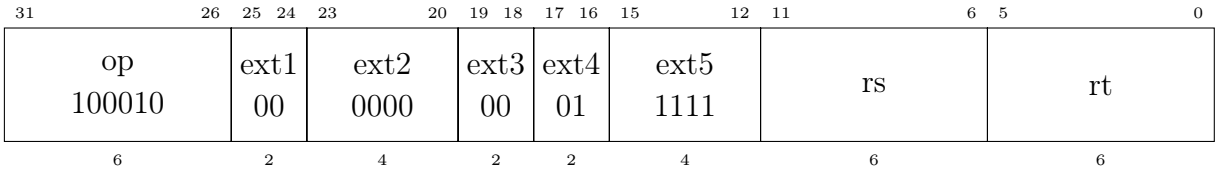
Encoding Format: PL12

Assembly Language Syntax: `teq %reg, %reg`

Operation (C Code):

```
if (RB[rs] == RB[rt]) {
  fprintf(stderr, "Trap generated at PC=0x%X\n", (uint32_t)ac_pc);
  exit(EXIT_FAILURE);
}
```

A.136 trunc.w.d



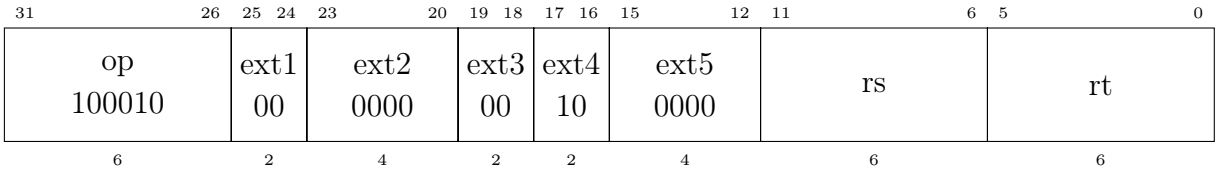
Encoding Format: PL12

Assembly Language Syntax: `trunc.w.d %freg, %freg`

Operation (C Code):

```
RBS[rs] = (int32_t)load_double(rt);
```

A.137 trunc.w.s



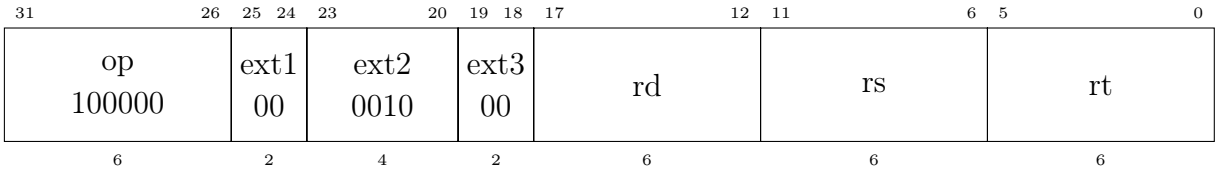
Encoding Format: PL12

Assembly Language Syntax: `trunc.w.s %freg, %freg`

Operation (C Code):

```
RBS[rs] = (int32_t)load_float(rt);
```

A.138 xor



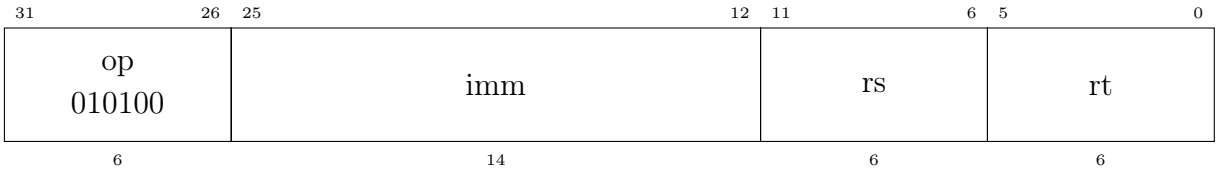
Encoding Format: PL18

Assembly Language Syntax: `xor %reg, %reg, %reg`

Operation (C Code):

```
RB[rd] = RB[rs] ^ RB[rt];
```

A.139 xori



Encoding Format: PL26i

Assembly Language Syntax: `xori %reg, %reg, %imm`

Operation (C Code):

```
RB[rt] = RB[rs] ^ (imm & 0x3FFF);
```