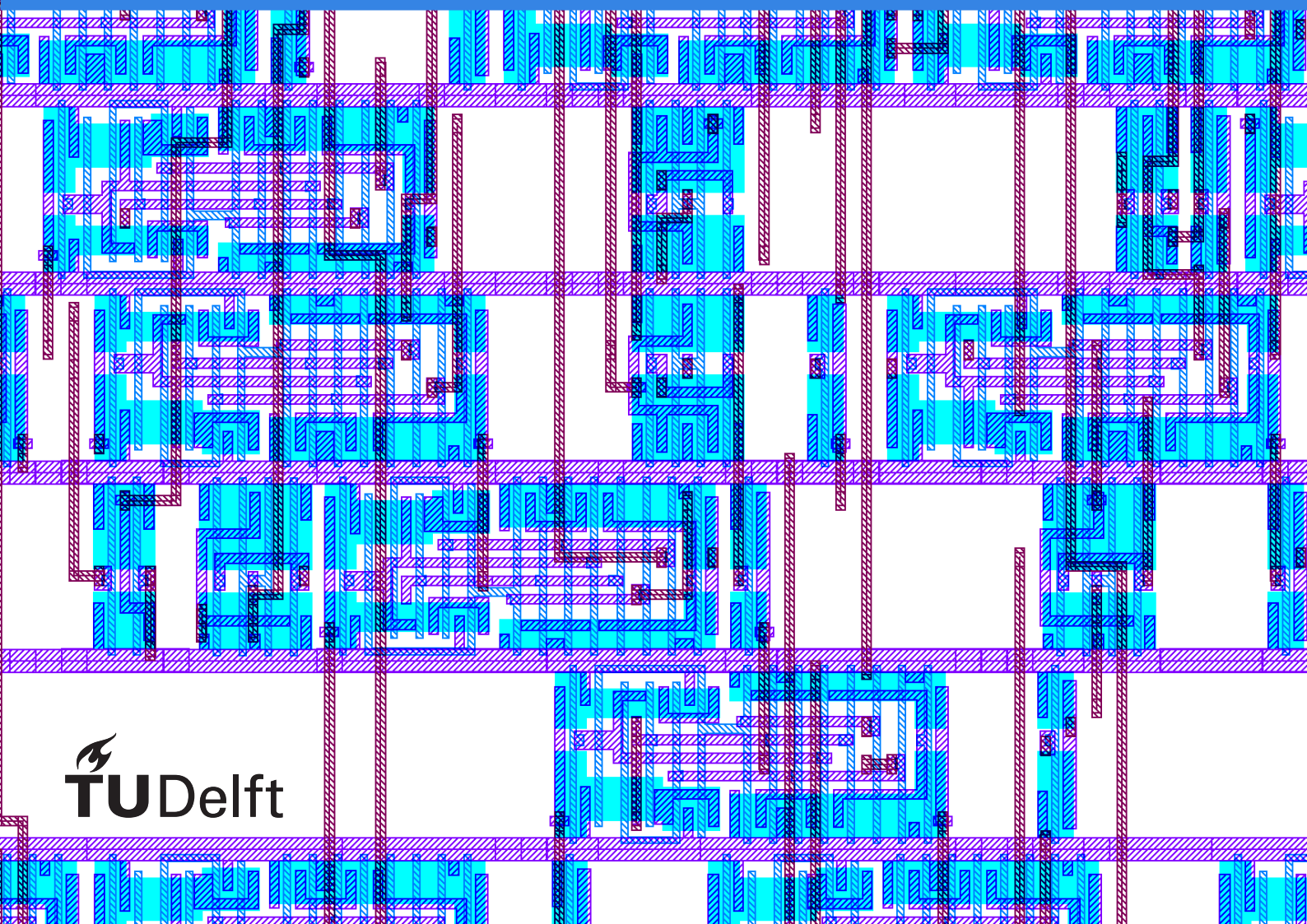
Dual Instruction Set on a Single Microarchitecure

Anne Stijns



TU Delft

Dual Instruction Set on a Single Microarchitecure

by

Anne Stijns

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday April 29, 2025 at 1:45 PM.

Project duration:	September 1, 2024 – April 29, 2025	
Thesis committee:	Prof. dr. ir. G. Gaydadjiev,	TU Delft, Thesis Advisor
	Assoc. Prof. dr. ir. L. M. Taouil,	TU Delft, Daily Supervisor
	Prof. dr. ir. K. Langendoen,	TU Delft

ARM Ltd. has no objections on the publication of the content of this thesis

TU Delft

Acknowledgments

Finishing this thesis was a challenging task for me, so I am relieved that others have lent me a hand. I would like to dedicate this page to thanking all the people who have helped me, both with technical advice and emotional support.

First, I would like to thank my supervisors Prof. Georgi Gaydadjiev and Prof. Mottaquiallah Taouil from Delft University of Technology. I appreciate the feedback I have gotten from them in our weekly meetings, their guidance during my project, and all the helpful comments I have received on the contents of this work as I was writing it.

Second, I would like to thank Andrea Kells and her technical colleagues at ARM for verifying that this thesis does not violate any intelectual property that ARM may have. I am happy to hear that you liked my work, and thank you for your cooperation in letting this thesis get published.

Third, thank you to my friends inside and outside the university for helping me. Thank you for keeping up my spirits, for keeping me company, and especially for the additional feedback.

Finally, I would like to thank my parents and my boyfriend for their immense emotional support and motivation. Thank you Joseph, Dory, and Jonathan. Without you, I never would have gotten this far.

It would not have been possible for me to finish this thesis without your support, thank you so much!

Anne Stijns
Delft, April 2025

Contents

Acronyms

ABI Application Binary Interface

ALU Arithmetic and Logic Unit

ASIC Application Specific Integrated Circuit

CISC Complex Instruction Set Computer

FPGA Field Programmable Gate Array

GCC The GNU Compiler Collection

HDL Hardware Description Language

IC Integrated Circuit

IoT Internet of Things

ISA Instruction Set Architecture

LUT Look Up Table

PFU Programmable Functional Unit

PIC Position Independent Code

RISC Reduced Instruction Set Computer

SoC System on a Chip

TNS Total Negative Slack

1 Introduction

This chapter introduces the concept of a single microarchitecture that can facilitate not one, but two instruction sets. It discusses the importance of facilitating multiple instruction sets, related work, and the main contributions. Section 1.1 presents the concept of an instruction set, and the difficulty of changing it, and the proposed solution. Section 1.2 describes the variety and history of instruction sets used in the industry. Section 1.3 reviews the state-of-the-art in facilitating multiple instruction sets and shows the challenges designers are facing. Section 1.4 covers the contributions of this thesis. Finally, Section 1.5 provides the outline of the rest of this work.

1.1. Motivation

Modern computer architectures are massively complex feats of engineering. They can be made to maximize performance, minimize power consumption, fit in small embedded systems, or some combination of these goals. The overwhelming complexity of contemporary processors, and the software they run, is too much for a single mind to handle. This is why various levels of abstraction are used in the software engineering and computer engineering disciplines. One such abstraction is the Instruction Set Architecture (ISA).

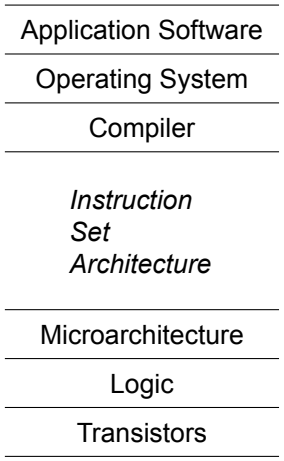
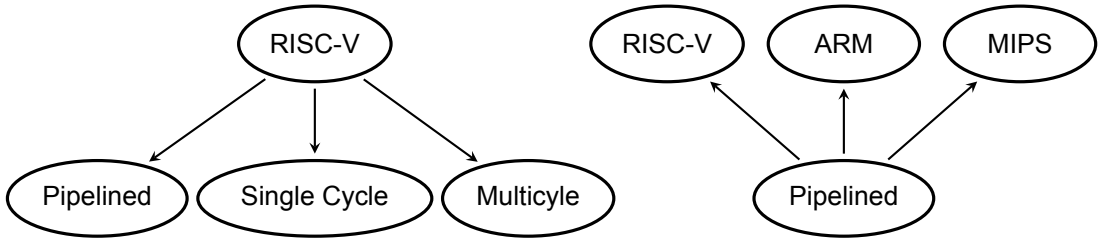


Figure 1.1: Layers of abstraction in Computer and Software Engineering

An instruction set can be seen as an agreement between software engineers and hardware engineers. Note how the ISA sits between the microarchitecture and compiler in Figure 1.1. In the words of David Chisnall, an early contributor to the RISC-V specification: “An instruction set is the lingua franca between compilers and microarchitectures” [9]. This agreement provides a target for computer engineers to implement, and a base for software engineers to build on top of. In this way, the ISA provides the foundation on which software is built.

The fact that an ISA is so fundamental to all software written for a computer system makes it difficult to change that ISA. For example, the x86 ISA, which is commonly used in desktops and laptops, has a legacy which goes back to the 8086 processor designed in 1978 [42, Chapter 3]. The fact that earlier software was written for the previous ISA creates an incentive to keep supporting that old ISA. This leads to design choices made in very different times affecting the architecture of microprocessors today.

Since the requirements of an ISA change over time, eventually it becomes beneficial to make breaking changes to the ISA. In fact, the personal computer manufacturing company Apple has done this twice in as many decades [20], [41]. First, the ISA switched from PowerPC to x86 in 2007, and then announced the architecture would switch from x86 to ARM in 2020. However, changing the ISA of a computer lineup is not a trivial decision. After all, all the software written for the old ISA will suddenly be incompatible. This is a reason why x86 has been extended by Intel and AMD for almost 50 years, instead of being replaced by a more modern instruction set.



(a) A single ISA implemented by multiple microarchitectures

(b) A single microarchitecture facilitating multiple ISAs

Figure 1.2: a) A single ISA can be implemented in multiple kinds of microarchitectures, b) so I propose a single microarchitecture could facilitate multiple ISAs. The abstraction level is lower at the bottom and higher at the top.

There is simultaneously a need to change the ISA from time to time, but also the cost of software compatibility that makes switching a difficult choice. Usually, a switch in ISA for a line of computers comes with many pieces of software being incompatible with the new hardware. For example, when Apple launched its second switch in 2020, Patrick Moorhead of Forbes wrote “While Apple may have promised miraculous levels of compatibility, it’s just not here yet for what I’m testing.” [30] It would benefit the computer industry if an ISA could be significantly modified without breaking software compatibility. Various solutions to ease the ISA transition have been proposed, and these serve as prior art to this thesis.

In the prior art, a microarchitecture always implements one ISA. But as illustrated in Figure 1.2, one ISA can be implemented in various different microarchitectures. For example, there are RISC-V microprocessors that are single cycle [23] multi-cycle [50], pipelined [10], or even superscalar [51]. So, what if we flip that perspective? If one ISA can be implemented in multiple microarchitectures, then why not have one microarchitecture that can facilitate multiple instruction sets? That is the motivating thought behind this thesis.

1.2. Background of Instruction Set Architectures

x86 is an example of a Complex Instruction Set Computer (CISC). Other architectures may be Reduced Instruction Set Computers (RISCs). The difference is in the instruction set, a CISC architecture has more complex instructions than a RISC architecture. According to Hennessy, this was a popular choice in the 1970s when compilers were less advanced, and there was a need to bridge the ‘semantic gap’ [18, Appendix K]. This semantic gap is the different way a programmer thinks about a computer compared to the computer engineer. An engineer may think in terms of registers and data busses, a programmer may think in terms of functions and data structures. As compilers began to fill in this semantic gap in the 1980s, architectures shifted to using less complex instructions for the sake of performance. ARM and MIPS are examples of an early RISC design, and RISC-V is a more modern example. RISC-V was introduced in 2014, while the first version of ARM was made between 1983 and 1985 [16, Chapter 2], [4]. The choice to make a RISC or CISC architecture is one of many.

Besides being RISC or CISC, any ISA design has trade-offs that lead to a choice that needs to be made. For example, an ISA might have a very irregular instruction encoding to favor code density, however this hinders ease of decoding. In 1991, Bhandarkar and Clark demonstrated this choice as a significant limitation of the VAX architecture compared to MIPS [6]. According to Hennessy and Patterson, this was mostly due to advances in technology [19]. Larger memories allowed programs to be larger, reducing the need for dense instructions. In other words, while reducing code size made sense when VAX was developed in 1977, it fell out of fashion over time.

The choices made in an ISA are not only a matter of time, but also of application. For example, he ARM Cortex architecture uses Thumb encoding. This is an encoding system that reduces code size, sacrificing ease of decoding. This is the exact trade off made for the VAX in 1977, and was considered by Bhandarkar and Clark to be obsolete in 1991 because code size was not as important in high performance computing. But the ARM Cortex is not made for high performance computing. It is used in embedded devices, where memory constraints are a concern even today. For example, the Embench Internet of Things (IoT) benchmarks explicitly do not use more than 64 kilobytes of memory because some systems used in IoT devices will not have more than 64 kilobytes of RAM available [5].

The original Embench benchmark even used as little as 16 kilobytes of ram [36]. So a design decision that may not be correct in high-performance computing, may still be correct in the context of embedded computing.

In general, some ISAs are better suited to certain applications than others. Ashish and Dean managed to extract as much as 20% more performance out of a multi-core system by using multiple ISAs [47]. They claim differences such as register pressure[1], code density, and accelerations for particular operations using e.g. SIMD or floating point instructions explained the difference in performance.

1.3. State of the Art in Facilitating Multiple Instruction Sets

Software Translation in Apple Rosetta 2

As said before, Apple has a history of changing the ISA of their computers. In order to facilitate this transition, Apple released a piece of software called Rosetta 2. This software can transpile an x86 program to ARM, allowing old software to run on the new ARM processors. Using this technology, the ISA can be changed completely while maintaining backward compatibility. As Moorhead wrote, Rosetta 2 was not a silver bullet to fix all compatibility issues [30]. However, Apple used more than just software for x86 compatibility.

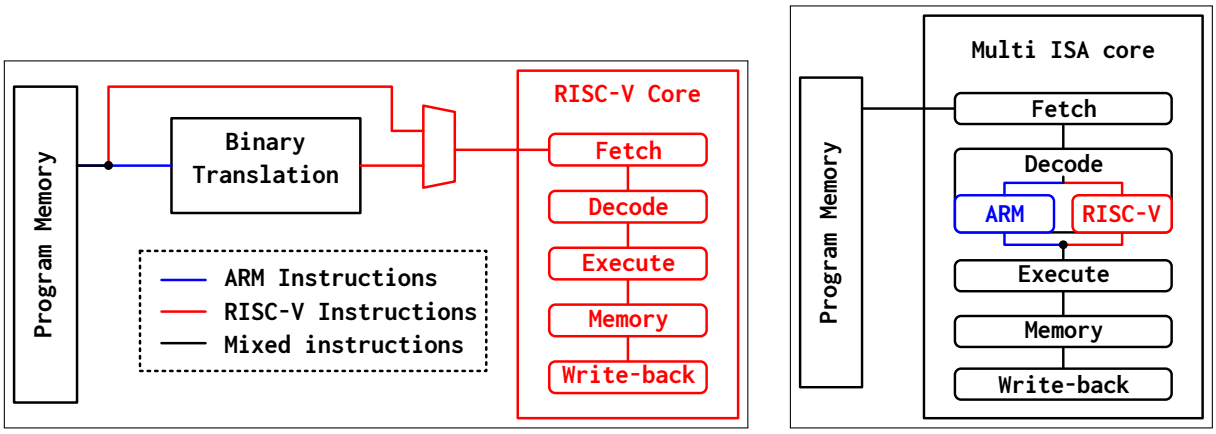
The processor cores Apple uses have some special extensions specifically designed to facilitate x86 emulation. J. Dougal has analyzed the code that Rosetta 2 produces to find these extensions [13]. This includes changes to the processor status flags, the floating point computation, and even having a special operating mode in which common instructions such as addition compute the same flags as they do on x86. These ARM processors also have support for a memory concurrency model that mimics x86. In other words, the ARM processor has been modified to be more like an x86 processor to facilitate the transition.

Software Translation in the Transmeta Crusoe

Binary translation software was also used by Transmeta for their Crusoe line of processors [11]. However, these processors had no hardware modification done to facilitate the transition. Since Transmeta was aiming for the laptop market at the turn of the century, power efficiency was a top priority [17]. However, the Crusoe was slower than other x86 laptop processors due to the overhead of transpiling every program. Ultimately, the Crusoe processors did not manage to disrupt the laptop market sufficiently, and the line was discontinued after the second generation [43].

The Crusoe processors used software to continuously optimize the program that was running. Transmeta called this technique Code Morphing. According to Klaiber, this was supposed to amortize the cost of translating the binary and allow the Crusoe processor to reduce execution time [25]. Kistler and Franz determined that although this approach may sometimes work, continuous software

[1] Register pressure refers to local variables being saved to memory because there are not enough architectural registers to store them in



(a) Hardware binary translation [21]

(b) Proposed multi ISA microarchitecture

Figure 1.3: Contrast between this work and hardware binary translation as used by RVAM16 [21]

optimization frequently cannot hit a break-even point [24]. Combined with the failure of Crusoe to stay in the x86 market for long, doubt can be cast on the efficacy of software binary translation on its own. This may explain why the engineers at Apple Inc. chose to modify the microarchitecture to fit the x86 instruction set as well as the ARM instruction set.

Hardware Binary Translation
Instead of transpiling instruction sets in software as with Rosetta and Crusoe, research has been done on transpiling instructions in hardware. For example, Madni *et al.* have conceptualized a microprocessor which translates x86 instructions to ARM [28]. This work is only a simulation framework; a full hardware design was left by the authors as future work. The authors therefore do not know the area overhead of their proposed design, although they were able to make an estimate of the performance overhead using a simulation.

Huang *et al.* took an extra step and designed a processor in hardware that can run both ARM and RISC-V binaries [21]. This processor is called RVAM16. The design of RVAM16 is shown in Figure 1.3a. Their design still uses a hardware translation system, in which ARM instructions are translated to RISC-V. RVAM16, has a microarchitecture that only implements RISC-V, with a frontend to translate ARM instructions to RISC-V. A major design constraint of RVAM16 was to keep the area comparable to an ARM Cortex M0. The design shown in Figure 1.3a is a full RISC-V microprocessor, with an additional hardware binary translation table. To make room for the table, the RISC-V processor would need to be smaller than the ARM processor it is replacing.

To maintain such a low area despite adding a binary translation table, Huang *et al.* made the choice to use a 16 bit ALU, even though ARM and RISC-V are 32-bit instruction sets. This caused a performance loss of 30% in running native RISC-V instructions. What's more, RVAM16 is about 25% slower at running ARM instructions than running RISC-V instructions, when running the Embench test suite. That is in addition to the 30% performance penalty of running RISC-V with a 16 bit ALU. In conclusion, making the microprocessor fit in a small area required a significant loss of performance. According to the authors, RVAM16 was designed to minimize resource consumption. It is not surprising that performance had to be sacrificed to achieve this. This leaves the question of how much area such a core would use if it had not sacrificed performance.

Heterogeneous ISA Multi Core Systems
So far, these solutions have aimed to support additional instruction sets for the purpose of compatibility, sacrificing performance. Heterogeneous ISA systems instead support multiple ISAs to extract performance. In 2019, Venkat *et al.* have created a multi core processor in which some cores implement a certain subset of the x86 ISA [46]. This is called a heterogeneous ISA multi core processor, as different cores have different ISAs. By scheduling programs for the appropriate core, a performance gain of 19% could be reached.

Later, in 2024, Venkat *et al.* took this one step further and created a heterogeneous ISA processor using entirely different ISAs [47]. Instead of using variations of x86, this multi core processor used cores that ran x86, ARM thumb, and DEC Alpha. This system also achieved a performance gain of 21%, similar to the one they made in 2019. In these cases, the cores could each only run one ISA. The multi core system facilitated multiple ISAs by having some cores that implement each instruction set. Whether performance could be gained depends on how well the task set matches with the resources the processor has available. Venkat *et al.* used a design space exploration method to find an optimal configuration for the benchmark they used.

1.4. Contributions

This thesis features Combi, a single microarchitecture that can facilitate both ARM and RISC-V binaries. In contrast to RVAM16, the proposed Combi microarchitecture in Figure 1.2b will execute both ISAs natively. That is, instructions are not translated from one ISA to another, but rather a microarchitecture is made which can facilitate instructions from either ISA. In contrast to microarchitectures that implement only one ISA, meaning they were never intended to run another instruction set, this processor can facilitate multiple ISAs without translating into a primary ISA. This microarchitecture is designed to run binaries of either ISA from the ground up.

Since the aim is to facilitate multiple instruction sets, a choice needs to be made. What instruction sets will be facilitated? Four ISAs came to mind: x86, ARM, MIPS, and RISC-V. These are commonly

Table 1.1: Instruction count of various versions of the ARM and RISC-V ISA

ISA	Unique instructions
ARMv7A	189
ARMv6	116
ARMv5	72
ARMv4	**63**
RV32I	37
RV32IM	**45**

used ISAs and have an extensive library of documentation. However, x86 is too big to be reasonably implemented in the span of a masters thesis. Consider that Shanley's book on the x86 ISA is over 1500 pages long [42]. Also, MIPS and RISC-V are very similar instruction sets. A microarchitecture that facilitates both would not be much different from a microarchitecture that was made to implement only one. Therefore, the aim of this work is to make a single microarchitecture that facilitates both ARM and RISC-V instructions.

ARM is not nearly as complex as x86, but the ARM ISA has grown in size as newer versions were made. The amount of unique instruction mnemonics each version has is shown in Table 1.1. Two versions of the RISC-V ISA are also shown - RV32I and RV32IM. We can see that every version of ARM since version 4 has had more instructions than the basic RISC-V instruction set. To make the combined microarchitecture in a reasonable amount of time, the somewhat dated version 4 instruction set will be implemented. For reference, microprocessors implementing ARMv4 were already being made in the year 2000 [2].

In this thesis, we will compare and contrast the RISC-V ISA and ARM ISA. Combi is compared to the state of the art RISC-V microarchitectures. Combi can also be configured to function as a pure RISC-V or ARM processor, which allows insights to be gleamed into the overhead of combining multiple instruction sets on a single microarchitecture.

The design of a single microarchitecture that facilitates both RISC-V and ARM binaries
We create a single pipelined microarchitecture that can facilitate two instruction sets, named Combi. The five stage pipeline can have both ARM and RISC-V instructions in flight at the same time, and can forward results from one instruction to another across ISA boundaries. There is no pipeline flush or stall required to switch the instruction set. In fact, the entire microarchitecture can seamlessly morph between being a RISC-V processor and an ARM processor.

A comprehensive and systematic classification of the RISC-V and ARM instructions
To inform the design of a microarchitecture that is able to facilitate both ISAs, we split the RISC-V and ARM instruction sets into three categories. These are register instructions, load/store instructions, and branches. We compare and contrast the differences and similarities between the ISAs using this classification as a framework.

By splitting ARM and RISC-V instructions across the same instruction type, we can determine which instructions one has that the other has no equivalent to. We show that even though RISC-V does not have some instructions that ARM has, hardware meant for instructions exclusive to ARM can be used to facilitate certain RISC-V instructions as well.

Evaluation of the area and performance overhead of combining instruction sets
We synthesize Combi both to an FPGA realization and an Integrated Circuit (IC). In addition to being able to morph between instruction sets, Combi can be configured at build-time to only implement ARM or only implement RISC-V. The three versions of Combi - combined ISA, RISC-V, and ARM - are compared in terms of area and maximum clock speed to evaluate the chip area and performance cost of facilitating more than one instruction set. Combi is also compared to a state of the art RISC-V microprocessor, Ibex [7]. This is done to show that Combi is not excessively large or slow when configured to run only the RISC-V ISA.

A novel methodology for cross-ISA calls
In addition to comparing the instruction sets themselves, we discuss how the instruction stream can be handed over from one instruction set to another. We show how the microarchitecture can make

calling libraries written in foreign instruction sets completely transparent to the caller. Furthermore, this does not take considerable area, nor does it require rewriting anything in the library or binary. We present a methodology for changing the instruction set during the execution of a program, which includes hardware methods, software methods, and a combination thereof.

1.5. Thesis Overview

The rest of this work is organized into five chapters. The first two chapters describe the two ISAs that are implemented. These chapters each focus on one ISA, but instructions from both ISAs are classified in the same way. The middle two chapters explain the methodology of switching between the instruction sets and the design of a microarchitecture that facillitates the instruction sets. The final chapter concludes this thesis.

The following list provides a detailed description of the contents of each chapter:

- In Chapter 2, the RISC-V ISA is described according to a general classification. This includes the register set, the instruction set, and the Application Binary Interface. The same classification is used in Chapter 3 to describe ARM instructions, which demonstrates the similarities and differences between the ISAs The instructions that are implemented in Combi are discussed comprehensively. The ABI is also described, which will be used in Chapter 4 as an example of methodology for calling functions in a foreign ISA.
- In Chapter 3, the ARM ISA is described comprehensively according to the same classification as the RISC-V ISA. As in Chapter 2, this is organized as first discussing the register set, then the instruction set, and finally the Application Binary Interface. The ARM ABI is described to serve as an example of the methodology for cross-ISA function calls.
- In Chapter 4, the approach used to combine these ISAs to a combined ISA is shown. The instructions of RISC-V and ARM are contrasted by category, with a focus on the similarity in instructions. The matter of disambiguating Instruction Set Architectures is discussed here. Finally, a methodology for calling procedures from a different ISA is presented, including hardware and software methods.
- In Chapter 5, it is shown how this combined ISA is implemented. We start from a generic pipelined microarchitecture. Then we extend it with the necessary features required to facilitate both instruction sets. Special attention is given to ensuring the added features are used by both instruction sets as much as possible. In other words, the aim is to make this microarchitecture unbiased toward any ISA. Also, the implementation is compared to the state of the art.
- In Chapter 6, this thesis is concluded. A summary of the contributions is provided and future work is discussed.

2

RISC-V Instruction Set Architecture

To move toward implementing Combi, we start with systematically breaking down the RISC-V ISA. In Chapter 3, the same is done for the ARM ISA. By analyzing both ISAs in the same way, the road to a full picture of the Combi microarchitecture will be paved. Section 2.1 explains the modular naming scheme of RISC-V, and provides an overview of the instruction encoding. Section 2.2 shows the register set of RISC-V. Sections 2.3, 2.4, and 2.5 go into detail of all RISC-V instructions according to a categorization that will also be used for ARM: instructions that operate on registers, instructions that interact with memory, and instructions that modify control flow, in that order. Finally, Section 2.6 touches on the ABI of RISC-V, which will be used to construct a methodology for function calls across ISAs in Chapter 4.

2.1. Modularity of the RISC-V Instruction Set Architecture

To understand the implementation of Combi, an understanding of the ISAs it facilitates is necessary. Therefore, this section describes the RISC-V instruction set. In particular, the RV32IM instruction set according to the RISC-V instruction set manual is described [49]. The RISC-V instruction set is designed to be modular and extensible. RISC-V CPU's must implement the core part of the instruction set and may implement any combination of extensions. The core instruction set is called RV32I, and extensions are usually single letters such as 'M' for multiply/divide instructions, 'A' for atomic instructions[1], and 'C' for compressed instructions[2]. Combi facilitates the base integer RISC-V instructions (RV32I) as well as the Multiply extensions (RV32M). The combined instruction set is called RV32IM.

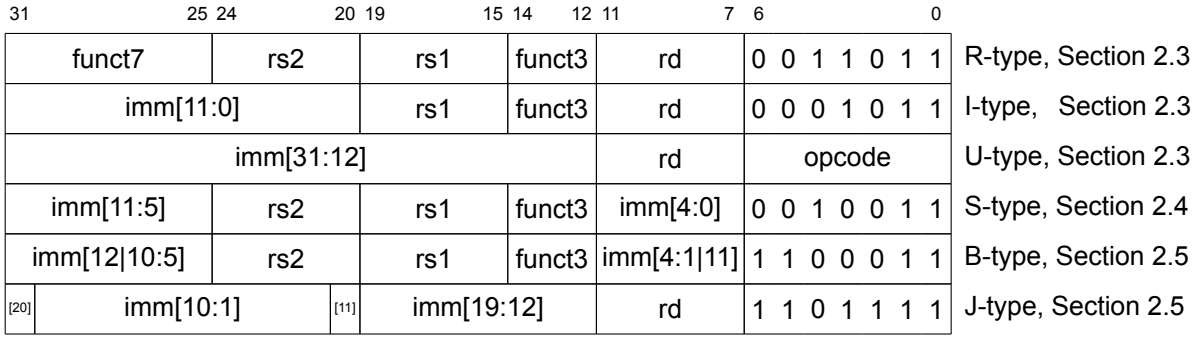
As explained in Chapter 3, the ARM ISA also has a multiplication instruction. Unlike RISC-V, the ARM ISA was not designed to be modular. To be compatible with ARM, a microarchitecture must implement the full ISA, including the multiply instructions. To be compatible with ARM binaries, Combi will have to facilitate multiplication in any case. Therefore, implementing the multiplication extension will be trivial.

The RV32IM instruction set also has instruction encodings for division and remainder operations, but these will not be implemented in Combi. This technically means that the full M specification is not facilitated, but the Zmmul subset is. Therefore, it is more accurate to say that Combi facilitates the RV32IZmmul instruction set.

All RISC-V instructions in the RV32IM ISA are encoded as 32 bits. At the start of every section describing an instruction, the encoding of that instruction is shown as a long bar separated into segments of bits, with the bit positions labeled above the bar. Many instructions in RISC-V follow a similar encoding. For example, most instructions that operate on two registers are encoded in a similar way. In RISC-V, this is called the R-type encoding. The name of the encoding type will be next to the bar, or sometimes the instructions that use an encoding will be next to it.

[1] The atomic instructions are used to ensure thread safety in concurrent programs
[2] Compressed instructions are 16-bit instructions that can be used to optimize code density

Table 2.1: Overview of the RV32I instruction types



RISC-V Encoding Overview

There are six instruction types in the base 32-bit RISC-V ISA. These are shown in Table 2.1. In general, the first seven bits of an instruction type are a unique opcode. This is used to identify the type of instruction, which is then used to interpret the meaning of the other 25 bits. The funct7 and funct3 fields then identify a more specific instruction from the instruction type. For example, the `add` instruction is an R-type instruction with the funct3 and funct7 field set to all zeroes.

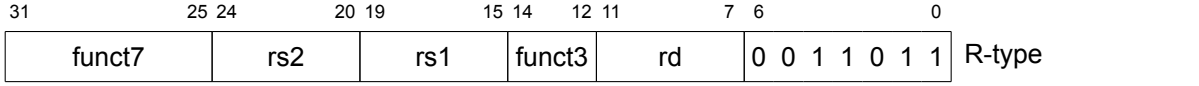
The R-type instructions are used for operations that calculate a result using two registers and store it in a third register. The I and U-type instructions instead use a constant to modify a register. Together, these form the kinds of instructions that will be discussed in Section 2.3. The S-type instructions are used for storing to memory, and loading from memory uses an I-type instruction. These instructions are covered in Section 2.4. Finally, the B and J-type instructions are used for conditional branches and unconditional jumps. These are explained in in Section 2.5.

2.2. Registers

The RV32I specification has 32 registers, called `x0` to `x31`. The `x0` register is special: writing to this register has no effect and reading from this register will always result in zero. All other registers are general purpose, all instructions can use any register. Therefore, an instruction that uses a register will specify it using a 5 bit field.

`x0` can also be used in any instruction. This can be useful in some circumstances. For example, adding an immediate to `x0` is the idiomatic way to load an immediate into a register, according to the RISC-V ISA manual [49]. `x0` can also be used to discard the register result of an instruction. For example, the RISC-V ISA manual recommends using the Jump and Link instruction (Section 2.5.1) for jumping without saving the return address [49]. In this case, `x0` is used as the destination register, and the return address is not saved.

2.3. Register Instructions



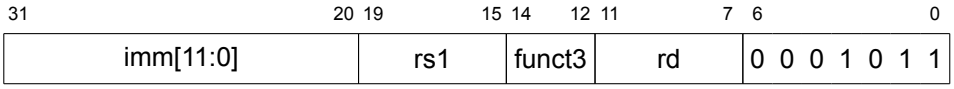
Register instructions perform an operation on two registers and store the result in another register. The two source registers are `rs1` and `rs2`, and the destination register is `rd`. Since there are 32 registers, the source and destination fields are 5 bits wide. These instructions are called R-type instructions and are listed in Table 2.2.

A 64 bit multiplication is done in two instructions: `mulh` to compute the most significant 32 bits, and `mul` to computer the least significant 32 bits. There are 3 variants of `mulh` for signed or unsigned operands. `mul` can be used for signed or unsigned arithmetic, because the lower 32 bits of the product are independent from the sign of the 32-bit multiplicand or multiplier [35, Section 9.4]. Signed or unsigned multiplication has the same result in the lower 32 bits, but a different result in the upper 32 bits. That is why there are special instructions for signed multiplication when computing the upper 32 bits, but not for computing the lower 32 bits.

Table 2.2: RV32IZmmul R-Type instructions

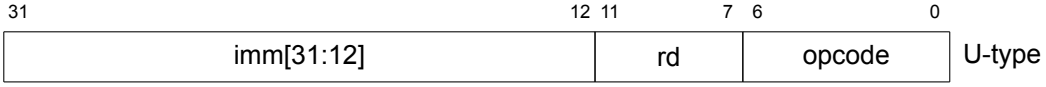
Instruction	funct3	funct7	Operation
`add rd, rs1, rs2`	000	0x00	`rd = rs1 + rs2`
`sub rd, rs1, rs2`	000	0x20	`rd = rs1 - rs2`
`sll rd, rs1, rs2`	001	0x00	`rd = rs1 << rs2`
`slt rd, rs1, rs2`	010	0x00	`rd = rs1 < rs2 ? 1 : 0`
`sltu rd, rs1, rs2`	011	0x00	`rd = rs1 <unsigned rs2 ? 1 : 0`
`xor rd, rs1, rs2`	100	0x00	`rd = rs1 ^ rs2`
`srl rd, rs1, rs2`	101	0x00	`rd = rs1 >> rs2`
`sra rd, rs1, rs2`	101	0x20	`rd = rs1 >>arith rs2`
`or rd, rs1, rs2`	110	0x00	`rd = rs1 | rs2`
`and rd, rs1, rs2`	111	0x00	`rd = rs1 & rs2`
`mul rd, rs1, rs2`	000	0x01	`rd = (rs1 * rs2)[31:0]`
`mulh rd, rs1, rs2`	001	0x01	`rd = (signed(rs1) * signed(rs2))[63:32]`
`mulhsu rd, rs1, rs2`	010	0x01	`rd = (signed(rs1) * rs2)[63:32]`
`mulhu rd, rs1, rs2`	011	0x01	`rd = (rs1 * rs2)[63:32]`

2.3.1. Immediate Instructions



Immediate instructions are like Register instructions, except that a 12-bit immediate is used instead of `rs2`. The instructions encoding is shown in Table 2.3. The 12-bit immediate is sign extended, even for non-arithmetic instructions. Even `sltiu` (Set if Less Than Immediate Unsigned), which is explicitly an unsigned comparison, will first sign extend the 12-bit immediate to 32 bits before the unsigned comparison, as pointed out explicitly in the manual [49]. There is no instruction to multiply with an immediate. There is also no instruction to subtract an immediate. However, an addition with a negative immediate is possible.

2.3.2. Upper immediate Instructions



There are two instructions that use the U-type format. These are `lui` and `auipc`. The operation of these instructions is shown in Table 2.4. The U-type format has the largest immediate size of any RISC-V instruction type, 20 bits long. When combined with an immediate type instruction (Section 2.3.1), these instructions can load an arbitrary 32-bit constant into a register using two instructions.

`auipc` will load the immediate value relative to the program counter, which is useful for Position Independent Code (PIC). Position Independent Code means that a program will execute correctly regardless of where in memory it is loaded. Consider a library that has 8 kilobytes of code followed by 4

Table 2.3: RV32IZmmul I-Type instructions

Instruction	funct3	funct7	Operation
`addi rd, rs1, imm`	000	0x00	`rd = rs1 + imm`
`slli rd, rs1, imm`	001	0x00	`rd = rs1 << imm`
`slti rd, rs1, imm`	010	0x00	`rd = rs1 < imm ? 1 : 0`
`sltiu rd, rs1, imm`	011	0x00	`rd = rs1 <unsigned imm ? 1 : 0`
`xori rd, rs1, imm`	100	0x00	`rd = rs1 ^ imm`
`srli rd, rs1, imm`	101	0x00	`rd = rs1 >> imm`
`srai rd, rs1, imm`	101	0x20	`rd = rs1 >>arith imm`
`ori rd, rs1, imm`	110	0x00	`rd = rs1 | imm`
`andi rd, rs1, imm`	111	0x00	`rd = rs1 & imm`

Table 2.4: RV32IZmmul U-type instructions

Instruction	opcode	Operation
`lui rd, imm`	0x37	`rd = imm<<12`
`auipc rd, imm`	0x17	`rd = pc + (imm<<12)`

kilobytes of data, as shown in Figure 2.1.

Now consider if this library is loaded in a program that is also 8 kilobytes large, like shown in figure 2.2. To make space for the program, the library has been moved and the library's data now resides at address `0x00004000` instead of `0x00002000`. This means that if the library was loading data from `0x00002000` directly, it will now read meaningless data instead. If it was instead using addresses relative to the program counter, this problem would not happen. This is because the library code was also moved `0x2000` bytes up. The relative difference between the address of the library code and the library data has not changed. In conclusion, using addresses relative to the program counter allows binaries to be relocated in memory, which is called Position Independent Code. This can be achieved by using the `auipc` instruction.

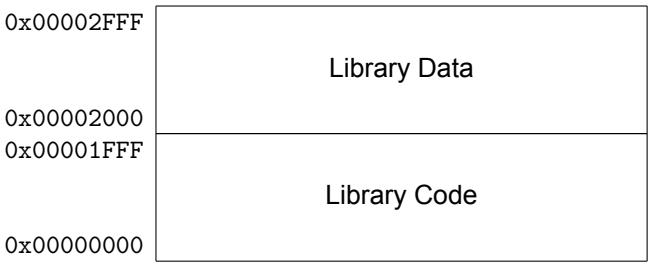


Figure 2.1: Example library memory map

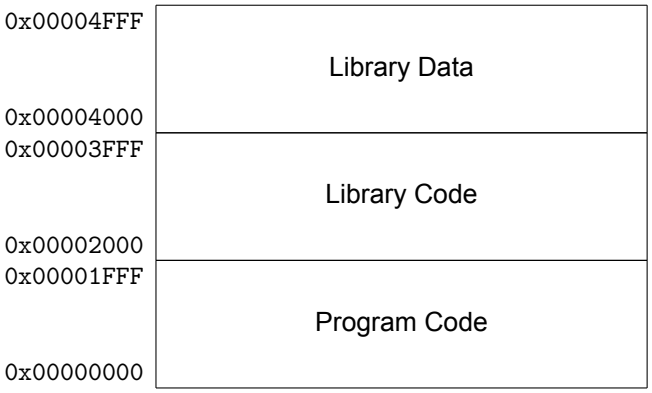


Figure 2.2: Example memory map of library loaded within an application

2.3.3. Immediate Encoding

As mentioned in various other sections, the encoding of immediates in RISC-V may seem strange at first glance. However, there is a design choice behind the way immediates are encoded. The encoding is done the way it is to make sure as many bits of the immediate are the same as possible. Figure 2.3 shows an overview of how every bit of the immediate is stored for all RISC-V instruction formats. Note that many bits are in the same position, even in different instruction types. For example, bits in the B-type instruction were shifted to the left to align with bits in the S-type, because the B-type immediate is shifted one position to the left in branch instructions. To understand why the B-type immediate should be shifted by one, refer to Section 2.5.

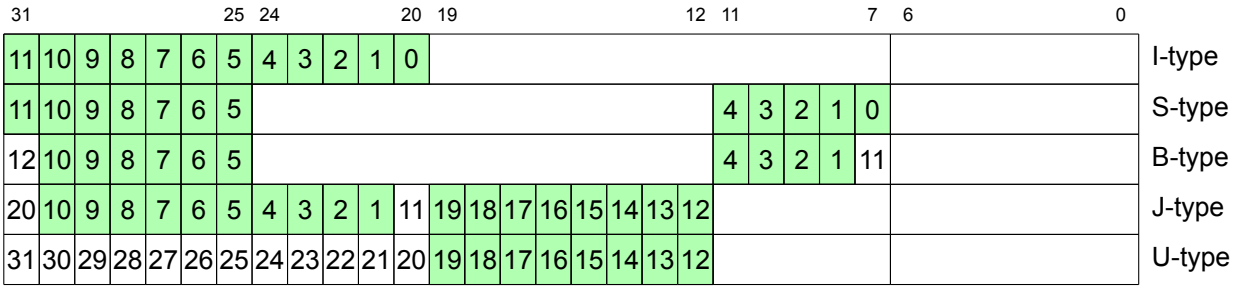
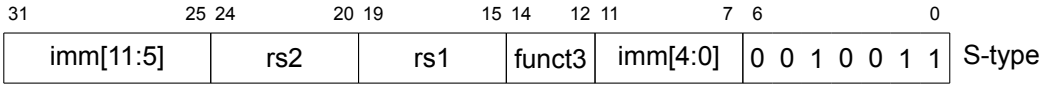


Figure 2.3: Immediate encodings for RISC-V instruction formats. Immediate bits with same position are highlighted in green.

2.4. Load/Store Instructions

Loads and stores in RISC-V follow the same basic addressing mode. The memory address is a register plus a signed 12-bit immediate offset. The ARM ISA has many more options than this, as shown in Section 3.4. The next sections describe the load and store instructions in more detail.

2.4.1. Store Instructions



Store instructions use a sign extended 12-bit immediate like immediate instructions. However, the immediate is encoded differently from an I-type instruction. The lower 5 bits are stored in the place where I-type instructions place the destination register. The store instructions do not have a destination register, since the data is stored to memory. The upper 7 bits are in the same place as the I-type instructions. There is only one addressing mode: a base register offset by an immediate. As shown in Table 2.5, there are three instructions for storing values: storing an 8-bit byte, a 16-bit halfword, or a 32-bit word.

Table 2.5: RV32IZmmul S-Type instructions

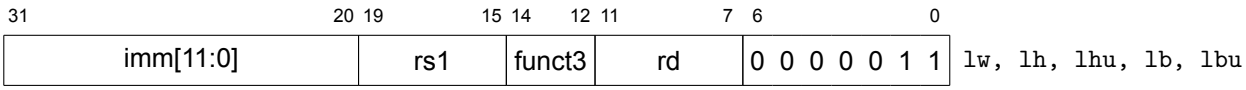
Instruction	funct3	Operation
`sb rs2, imm(rs1)`	000	`mem[rs1 + imm][7:0] = rs2[7:0]`
`sh rs2, imm(rs1)`	001	`mem[rs1 + imm][15:0] = rs2[15:0]`
`sw rs2, imm(rs1)`	010	`mem[rs1 + imm][31:0] = rs2[31:0]`

Word and halfword stores may not be aligned to their boundary. That is, the address may not be a multiple of the size of a word or the size of a halfword. The RISC-V ISA manual document allows an execution environment (like a CPU) to do one of two things if this happens, the CPU may:

- raise an exception;
- perform the load/store correctly, with a possible performance penalty.

As an example of how compilers handle this ambiguity, The GNU Compiler Collection (GCC) will emit aligned stores and loads depending on the target CPU [44]. If the CPU hardware does not support fast unaligned loads/stores, GCC will make sure that all memory accesses are aligned to the correct boundary by default. This behavior can be changed with a compiler flag.

2.4.2. Load Instructions



Load instructions use the same encoding as immediate type instructions, but with a different opcode. That is, bits 6 down to 0 have a different constant value. Unaligned loads are handled in the same way as unaligned stores, see Section 2.4.1.

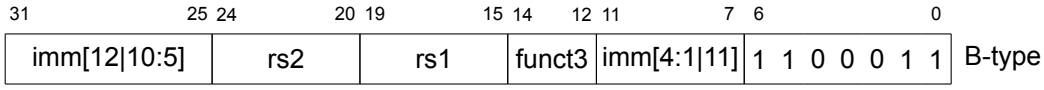
What is different from store instructions is the ability to sign extend a loaded byte or halfword. As shown in Table 2.6, there are two more load instructions than store instructions, `lbu` and `lhu`. The value stored in the register will be the value of the signed data in memory, sign extended to 32 bits. For example, if the value `0xFF` (-1) is stored as a byte in memory at address `0x100`, executing `ld x1, 0x100` will load the `x1` register with the value `0xFFFFFFFF` (-1). The `lbu` and `lhu` instructions do not sign extend the loaded value and instead extend the value with zeroes, in accordance with the ISA manual [49]. Returning to the example, `lbu x1, 0x100` will load the `x1` register with the value `0x000000FF` (255) instead of `0xFFFFFFFF` (-1), clearing bits 8 through 31.

Table 2.6: RV32IZmmul load instructions

Instruction	funct3	Operation
`lb rd, imm(rs1)`	000	`rd[7:0] = se(mem[rs1 + imm][7:0])`*
`lh rd, imm(rs1)`	001	`rd[15:0] = se(mem[rs1 + imm][15:0])`*
`lw rd, imm(rs1)`	010	`rd[31:0] = mem[rs1 + imm][31:0]`
`lbu rd, imm(rs1)`	100	`rd[7:0] = mem[rs1 + imm][7:0]`
`lhu rd, imm(rs1)`	101	`rd[15:0] = mem[rs1 + imm][15:0]`

*`se()` means sign extended.

2.5. Branch Instructions



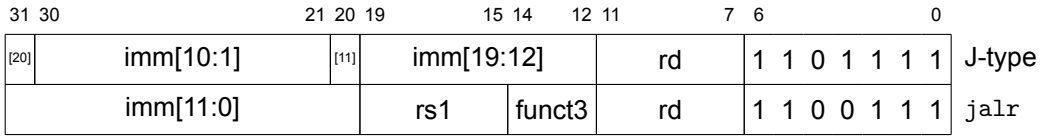
Branch instructions have an 12-bit immediate, which is shifted to the left by one. This is because instructions in RISC-V are always aligned to two bytes. All the instructions in this chapter are 4 bytes long, but the compressed instruction extension (RV32IC) has instructions that are 2 bytes long. Since the smallest instructions are 2 bytes long, all instructions are aligned to a 2 byte boundary. Therefore, any branch must be a multiple of 2 bytes away.

The 12-bit immediate should be shifted to the left by one, which is why it is encoded in the way it is. Bits 10 down to 1 are stored in the same place as the S-type instruction, with bits 11 and 12 stored in the remaining space. This is done to reduce complexity in the decoder, according to the RISC-V ISA manual [49]. The immediate is also sign extended to allow branches forward and backward. All branches are listed in Table 2.7. There are branches for equality, signed arithmetic relations, and unsigned arithmetic relations.

Table 2.7: RV32IZmmul B-Type instructions

Instruction	funct3	Operation
`beq rs2, rs1, imm`	000	`if(rs1 == rs2) pc += imm<<1`
`bne rs2, rs1, imm`	001	`if(rs1 != rs2) pc += imm<<1`
`blt rs2, rs1, imm`	100	`if(rs1 <signed rs2) pc += imm<<1`
`bge rs2, rs1, imm`	101	`if(rs1 >=signed rs2) pc += imm<<1`
`bltu rs2, rs1, imm`	110	`if(rs1 < rs2) pc += imm<<1`
`bgeu rs2, rs1, imm`	111	`if(rs1 >= rs2) pc += imm<<1`

2.5.1. Jump Instructions



RISC-V has two kinds of jump instruction: `jal` and `jalr`. `jal` is encoded as a J-type instruction, and `jalr` as an I-type. The immediate of the J-type instruction is shifted by one, and therefore is stored in a similar way to branch instructions to reduce decoder complexity [49]. Note that bits 10 down to 1

Table 2.8: RV32IZmmul Jump instructions

Instruction	type	funct3	Operation
`jalr rd, imm`	I	000	`rd = pc+4; pc = (rs1+imm) & ~0x01`
`jal rd, imm`	J		`rd = pc+4; pc += imm<<1`

are stored in the same position as the I-type instruction, and bits 19 down to 12 are stored in the same position as the U-type instruction. This might seem jarring, but the method to this peculiar immediate encoding is explained in Section 2.3.3.

`jal` will make the program counter jump to a position specified by a 20-bit offset. This offset is sign extended, and shifted by one for the same reason as branch instructions, see Section 2.5. The address of the next instruction (`pc+4`) is also stored in a register for the sake of function calls. If this is not the needed the RISC-V ISA manual recommends using register `x0`, the zero register, as a destination [49].

`jalr` performs the same operation as `jal`, but using a register as an offset instead of a 20-bit immediate. The manual explains that this allows jumps to anywhere in the 32-bit address space when combined with the `lui` or `auipc` instructions [49]. `lui` and `auipc` are explained in Section 2.3.2.

2.6. Application Binary Interface

As explained in Section 2.2, all registers are interchangeable from an ISA perspective. However, the RISC-V ABI does assign special meanings to these registers [8].

The list of registers, as well as their role in the RISC-V ABI, are shown in Table 2.9. Two registers are assigned to hold return values, these are `a0` and `a1`. Furthermore, eight registers are assigned as function arguments, these are `a0-a7`. This is different from the ARM ABI, and the way to consolidate this difference will be shown in Section 4.6.

When a function is called, this function will use some registers as part of its execution. The value of these registers may need to be restored by either the function called (callee) or the program that called the function (caller). Which one of the caller or callee is responsible for saving the value of a register at a function call is shown in the 'Saved' column of Table 2.9.

Function arguments that do not fit into `a0-a7` are put on the stack. The stack pointer (`x2`) will point to the first argument, with other arguments in the memory addresses above. The stack pointer must be aligned to a 16-byte boundary when a function is called. However, according to the RISC-V ABI, it is not necessary to keep the stack aligned within the function [8]. The only requirement is that the stack should be aligned when a ABI compliant function is called. The ABI does not state a reason for why the stack should be aligned, but it might have to do with simplifying caching the stack. In short, cache lines are usually aligned to a power of two. If the head of the stack is also aligned to a power of two, then it neatly fills a cache line. But if the stack is not aligned, it will occupy two lines of cache. Since the head of the stack is frequently accessed, optimizing those accesses to one cache line can be beneficial.

Note that three fo the 32 RISC-V registers do not have a defined saving convention. These are `x0`, `x2`, and `x3`. As discussed above, `x0` is immutable and will always read a value of `0`. Therefore, it does not need to be saved. `x2` and `x3` have a special function in the RISC-V ABI. In compliance with the ABI, these registers should be set only once: in the program's prelude or the thread's prelude respectively. Their value points to the location of the global variables and the global variables of the thread respectfully, plus `0x800` (2048). The exact meaning of this ABI function is outside the scope of this thesis; suffice to say, the values of these registers should not change in the normal execution flow of the program.

Table 2.9: RISC-V registers and usage in the RISC-V ABI [8]

Register	ABI name	ABI Function	Saved
x0	zero	always zero	
x1	ra	return address	caller
x2	sp	stack pointer (full)	callee
x3	gp	global pointer	
x4	tp	thread pointer	
x5-x7	t0-t2	temporary registers	caller
x8-x9	s0-s1	saved registers	callee
x10-x11	a0-a1	function arguments / return values	caller
x12-x17	a2-a7	function arguments	caller
x18-x27	s2-s11	saved registers	callee
x28-x31	t3-t6	temporaries	caller

3

ARM Instruction Set Architecture

This chapter will describe the ARM ISA using the same framework as was used in Chapter 2. Then, the two will be contrasted in Chapter 4, to serve as the basis for the Combi microprocessor. Section 3.1 provides an overview of the ARM instruction encoding. Section 3.2 will explain the special properties of ARM registers. Sections 3.3, 3.4, and 3.5 will go into detail about the ARM instructions in the same order as the RISC-V instructions: instructions that operate on registers, instructions that interact with memory, and instructions that modify control flow. Finally, Section 3.6 touches on the ABI of ARM, to be used in the methodology of switching ISAs in Chapter 4.

The instructions of the ARM ISA are 32 bits long, like RISC-V. The ARMv4 ISA also has a set of 16 bit instructions, called Thumb [1]. The Thumb instructions serve a similar purpose in ARM as the Compressed instructions do in RISC-V, however neither are implemented in this work.

At the start of every section describing an instruction, the encoding of that instruction is shown as a long bar separated into segments of bits, with the bit positions labeled above the bar. Some instructions in ARM follow a similar encoding. For example, most instructions that operate on two registers are encoded in a similar way. In ARM documentation, these is called data processing instructions. This name will be next to the bar describing how data processing instructions are encoded. Most other instructions use a unique encoding, and in this case the mnemonic of the instruction will instead be next to the encoding.

3.1. ARM Instruction Set Overview

In Section 2.1, the tidy organization of the RISC-V ISA was shown. Unlike RISC-V, the ARM ISA has evolved over time, with new instructions being added every version. As the product of several revisions, the ARMv4 ISA is less organized than the RISC-V one. Nevertheless, an attempt is made to provide an overview in this section. This will aid in understanding the context of the following sections, which go into detail on the instructions.

An overview of the instruction types is shown in Table 3.1. Some bits are left unspecified, since their interpretation is strongly tied to the type of instruction they represent. The meaning of these bits is often represented by a single letter, which would have no meaning in the context of this overview. Therefore, refer to the specified section to see the full encoding of an instruction type.

The most significant bits of all instructions specify a predicate for conditional execution. For example, an instruction can be specified to not execute if the result of the previous instruction was zero. The most significant bits after that specify the kind of instruction that will execute, with some exceptions. These instruction types were later additions to the ARM ISA, which explains the nonstandard encoding they use. The data processing, load/store, load/store multiple, and branch instructions follow the general convention that the most significant bits dictate the type of instruction. The multiplication and special load/store instructions do not follow this convention. Instead, an instruction of this type will have bits 7 and 4 specifically set to '1'. This happens to not ever occur in the Data Processing and Load/Store instructions, which is necessary for the encoding to be unique. If this was not the case, Multiplication instructions would have the same binary encoding as other Data Processing instructions.

Table 3.1: Overview of ARM instructions

31–28	27–26	25	24–21	20	19–16	15–12	11–8	7–4	3–0	
cond	0 0		opcode		Rn	Rd	operand 2*			Data Processing, Sec. 3.3
cond	0 1				Rn	Rd	operand 2*			Load/Store, Sec. 3.4
cond	1 0 0				Rn	register list				Load/Store Multiple, Sec. 3.4
cond	1 0 1		offset							Branch, Sec. 3.5
cond	0 0 0 0				Rd	Rn	Rs	1 0 0 1	Rm	Multiplication, Sec. 3.3
cond	0 0 0				Rn	Rd	imm[7:4]	1 op 1	Rm	Special Load/Store, Sec. 3.4

*bits seven and four of operand 2 are never both 1, which ensures it does not overlap with other instructions. See Figure 3.1

The instruction types were given a name in this thesis that matches with the category of instructions they encode. The data processing and multiplication instruction fit the description of instructions that operate on registers, and are therefore discussed in Section 3.3. The load/store, load/store multiple, and special load/store instructions all operate on memory and are explained in Section 3.4. The special load/store instructions are given that name because of their unique encoding. Finally, the branch instruction is the only ARM instruction that is specifically designed to change control flow, and is discussed in Section 3.5.

3.2. Registers

The ARMv4 ISA has 16 general purpose registers, named `r0`-`r15` [1]. `r14` is also called the link register, and it has a special interaction with the branch and link (`BL`) instruction. The `BL` instruction is explained in Section 3.5.

`r15` is also special. It is linked to the program counter. When `r15` is read, the value of the current program counter is read. And when `r15` is written to, the program counter is changed to that value. That is, a jump occurs in the program flow. The ARM Reference Manual actually specifies that the value of the program counter when read is not the address of the instruction that is executed, but rather that value plus 8 [1]. According to the ARM7 data sheet, this is due to instruction prefetching [2]. In other words, the program counter will already have advanced two instructions ahead.

There is another exception to reading from `r15`, according to the ARMv4 Reference Manual [1]. If the instruction is a store type, such as `str` or `stm`, the value may be 8 more than the address of the instruction, or 12 more. Which one it is depends on the implementation of the ARM ISA. For example, the ARM7TDMI-S data sheet claims the value will be 12 more than the address of the instruction [2]. Due to the ambiguity, the ARM Reference Manual advises against storing the program counter in memory using the `str` or `stm` instructions directly.

3.3. Register Instructions

31–28	27–26	25	24–21	20	19–16	15–12	11–0	
cond	0 0	I	opcode	S	Rn	Rd	operand 2	Data Processing

There are 16 data processing instructions in ARM. Which instruction is executed depends on the value of the `opcode` field. The 16 instructions are listed in Table 3.2. If the `S` bit is set, data processing instructions will modify certain bits in the status register. Which bits can be changed depends on the instruction, the ‘Flags’ field in Table 3.2 shows which fields are changed by an instruction. In general, instructions that use the adder or subtractor can set all flags, and instructions which do not will only set the negative and zero flags.

Table 3.2: ARM Data Processing Instructions [2]

Opcode	Mnemonic	Operation	Flags Set*
0000	and	Rd = Rn & op2	NZ
0001	eor	Rd = Rn ^ op2	NZ
0010	sub	Rd = Rn - op2	NZ
0011	rsb	Rd = op2 - Rn	NZCV
0100	add	Rd = Rn + op2	NZCV
0101	adc	Rd = Rn + op2 + C	NZCV
0110	sbc	Rd = Rn - op2 + C - 1	NZCV
0111	rsc	Rd = op2 - Rn + C - 1	NZCV
1000	tst	set condition codes on Rn & op2	NZ
1001	teq	set condition codes on Rn ^ op2	NZ
1010	cmp	set condition codes on Rn - op2	NZCV
1011	cmn	set condition codes on Rn + op2	NZCV
1100	orr	Rd = Rn | op2	NZ
1101	mov	Rd = op2	NZ
1110	bic	Rd = Rn ^ ~(op2)	NZ
1111	mvn	Rd = ~op2	NZ

*The flags are explained in Section 3.5.1

3.3.1. Operand 2

The lower 12 bits of a data processing instruction specify the second operand, also known as operand2. The encoding of these 12 bits is shown in Figure 3.1. This operand can take one of three forms:

1. An 8 bit immediate rotated to the right by an even amount of positions;

2. The value of a register (Rm) shifted by any amount;

3. The value of a register (Rm) shifted by an amount specified by a third register (Rs).

If the I bit is set, the immediate version is used. The 8 bit immediate is not sign extended. The immediate can be rotated to the right by an even amount of positions, but not an odd amount. Rotating bits to the right means that bits shifted below bit zero will appear starting from bit 31, as opposed to shifting. If the bits were shifted to the right, bits shifted to a position below bit zero will vanish.

If the I bit is not set, another register (Rm) will be used as the second operand. This operand can be shifted in four ways, as shown in Figure 3.1b. An arithmetic right shift sign extends the shifted value based on the 31st bit. The 'rotate right' operation is the same operation as the one used for immediates when the I bit is set. There can be two sources of the shifted amount. If bit 4 of the instruction is 0, an immediate value is used. If bit 4 is 1, the value of a register Rs is used. Thus, operations such as Rd = Rn + (Rm << Rs) can be encoded with just one 32-bit instruction. That is, up to three registers may be used in a single instruction.

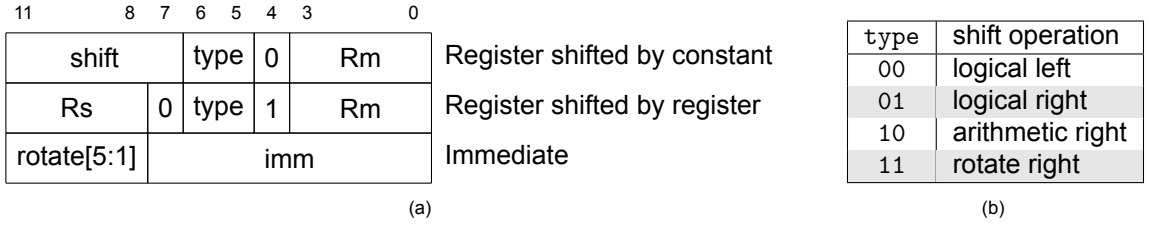
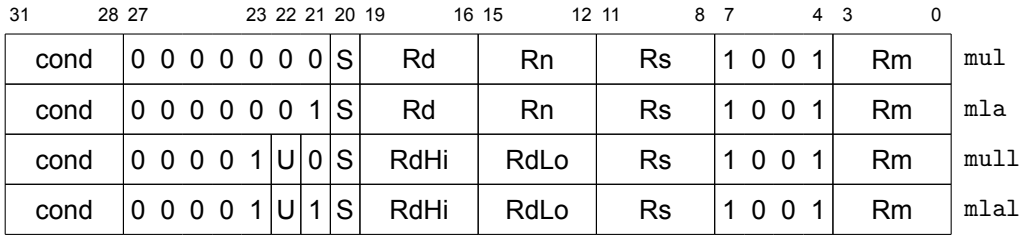


Figure 3.1: a) ARM Shift operations for operand 2, b) Shifts performed depending on type field in Figure a.

3.3.2. Multiply

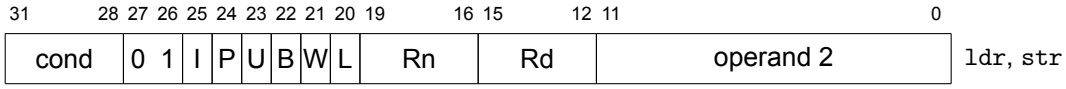


ARMv4 has four kinds of multiply instructions. These are multiply (`mul`), multiply-accumulate (`mla`), multiply long (`mull`), and multiply-accumulate long (`mlal`). `mul` and `mla` will be discussed first, followed by `mull` and `mlal`.

`mul` simply multiplies the 32-bit value of `Rm` with `Rs` and stores the result in the 32-bit register `Rd`. The upper 32 bits of the multiplication are discarded. If the `S` bit is set, the N and Z flags are set correctly, the C flag is set to what the ARM7 data sheet calls a meaningless value [2], and the V flag is unaffected. `mla` performs the same multiplication, but also adds `Rn` to the result. That is, `mla` performs the operation `Rd = Rs * Rd + Rn`.

`mull` performs a full 64-bit multiplication of `Rm` and `Rs`. The lower 32 bits are stored in `RdLo`, and the high 32 bits are stored in `RdHi`. If the `U` bit is set, an unsigned multiplication is performed. If the `S` bit is set, the N and Z flags are set correctly, but the C and V flags will be set to meaningless values. Otherwise, the values of both `Rm` and `Rs` are treated as signed numbers. `mlal` performs the same multiplication, but also adds the 64-bit value of `RdHi` and `RdLo` to the result. In other words, the operation `[RdHi, RdLo] = Rm * Rs + [RdHi, RdLo]` is performed.

3.4. Load/Store Instructions



Arm has various addressing modes for memory access. The basic form is a load/store with a base address stored in a register `Rn` with an offset specified by operand 2. Operand 2 works as described in Section 3.3.1, except that shifting by a register is not possible. Operand 2 is then added to the value of `Rn` to calculate the memory address.

This addressing mode can be changed by 4 bits in the instruction: P, U, B, and W. Table 3.3 shows the effect of the various bits.

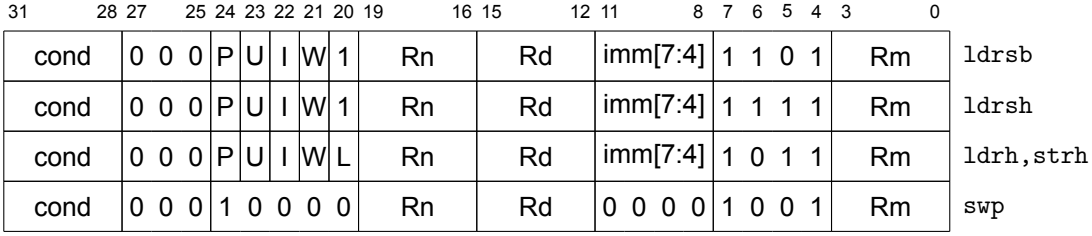
- If the P bit is set, operand 2 is added to `Rn` before indexing into the memory. If the P bit is clear, the memory is read at the address pointed to by `Rn` and `Rn` will have operand 2 added to it;
- If the U bit is set, operand 2 is subtracted from `Rn` instead of being added to it;
- If the B bit is set, only a byte is transfered. If this is a load, the value is not sign extended;
- If the W bit is set, the calculated memory address is written back to register `Rn`.

If the L bit is set, the value is loaded into `Rd` from the memory. Otherwise, the value into `Rd` is stored into memory. Note that if a load is specified and the memory address is written to `Rn`, two registers can be written to in a single instruction. This will be important when discussing the microarchitecture of Combi, the novel processor of this thesis.

Table 3.3: `ldr` and `str` bits and their meaning.

Bit	Mnemonic	Operation if clear	Operation if set
P	Pre-increment	`mem[Rn] = Rd; Rn = Rn + op2`	`mem[Rn + op2] = Rd`
U	Up	`mem[Rn - op2] = Rd`	`mem[Rn + op2] = Rd`
B	Byte	`mem[Rn + op2][31:0] = Rd[31:0]`	`mem[Rn + op2][7:0] = Rd[7:0]`
W	Write back	`mem[Rn + op2] = Rd`	`mem[Rn + op2] = Rd; Rn = Rn + op2`
L	Load	`mem[Rn + op2] = Rd`	`Rd = mem[Rn + op2]`

3.4.1. Special Load/Store Instructions

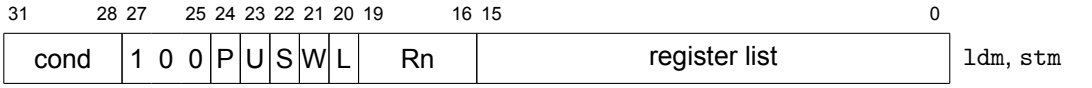


Some new load and store instruction were added in ARMv4, according to the ARM Architecture Reference [1]. These are:

- `ldrsb`: load a sign extended byte;
- `ldrsh`: load a sign extended halfword (16-bits);
- `ldrh,strh`: load a zero extended halfword or store a halfword;
- `SWP`: load the contents of the memory address into `Rd` and stores the value of `Rm` into the same address atomically.

The P, U, W, and L bits work as described in Table 3.3 and Section 3.4. The `I` bit, however, is unique to these instructions. If the `I` bit is set, the bits of the instruction at 11 down to 8 and 3 down to 0 are used to create an 8-bit immediate offset from the base register `Rn`. Otherwise, the value of register `Rm` is added to or subtracted from `Rn`. The immediate is always unsigned, but negative offsets are still possible using the U bit, since the U bit effectively controls the sign of the offset.

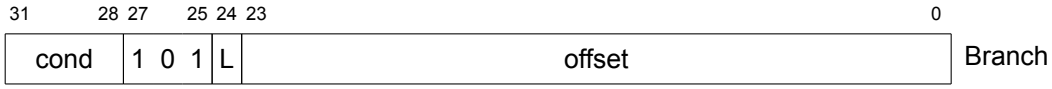
3.4.2. Load/Store Multiple



ARM has instructions for loading and storing the values of multiple registers at once. These are the load multiple (`ldm`) and store multiple (`stm`) instructions. These instructions use the bottom 16 bits as a bit field to specify registers to be stored/loaded. For example, if bits 3, 4, and 13 of the instruction are set, registers `r3`, `r4`, and `r13` will be loaded. Registers are stored and loaded from lowest to highest, so `r3` will bit loaded before `r4`. `r15` - the program counter - can also be specified. However, when using `stm` with `r15`, some architectures will add 8 to the program counter when stored, and some will add 12. Therefore, the ARM Reference manual discourages using `stm` with the program counter (`r15`) [1].

These instructions always increment (or decrement) the address pointed to by `Rn` by 4 for every register loaded or stored. No other immediate or register offset can be specified. Essentially, the registers will be loaded or stored into a contiguous block of memory. The P, U, W, and L bits of the instruction work as specified in Table 3.3 and Section 3.4. In this case, if the P bit is set, the first register will be loaded to the value of `Rn` plus 4. The S bit of this instruction is related to switching privilege levels, and is outside the scope of this thesis.

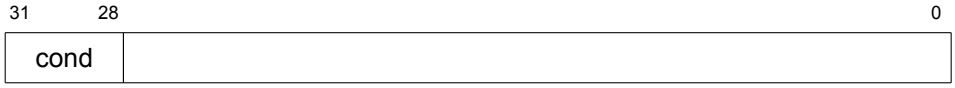
3.5. Branch Instructions



Branches in arm use the conditional execution system explained in Section 3.5.1 to either conditionally or unconditionally branch. The 24-bit immediate is a signed offset which is shifted by 2, since ARM instructions are always aligned to 4 bytes. This 26-bit value is added to the program counter, if the instruction executes. This allows both forward and backward branches. If the desired target is further away than 2^{25} bytes, the target could also be jumped to by using `r15` as the destination of a data processing or memory load instruction.

If the L bit is set, the value of the program counter plus 4 is stored in register `r14`. This is known as the branch and link (`bl`) instruction. `r14` is also known as the link register, and it is not possible to use another register for this function. This is used for calling functions, which will return to the address pointed to by `r14`.

3.5.1. Conditions



Every instruction in ARMv4 can be executed conditionally. This includes branches, of course, but also instructions such as `add` (add two registers) and `str` (store a register in memory). The first four bits of any ARM instruction specify the condition in which it executes. Whether this condition is true depends on the value of the flags register, which can be changed as a side effect of various data processing instructions, see Section 3.3.

According to Section 2.5.1 of the ARM Architecture Reference Manual here are four flags [1]:

1. N: the negative flag. Set when the 31st bit - the sign bit - of the result is `1`.
2. Z: the zero flag. Set when the result is zero.
3. C: the carry flag. Set when an unsigned addition overflows or an unsigned subtraction underflows.
4. V: the overflow flag. Set when a signed addition overflows or a signed subtraction underflows.

Table 3.4: ARM condition codes and their function

Code	Meaning	Flags
0000	equal	Z set
0001	not equal	Z clear
0010	unsigned greater than or equal	C set
0011	unsigned less than	C clear
0100	negative	N set
0101	positive or zero	N clear
0110	overflow	V set
0111	no overflow	V clear
1000	unsigned greater than	C set and Z clear
1001	unsigned less than or equal	C clear or Z set
1010	signed greater or equal	N equals V
1011	signed less than	N does not equal V
1100	signed greater than	Z clear and (N equals V)
1101	signed less than or equal	Z set or (N does not equal V)
1110	always	(ignored)
1111	never*	(ignored)

*The never condition was deprecated in ARMv4. Later versions of the ARM ISA use this bit pattern for new instructions.

3.6. Application Binary Interface

As shown in Table 3.5, the 16 ARM registers have a special meaning in the ARM ABI. ARM has four registers for function arguments and return values, all arguments that do not fit will be pushed to the stack. The stack pointer (`r13`) will point to the first argument, with other arguments in the memory addresses above. Unlike in RISC-V (see Section 2.6), the ARM stack pointer is not necessarily aligned.

When a function is called, this function will use some registers as part of its execution. The value of these registers may need to be restored by either the function called (callee) or the program that called the function (caller). Which one of the caller or callee is responsible for saving the value of a register at a function call is shown in the 'Saved' column of Table 3.5.

Note that two registers do not have a defined saving convention. These are `r12` and `r15`. Since `r15` will always return the value of the program counter, it is not necessary to save it. No data can be stored in `r15`. `r15` could be used to store the return address when returning from a function, but due to the ambiguity of the offset between `r15` and the instruction address, the ARM ABI discourages this [3].

`r12` is not saved for a different reason. This register is designated as the Intra-Procedure Call scratch register. It may be used by the linker to connect library code to binary code. For example, the ARM branch instructions (Section 3.5) can only reach code 32 megabytes (2^{25} bytes) away. To reach subroutines farther away, the linker can insert some code that uses `r12` to jump to a far-away address. For that reason, neither the caller nor the callee can assume the `r12` register is saved.

Table 3.5: ARM registers and usage in the ARM ABI [3]

Register	ABI name	ABI Function	Saved
r0-r3	a1-a4	Arguments	Caller
r4-r11	v1-v8	Variable Registers	Callee
r12	ip	Intra-Procedure Call scratch register	
r13	sp	Stack Pointer	Callee
r14	lr	Link Register	Caller
r15	pc	Program Counter	

4

Combining Instruction Set Architectures

The previous two chapters described the RISC-V and ARM ISAs separately. In this chapter, the two are contrasted to create a path toward the microarchitecture that can implement both ISAs. Sections 4.1 and 4.2 explain the concept of Combi and how registers are shared between the ISAs. Sections 4.3, 4.4, and 4.5 contrast the instructions of ARM and RISC-V across the familiar categories of register, load/store, and branch instructions. Sections 4.6 and 4.7 explain the methodology of library calls across ISA boundaries and some methods for disambiguating the ISA of a binary at a hardware level.

4.1. Multi ISA Support

The Combi microprocessor facilitates both the RV32IZmmul and ARMv4 ISA. Therefore, the opcodes are binary compatible with both RISC-V and ARM. However, some RISC-V instructions have the same binary sequence as ARM instructions. For example, the binary word `e0800113` encodes the ARM instruction `add r0, r0, r3 lsl r1`, but also encodes the RISC-V instruction `addi x2, x0, -504`. There is no way to tell if this word was meant to be interpreted as an ARM instruction or as a RISC-V instruction. To make Combi interpret the instruction stream correctly, a mechanism must exist to disambiguate RISC-V and ARM instructions. This is part of the Combi ABI. Since Combi facilitates both ISAs natively, switching is instant. That is, instructions could switch between ARM and RISC-V at no overhead, not even a pipeline stall.

4.2. Registers

Combi has all the architectural registers of RISC-V and ARM. To facilitate RISC-V programs calling ARM procedures and vice versa, ARM registers could be mapped to RISC-V in a way that facilitates overlap in their respective ABIs. This hypothetical mapping is shown in Table 4.1. This mapping facilitates procedure calls between architectures, at the cost of a slightly more complicated microarchitecture. This more complicated microarchitecture may require more area, run slower, or use more power.

Note that the ARM register `r15` is not available in RISC-V. This is because `r15` is used in ARM to access the program counter. RISC-V does not use a register for this, and instead has special instructions that specifically modify the program counter. In a similar vein, the RISC-V register `x0` has no analogue in ARM. In RISC-V, `x0` is used as a constant 0. ARM has no such register.

4.3. Register Instructions

ARM has 20 data processing instructions - also counting the shifts that can be applied to operand 2 - but RISC-V has only 10. Some of these instructions perform the same operation. As can be seen in Table 4.2, ten ARM instructions have a RISC-V equivalent. Two of these use a pseudo-instruction, an encoding that acts like one instruction but uses a different encoding. First, the ARM `mov` instruction can be performed in RISC-V by adding the source with `x0`. Since `x0` is always zero, this has the same architectural behavior as the ARM `mov` instruction. Second, the ARM `mvn` instruction can be performed in RISC-V by using `xori` with an immediate value of -1. Since the XOR operation with a '`1`' bit is

equivalent to negation, this RISC-V instruction inverts the bits of the source. This is equivalent to the ARM `mvn` instruction.

There are 10 ARM instructions which have no analogue in RISC-V. Of these, eight require the status registers, which RISC-V does not have. These are `adc`, `sbc`, `rsc`, `tst`, `teq`, `cmp`, and `cmn`. The last two, `bic` and `rsb`, perform operations which RISC-V simply has no equivalent to. Note that `rsb` (reverse subtract) is more useful in ARM than RISC-V. In ARM, only the second operand can be shifted in a register-register operation. It may therefore not be trivial to swap the operands of the subtract instruction. This is why `rsb` is useful. In RISC-V, no operand is shifted in this way. The operands of `sub` can simply be swapped. Therefore, `rsb` would not add any value to the RISC-V ISA.

There are also two RISC-V instruction which have no ARM equivalent. These are `slt` (Set if Less Than) and `sltu` (Set if Less Than Unsigned). Two possible applications of these instructions are bounds checking for array accesses and performing big integer arithmetic. In both of these cases a program written in ARM would use the carry flag of the status register.

4.3.1. Multiplication

ARM multiplication is encoded differently from other data processing instructions, but RISC-V uses the same encoding for both. Combi will support both encodings. In ARM, a single instruction can perform a 32x32 -> 64 bit multiplication, but in RISC-V this would take two instructions, one for the high 32 bits and one for the low 32 bits. That also means the ARM instruction will write to two registers in one instruction, whereas most instructions only write to one register. This will have consequences for the microarchitecture of Combi.

4.4. Load/Store Instructions

Despite the unique encoding of ARM loads and stores, the behavior is similar to RISC-V. As shown in Table 4.3, all single-byte ARM loads and stores have a RISC-V equivalent. The addressing mode is a base plus an offset, and bytes and halfwords can be read with sign extension or zero extension. However, like multiplication, the write back feature of ARM loads makes a single instruction write to multiple registers. The `ldm` and `stm` instructions can even modify up to 16 registers (`ldm`), or 16 words in memory (`stm`). Once again, this will have consequences for the microarchitecture of Combi.

4.5. Branch Instructions

Superficially almost every ARM branch has an equivalent in RISC-V. This is shown in Table 4.4. The only exception is the `bvs` and `bvc` pair of instructions, which branch if the previous instruction caused a signed overflow. RISC-V does not have such an instruction.

However, branches in ARM work quite differently from RISC-V. In ARM, there is only one branch instruction with a 24-bit offset. Conditional branches are performed using the `cond` bits that every ARM instruction has. Whether the branch is taken or not depends on the operation of previous instructions, for example `cmp` or `tst`.

In RISC-V, conditional branches are completely different from unconditional ones. The conditional branch has only a 12 bit offset, and it specifies the condition for branching itself. That is, the branch condition is calculated by the branch instruction, not an instruction preceding it.

Unconditional branches in RISC-V are called jumps, and there are two types. A jump may use a 20-bit immediate offset, or a jump may be to a register (with a 12 bit offset). The jump to a register

Table 4.1: ARM registers mapped to RISC-V registers

ARM Register	RISC-V Register	ABI Function
r0-r3	x10-x13 (a0-a3)	Arguments
r4-r5	x8-x9 (s0-s1)	Variable Registers
r6-r11	x18-x23 (s2-s7)	Variable Registers
r12	x5 (t0)	Intra-Procedure Call scratch register
r13	x2 (sp)	Stack Pointer
r14	x1 (ra)	Link Register
r15		Program Counter

Table 4.2: Combined set of ARM and RISC-V register-register instructions

ARM mnemonic	RISC-V mnemonic
`and`	`and`
`eor`	`xor`
`sub`	`sub`
`add`	`add`
`orr`	`or`
`mov r1, r2`	`add x1, x0, x2`
`mvn r1, r2`	`xori x1, x2, -1`
(operand2) `lsl`	`sll`
(operand2) `lsr`	`srl`
(operand2) `asr`	`sra`
(operand2) `ror`	
`rsb`	
`bic`	
`adc`	
`sbc`	
`rsc`	
`tst`	
`teq`	
`cmp`	
`cmn`	
	`slt`
	`sltu`

Table 4.3: Combined set of ARM and RISC-V load/store instructions

ARM mnemonic	RISC-V mnemonic
`ldr`	`lw`
`ldrsh`	`lh`
`ldrsb`	`lb`
`ldrh`	`lhu`
`ldrb`	`lbu`
`str`	`sw`
`strh`	`sh`
`strb`	`sb`
`ldm`	
`stm`	

operation is equivalent to an ARM instruction that moves something to register `r15`.

Branch and link are performed the same way in RISC-V and ARM. The linked address is the address of the branch instruction plus four. That is, the address of the instruction after the branch instruction is stored in the link register.

4.6. Application Binary Interface

The Combi ABI should facilitate not only procedure calls to the same ISA, but also from one ISA to another. Preferably, it would appear to both the caller and callee that the other is using the same ISA. That is, neither binary has to be modified. To achieve this, two things must be corrected in between cross-ISA procedure calls: the registers, and the stack layout.

In Figure 4.1 and 4.2, a general methodology for calling and returning to a foreign ISA is shown. The method to use depends on properties of both the ISAs and the microarchitecture itself. This methodology can be used by a dynamic loader or a static linker to facilitate, for example, a RISC-V application using an ARM library. Neither the application nor the library are affected, but the linker may need to do some more work.

Table 4.4: Combined set of ARM and RISC-V branch instructions

ARM mnemonic	RISC-V mnemonic	Condition
`b`	`j`	
`bl`	`jal`	
`bx`	`jalr`	
`beq`	`beq`	equal
`bne`	`bne`	not equal
`bcs`	`bgeu`	unsigned greater than or equal
`bcc`	`bltu`	unsigned less than or equal
`bhi`	`bltu`*	unsigned greater than
`bls`	`bgeu`*	unsigned less than
`bmi`	`bltz` (`blt rs, x0`)	signed less than zero
`bpl`	`bgez` (`bge rs, x0`)	signed greater than or equal to zero
`bge`	`bge`	signed greater than or equal
`blt`	`blt`	signed less than
`bgt`	`blt`*	signed greater than
`ble`	`bge`*	signed less than
`bvs`		overflow
`bvc`		no overflow

* Swap the arguments of this instruction, i.e. `bhi r1, r2` is equivalent to `bltu r2, r1`

The first difference in possible methods depends on whether the microarchitecture or the linker handles mapping ABI registers. This is represented by the first choice in Figures 4.1 and 4.2. Using the register mapping from Table 4.1, all ARM registers would correspond to the correct RISC-V register in a procedure call. If the microarchitecture does not handle this, the linker will need to insert code to swap the registers in software for ABI compatibility. If the microarchitecture does remap the registers from one ISA to another, the linker does not need to do any swapping in software.

However, RISC-V has more registers than ARM. RISC-V uses 8 argument registers in its ABI, and ARM has only 4. The second and third choices in Figure 4.1 show what method to use based on which ISA is calling and which is called. The 4 registers that are present in RISC-V but absent in ARM must be pushed to the stack before an ARM procedure is called from RISC-V. This would not be necessary when calling a RISC-V function from ARM. However, after switching ISAs, some arguments that were pushed on the stack may need to be popped into RISC-V registers.

On the other hand, a RISC-V procedure only uses 2 registers for the return value, and ARM uses 4. These two registers would have to be pushed to the stack when returning from ARM. The opposite applies to calling a RISC-V routine from ARM. The second and third choices in Figure 4.2 show what method to use based on which ISA is returning. Registers in RISC-V that are absent in ARM have to be filled with data popped from the stack when an ARM function calls a RISC-V function. These additional instructions can be inserted by a linker when a library with one ISA is being used by a program with the other ISA.

Figure 4.3 shows an example of code being inserted. On the left, the RISC-V code calls a routine `fun`, which has five arguments. The RISC-V code is agnostic to the fact that this is an ARM routine. The linker has inserted some glue code that will make the RISC-V function call appear like an ARM function call for the ARM callee routine. The RISC-V code has provided the fifth argument in register `a4`, but the ARM routine expects to pop it from the stack. Therefore, the glue code allocates a bit more stack space and stores the value of `a4` there. The ARM routine loads this value from the stack and proceeds as normal, without having to know that it was called from RISC-V code.

4.7. Disambiguating ISAs

There is also the issue of switching the Combi processor from interpreting RISC-V instruction to interpreting ARM instructions. There are three methods to do this:

- Use a dedicated opcode to switch;
- Detect the ISA based on statistical properties;

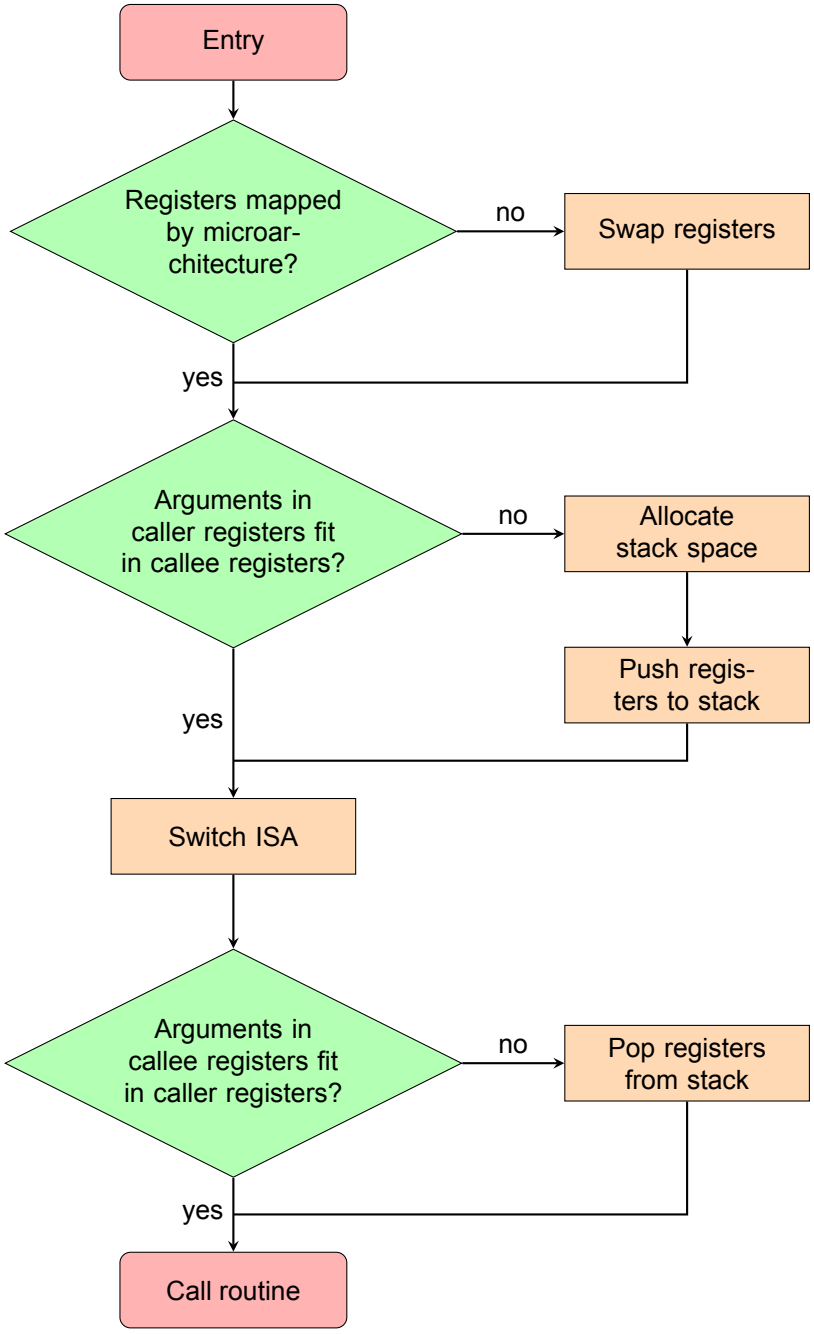


Figure 4.1: Possible methods for calling a different ISA

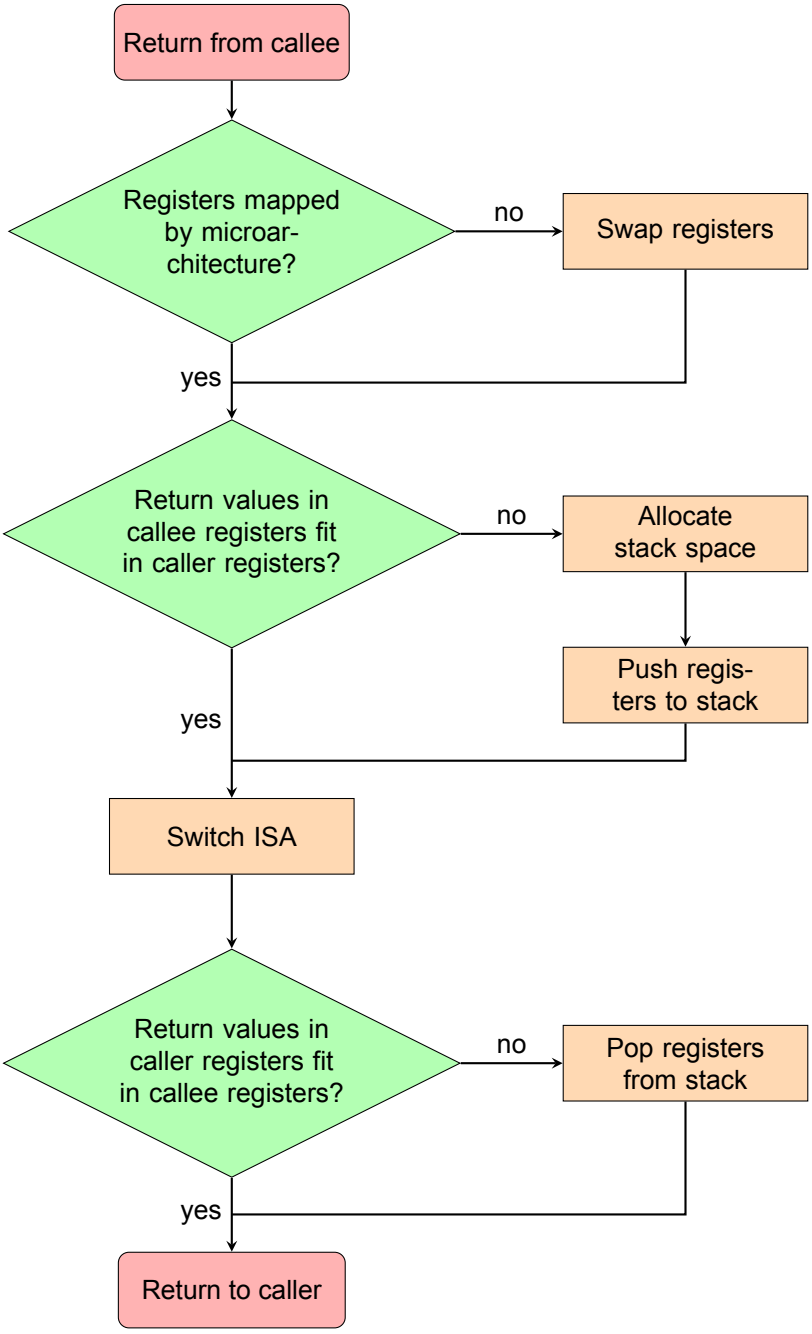


Figure 4.2: Possible methods for returning from calling a different ISA

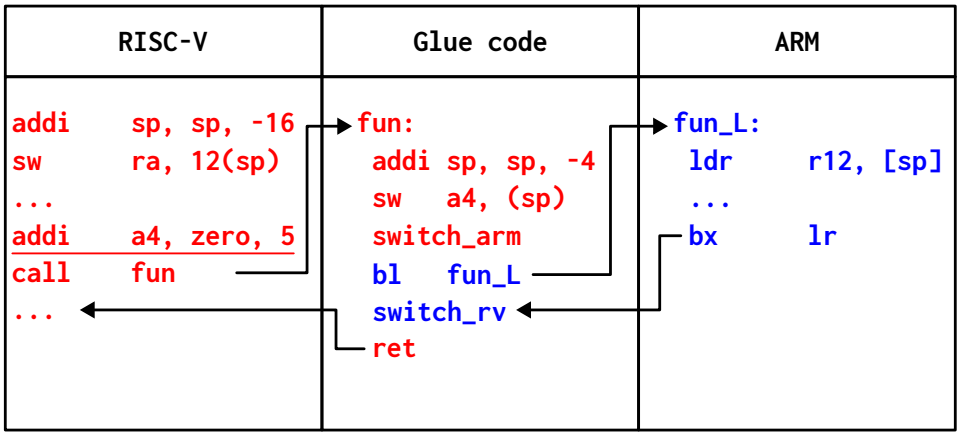


Figure 4.3: Example of a RISC-V binary calling an ARM routine with 5 arguments. RISC-V code is red, ARM code is blue.

- Assign an ISA to individual pages in the memory map.

These options will be explored in more detail.

Dedicated Opcode
Using a dedicated opcode to switch would involve having some 32-bit value that has no meaning in ARM to encode the 'switch to RISC-V' instruction, and having a 32-bit value for RISC-V likewise. All ARMv4 instructions that have the binary sequence '011' at bit positions 27 to 25 are invalid, so an instruction like that could be used. RISC-V has certain opcodes marked as 'custom', which guarantees that they will not be used for standard extensions. The RISC-V ISA manual expressly recommends these for custom instructions.

Use Statistical Properties
Most ARM instructions have the bit sequence '1110' in bits 31 to 28, but few RISC-V instructions do. This is because bits 31 to 28 of an ARM instruction are the condition bits, and '1110' mean to execute the instruction always. In RISC-V these bits have no special meaning. It is therefore possible to tell at a glance if a binary instruction sequence is ARM, and it would be possible to make a processor that can detect this pattern to determine the mode it should operate in. However, this method is error prone, and decoding even a single instruction wrong could lead to the entire program state being corrupted.

Use Virtual Memory
Finally, the paging system could be used to tell if a piece of code is ARM or RISC-V. If the program is loaded in such a way that RISC-V and ARM code never occur on the same page, a bit could be used to signal which executable code is RISC-V and which is ARM. This is likely to be the case if the ARM code is dynamically loaded, but not if the ARM code is statically linked.

Conclusion
Considering that the linker has to insert some instructions between most function calls anyway, the best option for this work is to use a dedicated opcode. The overhead may be small compared to other code the linker inserts. As shown in Figure 4.3, the linker already has to insert two instructions if the function called has more than four words of arguments. This system works in both static and dynamically linked contexts, and is not error-prone.

If an operating system is present, and applications are loaded into virtual memory, using paging to distinguish ARM and RISC-V code would avoid needing to add an explicit 'swap ISA' instruction. As shown in Figures 4.1 and 4.2, this may even allow a foreign ISA call to occur without the linker needing to insert any code, although this would be limited to functions that take only a few arguments, and return only a few values. In the case of RISC-V and ARM, any function that takes four word sized arguments and returns a result of two word sizes would qualify. For example, the standard C library function `memset` has a signature of `void *memset(void s, int c, size_t n)`. If the right method is used, one could call this function from an ARM library inside a RISC-V program without needing to modify or insert any code.

5

Combi Implementation & Results

In this chapter, the implementation of Combi - the microprocessor which facilitates two ISAs - is presented. Section 5.1 describes how the platform to base Combi on was selected. Section 5.2 delves into the details of how Combi was created from this platform. Section 5.3 discusses the results of this implementation using facts and figures. Finally, Section 5.4 puts the results of this work into context with the state of the art in this field.

5.1. Platform Choice

In order to start making the Combi microarchitecture, it would make sense to start with a platform that already implements RISC-V and could implement ARM, or vice versa. A structured platform is also a prerequisite, since the aim of Combi is to reuse as many parts to facilitate the RISC-V and ARM ISA.

The VexRiscV is a RISC-V microprocessor that is designed to be extendable, which implies a sense of modularity. In fact, every RISC-V instruction is implemented separately, and if none of the instructions are specified as part of the CPU, only an empty 5-stage pipeline is generated [33]. VexRiscV was made in a software oriented way, and is written in a language called SpinalHDL [34]. VexRiscV is certainly an option, however a design using a more common Hardware Description Language (HDL) such as SystemVerilog would be preferable.

Another option was the microarchitecture described in the book Digital Design and Computer Architecture, by Sarah and Davis Harris [39]. This book is an educational piece, and therefore describes the microarchitecture in extensive detail. There is also an ARM edition [38]. This is a promising start to a microarchitecture that combines the RISC-V and ARM ISA. Do note that the microarchitecture described in Digital Design and Computer Architecture only implements a few instructions. The RISC-V edition implements `add`, `sub`, `and`, `or`, `sw`, `lw`, and `beq`. The ARM edition implements `add`, `sub`, `and`, `orr`, `str`, `ldr`, and `b`. This is far from a complete set, and all other instructions will have to be implemented in Combi.

5.2. Implementation Details

In this section, we give a general overview of the Combi microarchitecture, starting from the design of Digital Design and Computer Architecture. Next, we show the additional components needed to facilitate all RISC-V and ARMv4 instructions.

5.2.1. Overview

In this section, the framework that Combi inherits from the Digital Design and Computer Architecture book is described [39]. Combi has a 5 stage pipeline, as shown in Figure 5.1:

- Fetch: a 4-byte instruction is fetched from memory. The program counter is also incremented. If a branch occurs, the program counter is overwritten by data from the execute stage;
- Decode: the instruction is decoded into control signals for the following stages. Multi cycle instructions are dispatched by sending different control signals each cycle while the fetch stage is

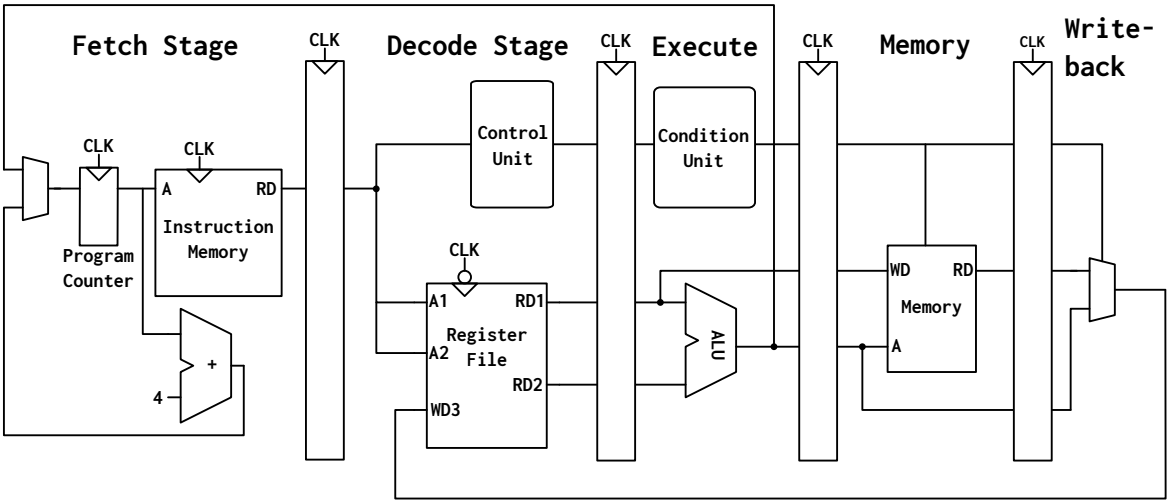


Figure 5.1: Combi 5 stage pipeline with most significant modules

stalled. The register file is also in this stage;

- Execute: The ALU performs an operation on the data from the register file. The conditional logic for branches and conditional ARM instructions is also handled here. If a branch occurs the new address is immediately forwarded to the fetch stage;
- Memory: The result of the ALU can be used as an address into memory. The memory value read from this address advances to the write-back stage, or the value of one of the registers can be written to memory;
- Write-back: Either the ALU result or the memory value is written back to a register. If the register in question is the program counter - as is possible in ARM - the value is instead forwarded to the fetch stage.

In addition to the pipeline, a hazard handling unit ensures all instructions are executed correctly in parallel. The hazard unit uses three systems to ensure the proper operation of the pipeline:

- Forwarding: results from one stage are forwarded to another, skipping the usual pipeline stages. This incurs no performance penalty;
- Bubbles: one of the pipeline stages has its control bits cleared, and the stages before it don't update for a cycle. Normal control flow resumes in the next cycle, except a no-operation 'bubble' is traveling down the pipeline from the stage that was cleared. A performance penalty of one machine cycle is incurred.

There are three common pipeline hazards to handle:

- Data dependencies: an instruction uses registers that will be written to by an earlier instruction which is still in flight. The value can be forwarded to the execute stage from a later stage. This is shown in Figure 5.2. In this figure, the result of instruction 1 is used in instruction 2. That is why the data is forwarded from the memory stage to the execute stage at cycle 4. The result of instruction 1 is also used in instruction 3, which is why it is forwarded from the write back stage again in cycle 5;
- Load dependencies: an instruction uses a register that will be written to via the memory unit. The value cannot be forwarded since it will only be available in the write-back stage. A bubble must be inserted in the execute stage. Figure 5.3 shows that forwarding is impossible, because the red arrow travels back in time from cycle 5 to cycle 4. Instead, the execute and decode stages must be stalled for 1 cycle to make forwarding possible. This stalling inserts a bubble;

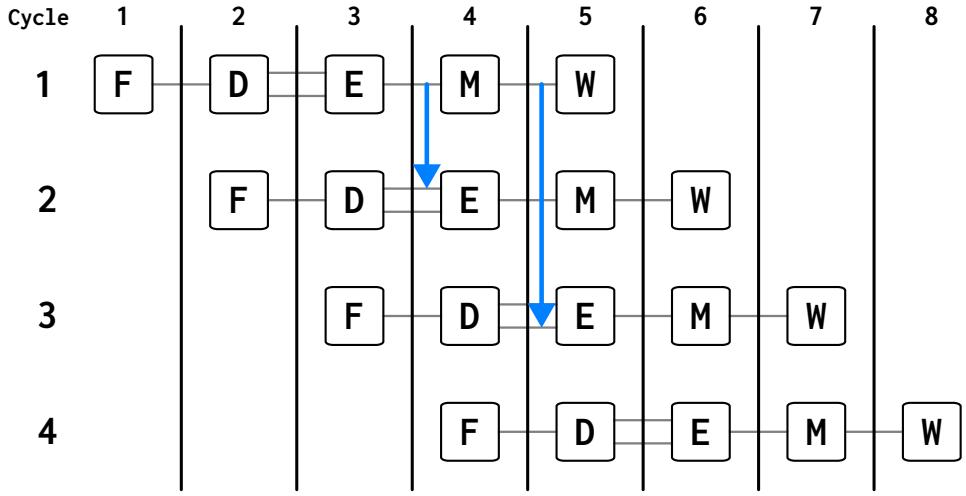


Figure 5.2: Data hazard resolved by forwarding

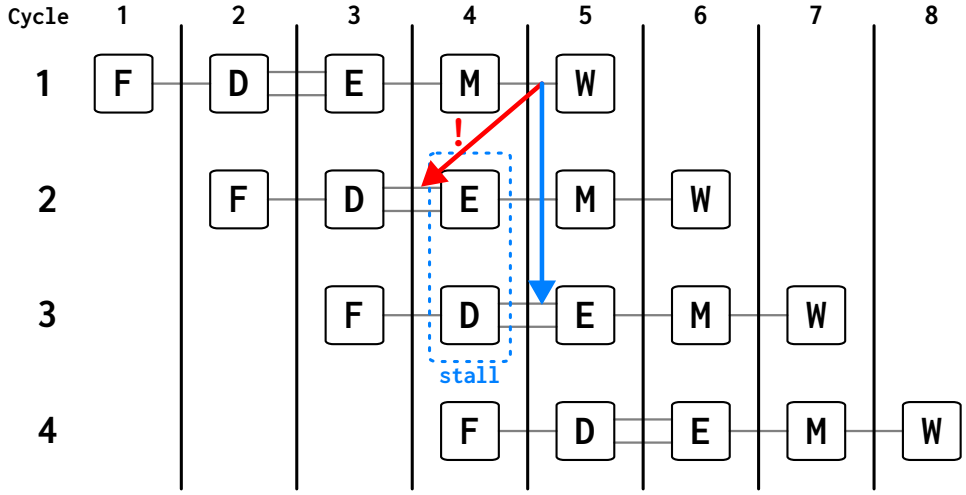


Figure 5.3: Load hazard resolved by stalling for one cycle and forwarding

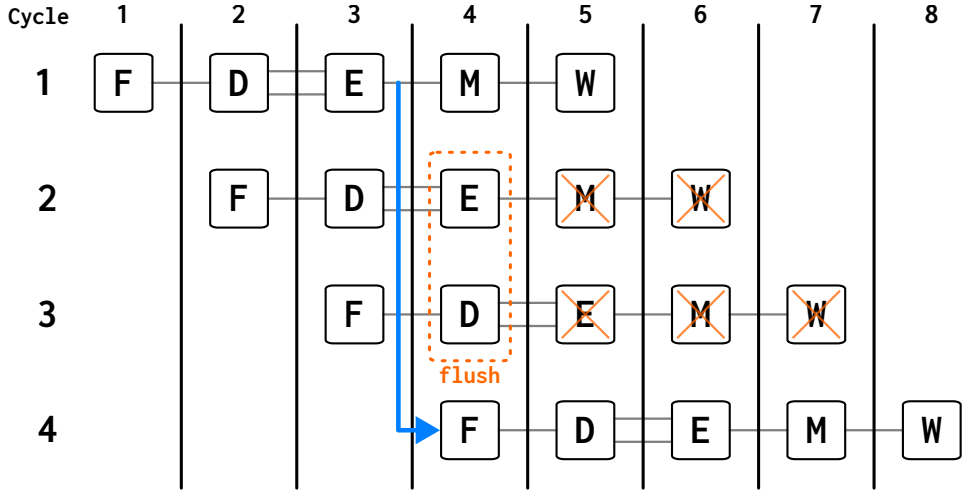


Figure 5.4: Control hazard (branch) resolved by flushing two instructions

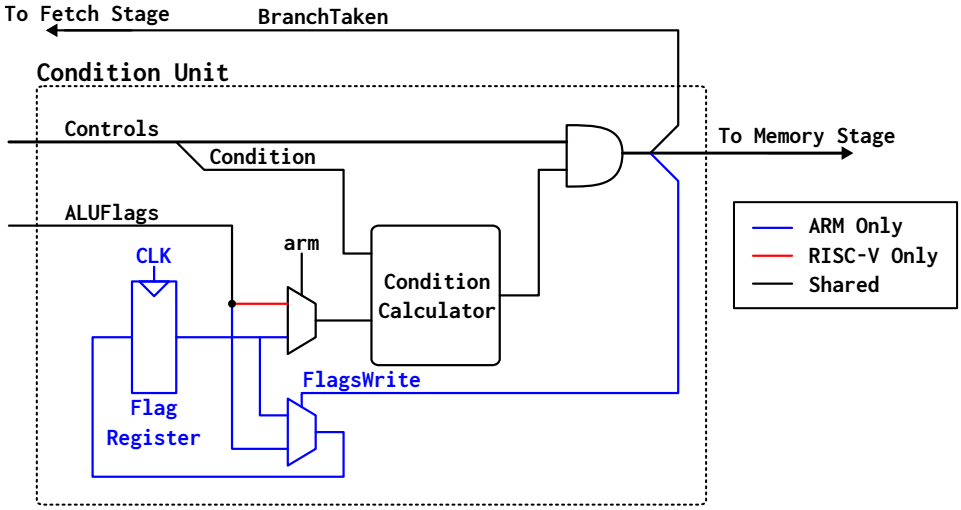


Figure 5.5: Combi condition unit, responsible for conditional execution of ARM and RISCV instructions.

- Branches: when a branch occurs in the execute stage, the fetch and decode stages contain instructions that were fetched from the wrong address. These instructions were fetched assuming the branch was not taken. The instructions in the fetch and decode stage must be flushed. Two bubbles are inserted. As shown in Figure 5.4, the execute and decode stages are flushed such that instructions 2 and 3 never change any architectural state.

An additional bit is propagated from the decode stage to the subsequent stages. If it is a '1', that stage is executing an ARM instruction. Otherwise, the stage is executing a RISC-V instruction. Behavior that should different between the two is switched on or off based on this bit.

5.2.2. Branches

Despite branches in RISC-V and ARM working differently, Combi can implement both in much the same way. In ARM, there is one branch instruction, simply `b`. The conditional execution bits reserved for any instruction are used to make conditional branches (`beq`, `bne`, etc.). Whether a branch is taken depends on the result of the instruction that previously set the condition bits in the status register, for example `cmp, r1, r2`. In RISC-V, there is no status register. Conditional branches specify the registers to be compared directly (`beq, r1, r2`). RISC-V and ARM do share similar branch conditions, though.

These two different mechanisms are unified in Combi's condition unit. The internals of the condition unit are shown in Figure 5.5 This unit takes the flags generated by the ALU and calculates whether the instruction should continue executing. This is done by clearing the control bits if the instruction should not execute, essentially transforming the instruction into a pipeline bubble. The only difference between ARM and RISC-V mode is the source of these flags. In RISC-V mode, these flags come from the ALU directly. In ARM mode, these flags are stored in the pipeline register between the decode and execute stages. The condition unit will therefore use the flags from a previous instruction.

Target address calculation

In ARM mode, the ALU is used to calculate the target address. In RISC-V mode, the ALU is already occupied with generating the flags for conditional branches, so the branch address is generated by a separate adder. The placement of this adder is shown in Figure 5.6.

There are three sources of a branch in Combi. One is unique to RISC-V, one is unique to ARM, and one is shared between both. The result of the separate branch adder is unique to RISC-V. It is used for branches and jumps. The result at the write-back stage is used in ARM. It is used when `r15` is the target of an instruction. The result of the ALU in the execute stage is used by both. In ARM, this is used in the common branch instruction. In RISC-V, it is only used by the jump and link register `jalr` instruction. This RISC-V instruction is the only one with a register source and the program counter as a target. It is also an unconditional jump, so the ALU does not need to generate any flags unlike conditional branches in RISC-V. Hence, the ALU can be used to calculate this target address. The two

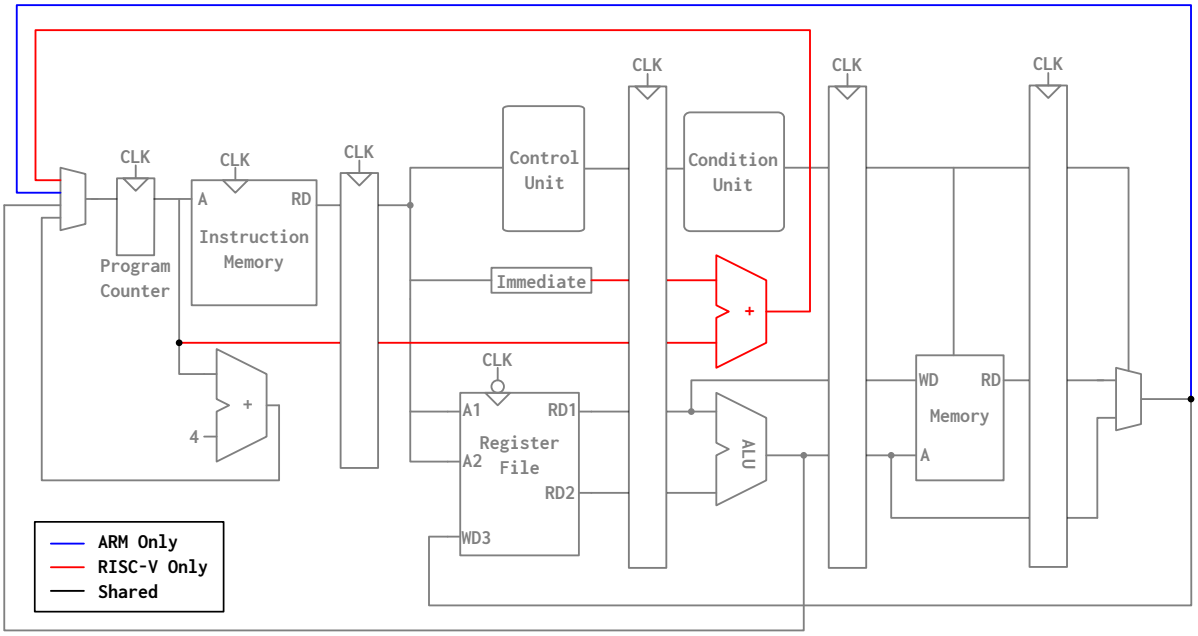


Figure 5.6: Unique branch sources for RISC-V and ARM

unique branch sources are shown in Figure 5.6. The common branch source was already shown in Figure 5.1.

5.2.3. Multi Cycle Instructions

If the register file has one write port, the machine can change the value of one register per cycle. Similarly, if the register file has two read ports, the machine can use the value of two registers per cycle. Some ARM instructions require more than this. These are:

- Data Processing instructions which use a register specified shift amount;
- Loads that write back to the base register;
- Stores that use a register offset to the base register;
- Load/Store multiple;
- All multiply instructions except the basic `mul`.

Some of these instructions could be done in a single cycle, at the cost of area. However, the register file, pipeline registers, and Arithmetic and Logic Unit (ALU) would need a bigger amount of ports. Additionally, more 32-bit busses would have to be routed in the CPU. Alternatively, these instructions could be allowed to take multiple cycles.

The Combi data path shown in Figure 5.1 has two values read into the ALU, and one value written back into the register file. The Load/Store multiple instructions can use up to 16 registers at once, so those should definitely be executed over multiple cycles. If the hardware for multi-cycle instructions is implemented anyway, other instructions which do not fit the Combi data path can be done in multiple cycles as well. No RISC-V instruction needs to execute in multiple cycles, and many ARM instructions can run in one cycle too. The types of instructions that do need multiple cycles to execute are shown in Table 5.1. These need to be run over multiple cycles because they either read from more than 2 registers, or because they write to more than 1 register.

In order to implement multi-cycle instructions, the instruction decoder is augmented. The instruction decoder can now instruct the hazard unit to stall the fetch stage. It also has an internal micro program counter, which counts up as micro instructions are issued. When the final micro instruction has been issued, the fetch stage stall is deasserted and program flow continues. Finally, the instruction decoder can force the ALU result to be forwarded from the memory stage back to the execute stage. This

Table 5.1: ARM instructions that Combi executes over multiple cycles.

Command	Action	Registers read	Registers written
`add rd rm rn LSL rs`[2]	`rd = rm + (rn << rs)`	**3**	1
`ldr rd, [rn,rm]!`[2]	`rd = mem[rn+rm]; rn = rn + rm`	2	**2**
`str rd, [rn,rm]`[2]	`mem[rn+rm] = rd; rn = rn + rm`	**3**	1
`ldm rb ...`	Pops registers from the stack	1	$\leq$ **16**
`stm rb ...`	Pushes registers to the stack	$\leq$ **16**	0
`mla rd rm rs rn`	`rd = (rn * rs) + rm`	**3**	1
`mull rdLo rdHi rs rn`	`[rdHi, rdLo] = rn * rs`	2	**2**
`mlal rdLo rdHi rs rn`	`[rdHi, rdLo] = [rdHi, rdLo] + (rn * rs)`	**4**	**2**

[2]Also includes other register and load store instructions using the shifting, post-, or pre-increment option.

overrides the hazard unit's forwarding logic. This allows subsequent microcode instructions to use previous ALU results without modifying architectural registers. In fact, since this system reuses the pipeline registers, no additional data registers are needed to implement multi-cycle instructions at all.

5.2.4. Register Instructions

Both RISC-V and ARM have instructions that process two registers and write the result to one register. In ARM, these are called Data Processing instructions, and in RISC-V, these are called R-type instructions. We will refer to both of these as register instructions.

Some of the register instructions of ARM and RISC-V overlap in functionality. Table 5.3 shows all the operations the ALU of Combi can perform, and which instructions use the ALU in this mode. There are 23 functions the Combi ALU can perform of which 11 are used for both RISC-V and ARM instructions, 2 are unique to RISC-V, and 8 are unique to ARM. Two of these instructions (`adc` and `sbc`) only require the use of the carry flag, which is absent in RISC-V. The remaining 6 instructions require more involved modification of the ALU.

Control Signals of Register Instructions

Like many instructions, the register instructions of ARM and RISC-V use the same control signals. To simply, not nearly all control signals have been discussed here. This section aims to show the general principle that applies to all control signals: ARM and RISC-V instructions can be decoded to the same signals, making the Combi microprocessor execute either ISA.

Some instructions from both ISAs, and the most relevant control signals are shown in Table 5.2. The control signals are also shown in Figure 5.7, which shows how the control signals affect the various modules of the Combi microarchitecture. Note that all of these control signals are shared between RISC-V and ARM instructions.

The `ResultSrc` signal selects where the value that is written to the register comes from, for these instructions that is always the ALU, but for store instructions this would be memory, and for branch and link instructions (`bl` on ARM and `jalr` on RISC-V) this would be the program counter. The `RegWrite` signal enables writing to the register file in the write back cycle, which is why it takes a long loop in Figure 5.7 to propagate through the other stages first. The `MemWrite` signal similarly enables writing to the memory, which is turned off for register instructions. Instructions that store registers to memory would have these values switched. The `ALUControl` signal selects one of the ALU functions shown in Table 5.3, and this is what differentiates many of the register instructions. For example, the RISC-V `add` and `sub` instructions have the same control signals except for `ALUControl`. Finally, the `Condition` signal is used for conditional execution of ARM instructions. This is used by the Condition Unit, as shown in Figure 5.5. For ARM instructions, this signal is wired to the upper bits of the instruction. RISC-V instructions have no conditional execution predicate, so the `Condition` signal is wired to `always` (`1110`) for RISC-V instructions. Conditional branches in RISC-V do use the condition bits to specify the condition for which the branch is taken.

Register Instructions Using the Program Counter

The ARM `r15` register is always a unique one. When read, Combi will substitute the value read from `r15` with the program counter plus 8 in the decode stage. This can be done by using the pc plus four

Table 5.2: Microcode of some Register Instructions

Type	Instruction	ResultSrc	RegWrite	MemWrite	ALUControl	Condition
RISC-V	`add`	`ALU`	1	0	`add`	`always`
ARM	`add`	`ALU`	1	0	`add`	instr[31:28]
RISC-V	`or`	`ALU`	1	0	`or`	`always`
ARM	`orr`	`ALU`	1	0	`or`	instr[31:28]
RISC-V	`sub`	`ALU`	1	0	`sub`	`always`
ARM	`rsb`	`ALU`	1	0	`sub` reverse	instr[31:28]

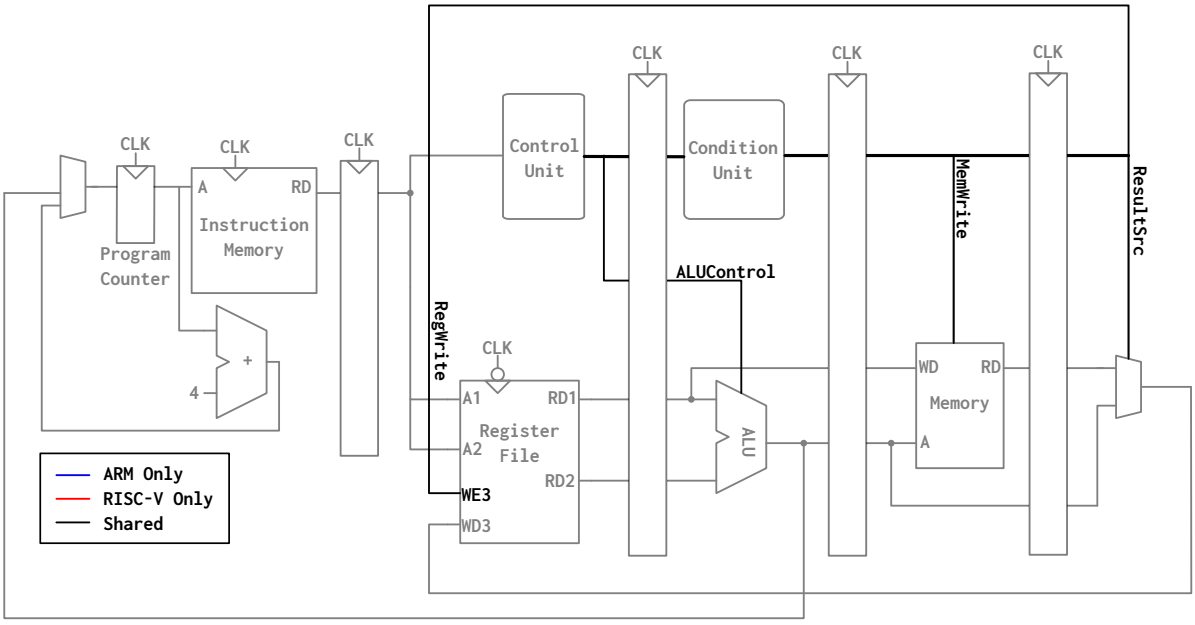


Figure 5.7: Control signals used for register instructions.

Table 5.3: ALU operations and the instructions that use them

Operation	Functional Unit	ARM instruction	RISC-V instruction
op1 & op2	Logic	and, tst	and
op1 ^ op2	Logic	eor, teq	xor
op1 - op2	Adder	sub, cmp	sub
op1 + op2	Adder	add, cmn	add
op1 | op2	Logic	orr	or
~op2	Not	mvn	
op1 << op2	Shifter	(operand2) lsl	sll
op1 >> op2	Shifter	(operand2) lsr	srl
op1 >>arith op2	Shifter	(operand2) asr	sra
op1 >>rot op2	Shifter	(operand2) ror	
op2 - op1	Adder	rsb	
op1 & ~op2	Logic	bic	
op1 + op2 + carry	Adder	adc	
op1 - op2 - ~carry	Adder	sbc	
op2 - op1 - ~carry	Adder	rsc	
op1 - op2 < 0 ? 1 : 0	Adder		slt
op1 < op2 ? 1 : 0	Adder		sltu
op1 *low op2	Multiplier	mul	mul
op1 *high op2	Multiplier	mull	mulhu
signed(op1) *high signed(op2)	Multiplier	smull	mulh
signed(op1) *high op2	Multiplier		mulhsu
op1		(multi cycle instructions)	
op2		mov	lui

signal in the fetch stage, because the program counter has already been incremented by four in the fetch stage. The fetch stage is already fetching the instruction after what's in the decode stage, so it is already incremented by four. The only exception to this rule is if the fetch stage is running a jump. In this case, the instruction in the decode stage will be flushed anyway, since the Combi microarchitecture always speculatively execute that branches are not taken. This trick is also used by Sarah and David in the ARM edition of the Digital Design and Computer Architecture book [38].

When `r15` is used as a destination, the program counter must be updated. To allow this, the hazard unit inserts four bubbles in the decode stage until the instruction that writes to `r15` has exited the write-back stage. This is a significant performance overhead, but the common branch instruction does not use this path. Programs that write to `r15` often instead of using the branch instruction will take longer to execute, but the ARM reference manual discourages using `r15` in this manner except for jumps that are more than 32 megabytes away [1]. This may therefore be a rare occurrence.

When an ARM register instruction uses a register shift amount, i.e. `ADD rd, rn, rm LSL rs` it needs to be executed over two cycles. In the first cycle, `rm` is shifted by `rs`. In the second cycle, the result of the first cycle is forwarded and added to `rn`. Instructions that use a constant shift amount can be executed in one cycle. The alternative would be to have a third read port on the register file, a bigger pipeline register, and more forwarding logic. This would allow instructions that use a register shift amount to also execute in one cycle.

5.2.5. Multiplication

ARM and RISC-V have very different multiplication instructions. In RISC-V, a 32*32->64 multiplication is done in two instructions: `mul` for the low 32 bits and `mulh` for the high 32 bits. ARM can also calculate only the bottom 32 bits as a separate instruction. The best way to combine these is to make the 32*32->64 multiplication spread out over two cycles, the first takes the same data path as the RISC-V `mul` and the second will take the path of RISC-V `mulh`. Thus, a 64-bit result is written in two cycles. This is also maximally efficient usage of the single 32-bit write port on the register file.

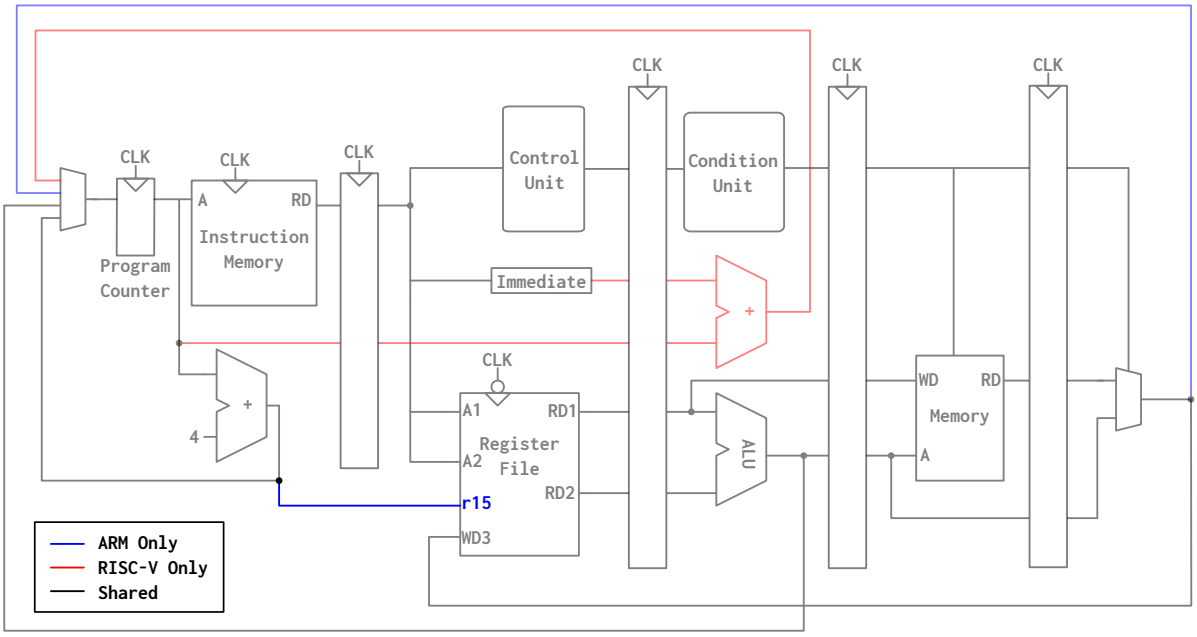


Figure 5.8: Forwarding the program counter for ARM instructions using register 15

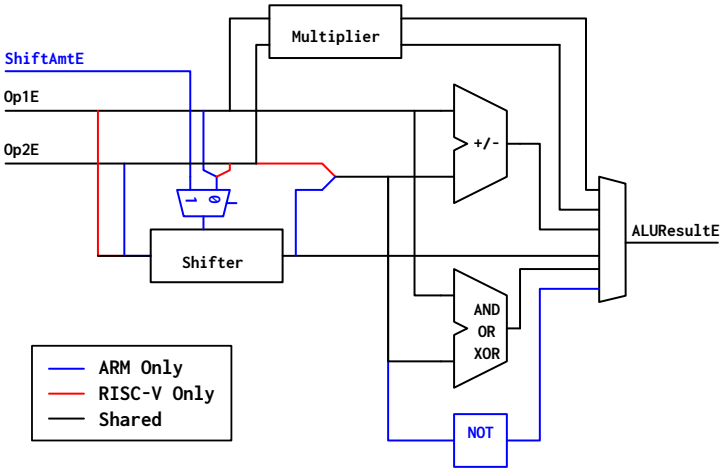


Figure 5.9: Combi ALU

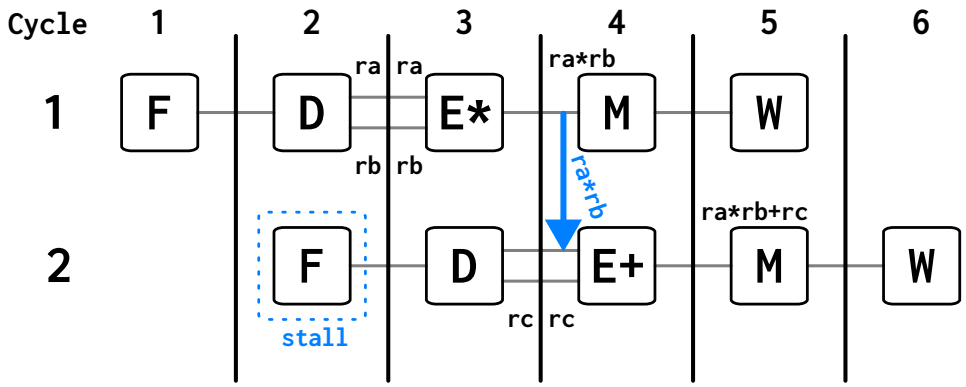


Figure 5.10: The ARM `mla` instruction is executed over two cycles.

ARM also has two multiply-accumulate instructions. One has a 32-bit result, the other 64 bit. The 32-bit `mla` is executed in two cycles. This can be implemented by reusing the adder inside the ALU. The flow of data is shown in Figure 5.10. In the first cycle, the product is calculated. In the second cycle, this result is added to the third argument. Since the ALU has two inputs, the product will have to be calculated on a cycle before the addition. The 64-bit multiply accumulate (`mlal`) has a more interesting microcode. It goes in four cycles. First, the low multiplication is performed, resulting in the low 32 bits of the product. Then, the result is added to `RdLo`, which will generate a carry for later. Then, the high multiplication is performed, resulting in the high 32 bits of the product. Finally, that result is added to `RdHi`, which uses the carry generated in the second step. This way, the 64-bit multiplication and addition is done correctly with a 32-bit data path in four cycles.

5.2.6. Load/Store

Although loads and stores are encoded very differently in ARM and RISC-V, the fundamental operation is quite similar. The value to be stored is either the lowest byte, lower halfword, or full word of a register. When loaded, halfwords and bytes can be sign extended or zero extended. The address of the store is the sum of two values.

In RISC-V, this will always be a register and an immediate, meaning the instruction uses two registers and can be done in one cycle. In ARM, the value can be the sum of two registers, meaning three registers (base, offset, and value) may be needed for a store. Since only two registers can be read in one cycle, this instruction will take two cycles. If the offset is an immediate rather than a register, the instruction will take one cycle. Writing back the result to a register will also take an extra cycle.

Load/Store Multiple

The ARM `ldm` and `stm` instructions are something completely unalike RISC-V. These instructions can load and store up to 16 registers in one instruction.

In Combi, these instructions take one cycle per register loaded or stored, plus another cycle if write back to the base register is specified. A 16-bit priority encoder is used to select the next register every cycle. This 16-bit priority encoder is made using two 4-bit priority encoders, as shown in Figure 5.11. A ‘divide and conquer’ approach is used, as is common for computer arithmetic [35]. The 16-bit input is split into four 4-bit nibbles. The nibbles each form the input of one OR gate, whose output is one if any bit in the nibble is set. By using a 4-bit priority encoder on the output of the four OR gates, the highest two bits of the 16-bit encoders output is generated. These two bits are also used to select which of the four nibbles go to the second 4-bit priority encoder. This 4-bit priority encoder then generates the lower two bits of the complete output. Using one more NOR gate, another output can be made which is true if and only if every bit of the input is clear.

This priority encoder scans the bottom 16 bits of the instruction, returning the index of the first ‘`1`’ it sees. This is used as the address of the register to dispatch to the execute stage and eventually to/from memory. As long as the priority encoder finds a set bit in the bit field, the instruction decoder will not advance its microcode. Every bit in the bit field is cleared as the register is dispatched. When the bit field is clear, the instruction decoder can advance its microcode counter. This is why the priority encoder also has an output which is set to ‘`1`’ if the bit field is clear.

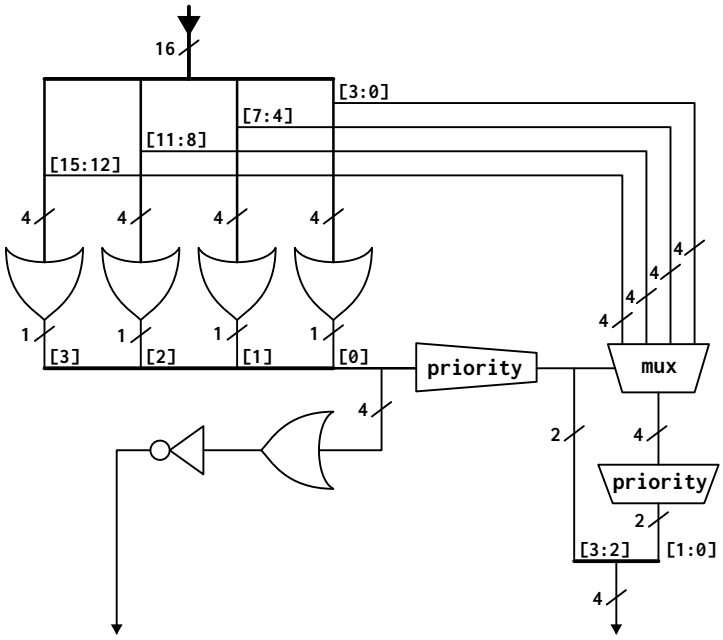


Figure 5.11: 16-bit priority encoder for ARM `ldm` and `stm`

Table 5.4: Instructions and their sources/destinations. [ldm] denotes the output of the priority encoder

Instruction	op1	op2	dest
add	[19:16]	[3:0]	[15:12]
b	15		
bl	15		14
(operand2)	[11:8]	[3:0]	
mla	[15:12]		[19:16]
stm	[19:16]	[ldm]	
ldm	[19:16]		[ldm]
str	[19:16]	[15:12]	

5.2.7. Sources and Destinations

Immediates in ARM and RISC-V are completely different, so the sign extending unit of Combi is essentially two separate units.

The registers used as sources and destinations in RISC-V are entirely homogeneous. The two sources are at bits 24 to 20 and 19 to 15 of the instruction, and the destination is at bits 11 to 7. Things are not so simple in ARM. Various options for the two source and destination registers are chosen by the Combi decoder depending on the ARM instruction decoded. Some examples are shown in Table 5.4.

The first register source can be hard coded to `r15` in case of a branch, at position 11 to 8 for a shift, at 15 to 12 for `mla` and `mlal`, or at 19 to 16 for normal data processing instructions. The second register source can be the output of the LDM priority encoder, the destination register in case of a load or store, or at position 3 to 0 for normal data processing instructions. The destination register can be the output of the LDM priority encoder, at position 19 to 16 for load/store write back or multiplication, hard coded to `r14` for branch and link, or at position 15 to 12 for normal data processing.

5.3. Results

In this section, the Combi architecture is presented as a candidate for a multi-ISA microarchitecture. The overhead in area and performance required to facilitate two instruction sets is shown. To do this Combi is compared to a RISC-V core with a similar architecture, Ibex - formerly known as zero-riscy [7]. Ibex was configured to have a single-cycle multiplication like Combi. Ibex also has a single-issue in order pipeline like Combi. However, it does not have a five stage pipeline. Ibex has three pipeline

stages: instruction fetch, decode and execute, and write-back. Also, Ibex has support for the C extension of RISC-V, which are a small set of 16-bit instructions. This makes the architectures a bit different. Ibex serves as a reference of a 'typical' RISC-V microarchitecture, unlike Combi which was designed to facilitate the ARM ISA too.

In addition to contrasting Combi to Ibex, different configurations of Combi are contrasted to each other. Ibex has architectural differences to Combi, which will affect the results. Thus accuracy is lost since it is desired to only see the effects due to implementing multiple ISAs. Contrasting different versions of Combi allows us to know the overhead to implementing multiple ISAs without any other architectural changes.

Combi was made for two targets: ARM and RISC-V. The architecture can be set to facilitate only one ISA by hard-wiring the `arm` bit of Section 5.2.1 to either '1' or '0'. This will let the synthesis tool infer all the circuitry of Combi that is not needed for that ISA and remove it, leaving only the essential parts. This way, the total area required for each configuration can be found. The performance lost when facilitating both instruction sets can also be determined this way.

5.3.1. Tests for Correctness

To verify the correctness of Combi, the RISC-V ISA test suite was used [31]. This is an official test suite of the RISC-V Foundation, and is therefore an authoritative resource on the functionality of a RISC-V core. In fact, another RISC-V design project - PicoRV32 - also uses this test suite to verify the correctness of its core [37]. The test suite contains one program for every instruction. To test the correctness of this core, the program corresponding to each and every instruction in the RV32IZmmul ISA was run. The fact that Combi passes these tests shows that Combi facilitates the RISC-V ISA completely.

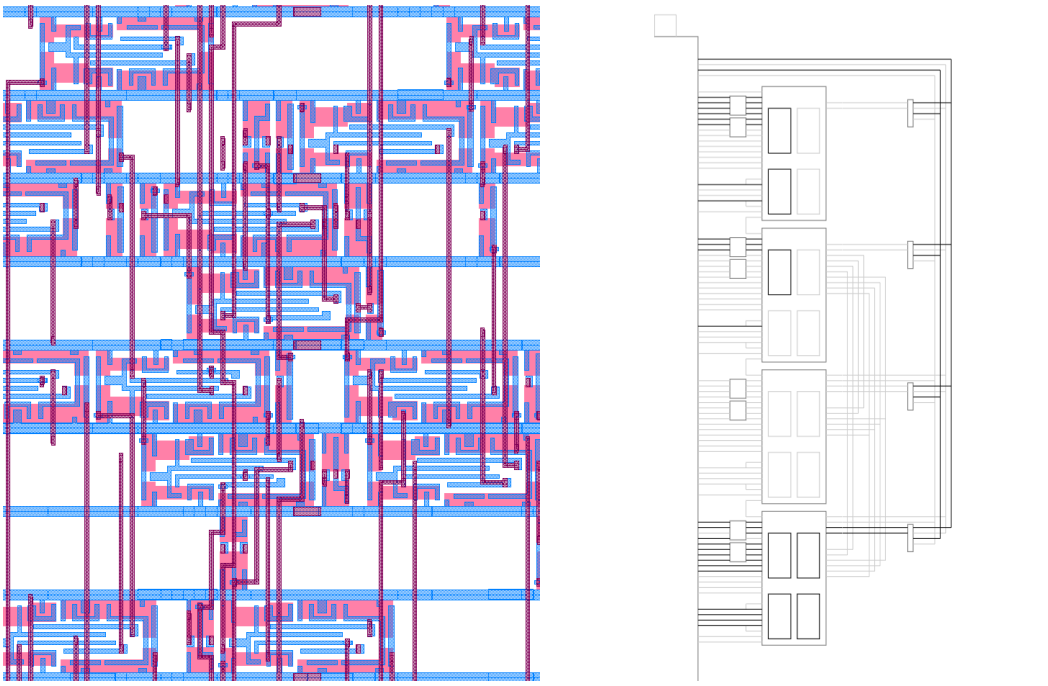
There was no official source for an ARM ISA compatibility test. Since ARM Ltd. makes the cores that implement the ARM ISA in-house, test suites are generally not made public. The RISC-V test programs are organized using macros. Each macro defines some preconditions, a command to execute, and postconditions. For example, to test the `add` instruction, the preconditions would be that the arguments are 1 and 2, and that the result is 3. Many such tests are performed on every instruction. To port this system to ARM, the macros themselves were ported first, and then the tests were extended to also test the ARM instructions that have no equivalent in RISC-V.

These tests were performed using Verilator, which transpiles the SystemVerilog description to C++. By compiling the C++ to machine code, a simulation can be run about 100 times faster than what would be possible using a simulator that interprets SystemVerilog [48]. By using a simulator, the complete state of the CPU can be inspected at every cycle. This made it easier to find problems in the microarchitecture, and fix them. The tests were run before every commit to the Combi repository, and comprehensively tested every instruction of both the ARM and RISC-V ISA.

For example, in section 5.2.3, we modified the instruction decoder to sometimes force the result of a previous instruction to be forwarded to the ALU. This was used to facilitate some multi-cycle instructions of the ARM ISA. However, when this was first implemented, some RISC-V tests that were passing before suddenly started failing. The failure mechanism was quite complicated, so it would have been much more difficult to catch without these comprehensive tests.

The cause of the error was as follows. Since the ARM instruction decoder still physically exists in the Combi microarchitecture, it was decoding the RISC-V instructions as if they were ARM instructions. The Combi microarchitecture will ignore the control signals from the ARM decoder when running RISC-V instructions, however the signal to force a result forwarding was not being ignored. This meant that, in the rare case that a RISC-V instruction happened to also decode to a multi cycle ARM instruction, the data path of the Combi pipeline would break. The test suites were instrumental to catching this failure, as it would surely have been difficult to identify later.

When the tests passed on the HDL sources, the tests were also run using a post-synthesis description of the microarchitecture. Experience teaches that designs which work before synthesis may not always work after synthesis, after all. But when the post-synthesis description also passes the tests, the result is that we can be confident the Combi microarchitecture does indeed run RISC-V and ARM programs correctly.



(a) ASIC realization (first three metal layers) (b) FPGA realization (one PFU, schematic from nextpnr)

Figure 5.12: A sample of Combi realized as an ASIC and on an FPGA

5.3.2. Experimental Setup

After implementing Combi in SystemVerilog, the design was synthesized on two platforms. First, a Field Programmable Gate Array (FPGA) platform, and second an Application Specific Integrated Circuit (ASIC) platform. The FPGA platform can be used to run the Combi realization at real-time speed. The ASIC platform is used to asses how Combi would perform if it was fabricated into an IC. On both platforms, area after place and route was measured to asses the size of Combi. The speed of Combi was assessed using the maximum clock frequency of the ASIC implementation.

The FPGA chosen was a Lattice ECP5 [26]. This FPGA is supported by the open-source Yosys tool chain [52][40]. It can therefore be programmed without needing to use a proprietary tool, which simplified the synthesis flow. The ECP5 itself is also affordable, and third-party development boards are available at a reasonable price. In fact, such a development board was already in our possession. It does not support very high speed designs, though. This will be reflected in the speed at which Combi can run on this FPGA.

The ASIC was also made with an open-source tool chain. The OpenROAD project allows anyone to synthesize an ASIC [32]. This project uses the NanGate45 cell library, which cannot be fabricated to a real IC. This library was made by NanGate, Inc using the NC State University FreePDK45 [14]. To fabricate to a real IC, a proprietary cell library would have to be used. Nevertheless, the OpenROAD results are similar to a real ASIC. We can compare the Combi realization to Ibex synthesized using the NanGate45 library. The mflowgen project hosts docker containers that have all the tools needed [29].

The maximum speed the ASIC can run at was determined by reducing the cycle time until the router could not realize the design. As nextpnr optimizes the placement of the cells, it report the Total Negative Slack (TNS). This number will slowly become zero as the routing improves. However, if the required clock frequency is too high, the TNS stops reducing after every generation of optimization, and the design cannot be realized. The clock period was decreased by units of $0.1\,\mathrm{ns}$ until nextpnr failed to resolve the timing constraint.

One part of the design was changed between the FPGA and ASIC realization of the design. In the FPGA design, the register file is implemented using D-type flip-flops. These flip-flops are a standard unit in an FPGA, so it is best to use them. However, this is not the case for an ASIC. D-type flip-flops are also available in the NanGate45 cell library, but they are significantly larger than latches. Therefore, the register file is implemented using latches instead. The design is based on the Ibex RISC-V core,

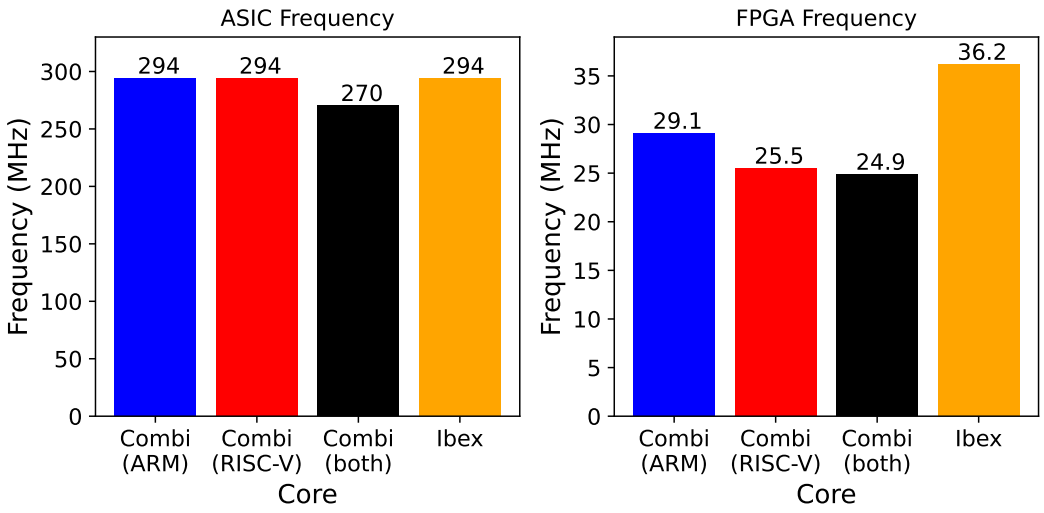


Figure 5.13: Maximum clock frequency of Combi in all configurations

which also uses latches [22]. A register using flip-flops is used at the write port of the register file to avoid a combinatorial cycle.

Measuring area on the ECP5 is less straightforward than an ASIC. Figure 5.12 shows a zoomed in part of Combi realized on the ECP5 and as an ASIC. On an Integrated Circuit, the cells (Fig. 5.12a, in pink) take up a certain amount of square micrometers, and the area used on the die itself can also simply be measured in square micrometers. However, an FPGA realizes the design in a different way. The ECP5 has generic cells that can be configured to have multiple functions. A cell could for example be a full adder, an arbitrary logic function, or a multiplexer. All of these functions would use the same area, the area equal to one FPGA cell. Thus, the base unit of area on an FPGA is a cell, not square micrometers. During routing, up to eight logic cells are assigned to a Programmable Functional Unit (PFU) in the ECP5. Additionally, each Programmable Functional Unit can have up to eight D-type flip-flops. The PFU shown in Figure 5.12b is split into four slices, which each have two logic cells on the left and two flip-flops on the right.

Ideally, all PFUs are filled with eight cells and eight flip-flops. However, this is not always possible. In Figure 5.12b, only four cells and two flip-flops are used. Some PFUs will be filled more than others, depending on specifics of the design. This is similar to how cells in an ASIC cannot always be packed right next to each other. Sometimes a gap between the cells is necessary for routing.

For the ECP5 realization, post routing area was taken to be the total count of PFUs used. This represents how much of the FPGA is dedicated to the realization of Combi. The remaining PFUs would be available for other functions. Counting the amount of used PFUs is done using a simple python script called by nextpnr after routing [40].

5.3.3. Performance Analysis

Combi was also compared to Ibex in terms of performance. The OpenROAD synthesis flow can be used to predict the maximum clock speed these designs can run at.

The maximum clock speeds determined with OpenROAD are shown in Figure 5.13. The RISC-V and ARM versions of Combi run at a clock speed as fast as Ibex. Ibex was configured to also have a single-cycle multiplier, like Combi. It also has a single-issue in-order pipeline design. This makes Ibex comparable to Combi. However, Ibex has only three stages, while Combi has five.

This change in microarchitecture may have consequences for the performance in terms of execution time of a given program. Usually, a benchmark program is used to evaluate the performance loss due to such differences. However, since the C standard library was not ported to Combi, this is difficult.

We can still conclude that, when configured for two instruction sets, Combi will run slower than for one instruction set. In this case, the microarchitecture of Combi does not change between configurations. Therefore, the only loss in performance is due to a slower clock speed. In this case, Combi loses $8.9\,\%$ of performance when configured for both instruction sets compared to being configured for only

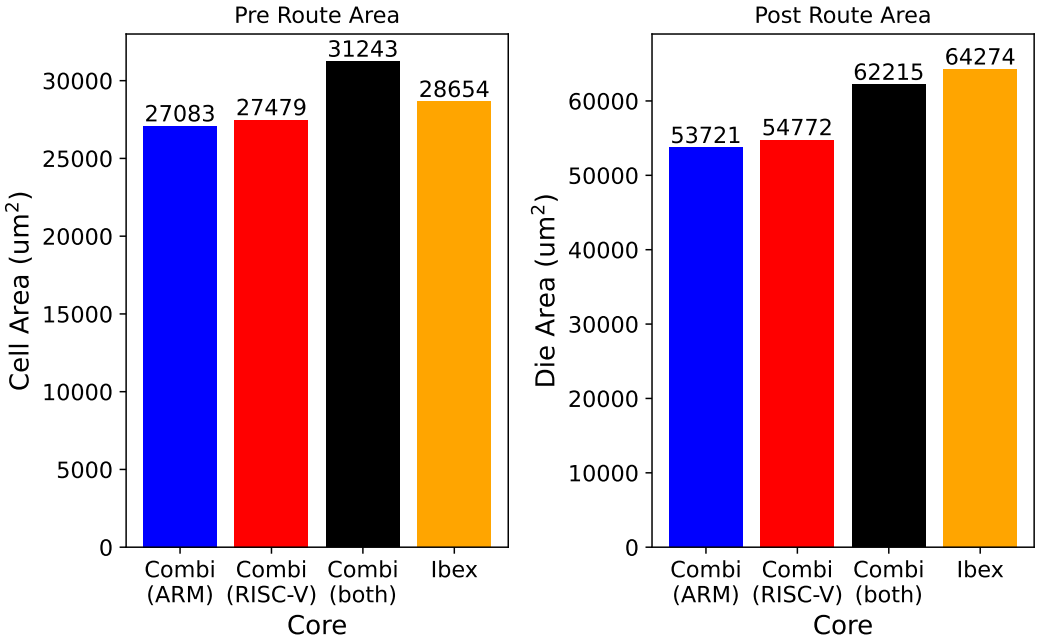


Figure 5.14: Area used by Combi in all configurations ($45\,\mathrm{nm}$ process), lower is better

one ISA.

Using nextpnr, the maximum frequency on the FPGA can also be found. The maximum frequency of all Combi configurations is shown in Figure 5.13. On the FPGA, Combi is fastest when configured to only implement ARM instructions. The RISC-V and combined ISA versions are about the same speed. This is not the case for the ASIC implementation, where all three are equally fast.

After some critical path analysis, the reason for this was found. The critical path of Combi is the sequence of logic that takes the longest time to propagate from an output to an input. Inside the Combi processors, these would also be the outputs and inputs of flip-flops. The longest chain determines the maximum clock frequency of the entire design. For Combi, the critical path goes through the Condition Unit shown in Figure 5.5.

Recall from Section 5.2.2 that branches in RISC-V and ARM are different in a key aspect. ARM branches are based on the output of the flag register, which is a flip-flop. However, the RISC-V branches are based on the output of the ALU directly, which is a combinatorial circuit. As such, the data path for RISC-V instructions goes though the ALU and the Condition Unit in one cycle. This is different from ARM instructions, which have an instruction flags register in between this path. Thus, when configured for ARM instructions, Combi can have a smaller cycle time. But when running RISC-V instructions, the microarchitecture needs to have this longer path, and is therefore slower.

5.3.4. ASIC Area Usage Analysis

The results of this analysis are shown in Figure 5.14. The three configurations of Combi are compared to each other and to Ibex, a RV32IMC core that is used as a reference. The overhead of combining both instruction sets is only 13.7% compared to the RISC-V configuration of Combi.

Combi uses more cell area than Ibex, yet uses less die area after routing. In a sense, the Combi microarchitecture is more compact than Ibex. We can conclude that Ibex has more gaps between the cells than Combi. Ultimately, the die area after routing determines the size of the Integrated Circuit. Therefore, Combi can be considered to be smaller than Ibex. In contrast, the relative difference between the combined ISA version of Combi and the RISC-V configuration did not change.

Figure 5.15 shows how much area is used by Combi by stage and by functional unit. Do note that the register file is part of the decode stage, and the multiplier is part of the execute stage. The wedges labeled ‘Execute’ and ‘Decode’ represent everything else in those stages. Also note that the memory units of this design are external, which is why the ‘Fetch’ and ‘Memory’ stages are so small. These stages interact with memory units that are not a part of Combi, but rather part of a hypothetical larger

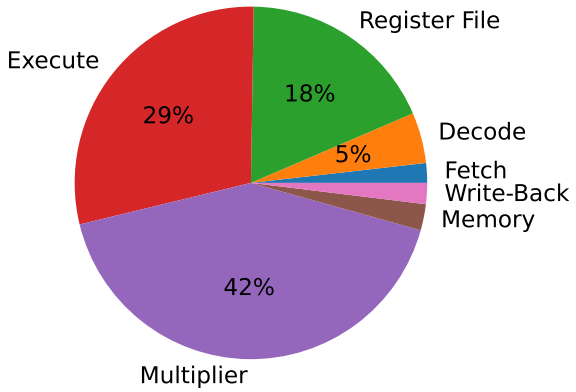


Figure 5.15: Pre-route area used by Combi per stage (ASIC)

System on a Chip (SoC).-

Since Combi decodes not one but two instruction sets, one might expect the decode stage to be large. However, the decode stage is only 5% of the total area. The biggest contributors to area are the register file, execute stage, and multiplier. The register file and multiplier are entirely shared between the RISC-V and ARM instruction streams, which shows how much hardware can be reused when facilitating two instruction sets.

5.3.5. FPGA Area Usage Analysis

The area that Combi uses in the RISC-V, ARM, and combined configuration is shown in Figure 5.16. Pre-route area is measured in the amount of reconfigurable cells of the ECP5 used. These cells might represent a four-input Look Up Table (LUT), a D-type flip-flop, a full adder, or other features which map to an FPGA cell. Post route area is measured in Programmable Functional Units (PFUs). These represent the resources that are used by the FPGA to realize the Combi microarchitecture. A design may need more PFUs if it has an architecture which does not neatly map to the FPGA fabric.

In the ASIC area results (Figure 5.14), we saw little difference in the ratio between the pre- and post-route area. In Figure 5.16, we do see a change. Before placement and routing, the design that only implements RISC-V is 27% smaller than the configuration that facilitates both ISAs, and the ARM version is 23% smaller. After placement and routing, the RISC-V design is 19% smaller and the ARM area is reduced by only 11%. This is a notable decrease in area overhead after placement and routing. You could say the combined architecture is more ‘compact’ than the designs which implement only one instruction set. This might be because a major contributor of the extra cells in Combi when implementing both ISAs is multiplexers. These multiplexers act as switches between functions that are unique to ARM and RISC-V. Since they can be placed next to the cells that realize the function in question, these cells may fill in some of the gaps inside a PFU.

Compared to Ibex, Combi takes up significantly less area, yet is also significantly slower. This is an unexpected result, considering they were quite similar when using the ASIC as a target. Ibex was reconfigured to use flip-flops for its registers, since these work better than latches on an FPGA. The FPGA platform was mostly designed for testing though, so this strange result is acceptable. Perhaps with some more careful analysis of the FPGA design flow the cause of this difference would become clear. However the comparison with Ibex was not as thoroughly done on the FPGA platform as it was on the ASIC.

5.4. Comparison with State of the Art

It is hard to estimate the size of a comparable ARM core. The last ARM core to implement the ARMv4 ISA was the ARM7 [2]. No official documentation of ARM can be found on the area of this core. Secondary sources are very mixed. According to Fujitsu, this core has an area of $1.1\,\mathrm{mm}^2$ with a $180\,\mathrm{nm}$ technology node [15]. According to Dr. Umapathi, the area is $0.53\,\mathrm{mm}^2$ with a $180\,\mathrm{nm}$ technology [45].

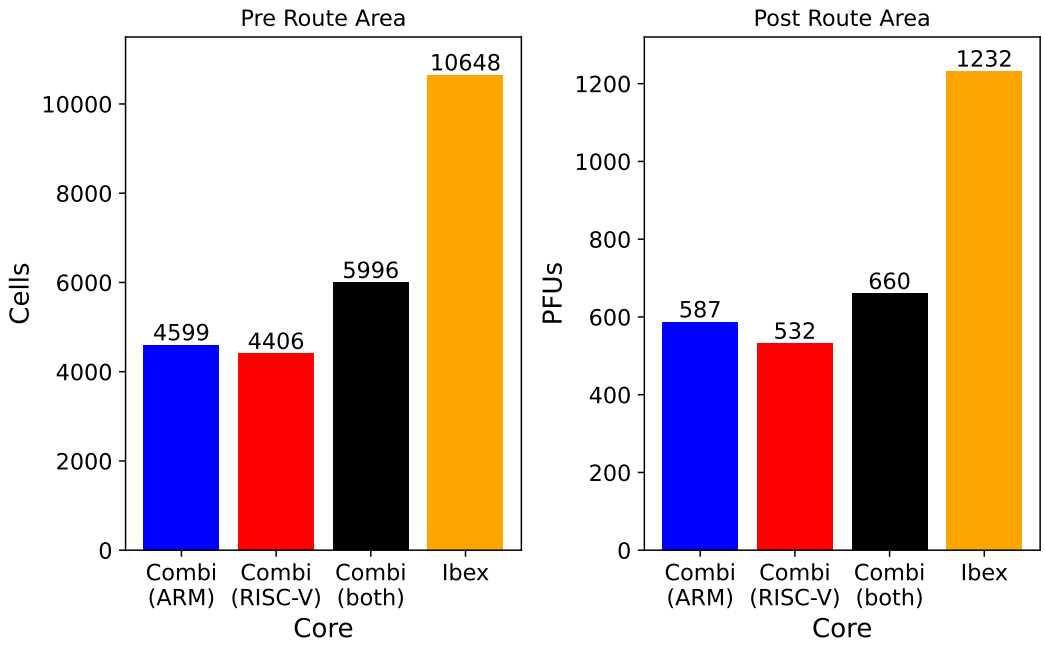


Figure 5.16: Area used by Combi in all configurations on the FPGA, lower is better

And finally, according to Domingo *et al.*, the area is $2.1\,\mathrm{mm}^2$ with a $250\,\mathrm{nm}$ technology [12]. These are all very different figures, so it is impossible to know based on this information exactly how big the ARM7 is. Therefore, Combi is compared with another RISC-V core, Ibex. Since Ibex is open source, it can simply be tested using the exact same methods as Combi.

The comparison to Ibex in Figure 5.14 shows that Combi in the RISC-V only configuration is comparable to other RISC-V cores. Note that Ibex has some features Combi does not have, namely support for the compressed (C) instruction set[1], and a few status registers for operating system support that are out of scope for this thesis. This explains why Ibex uses slightly more area than Combi.

5.4.1. Comparison to Multi ISA Systems

Compared to RVAM16, Combi uses 32% less area. However, RVAM16 uses a 16-bit data path, which leads to a 30% reduction in performance according to Huang *et al.* [21]. This is a clear performance-area trade off. While RVAM16 is preferable for low-cost designs, Combi is better suited for higher performance designs. Also note that RVAM16 was designed to run at 100 MHz, and Combi runs at 270 MHz. This once again shows that Combi is a higher-performance processor. Rosetta 2 has 32% to 42% overhead, a bit worse than RVAM16 [27]. It may take zero area overhead, but the performance is once again lost.

Heterogeneous ISA multi-core systems can only work at maximum capacity if the distribution of workloads over the facilitated ISAs exactly match the available compute power per ISA. Combi does not have this limitation. If a Heterogeneous ISA system has 2 RISC-V cores and 2 ARM cores, and 4 threads to run using the ARM ISA, only two out of four cores can be used. If a 4-core Combi system had the same workload, all four cores could run in ARM mode. Combi is therefore more versatile than a heterogeneous ISA system. However, a multi-ISA Combi core is 13% larger than a comparable single-ISA core. Therefore, each core in a Heterogeneous ISA multi-core system will be slightly smaller than a multi-core Combi system. This is a downside of using the Combi microarchitecture.

[1]The C instruction set is a subset of the RV32I ISA that is encoded in 16 bits instead of 32 bits

6

Conclusion

This chapter provides a summary of the achievements of this thesis. Additionally, it presents some potential future research directions. Section 6.1 presents the summary of this work, and Section 6.2 shows the possibilities for future work.

6.1. Summary

We have systematically compared the RISC-V and ARM ISA. By dividing the instructions of both ISAs into the same categories, we could identify the similarities and differences of the architectures. We compared the register instructions, memory instructions, and branch instructions of both sets. Many instructions were shared between ARM and RISC-V, and the hardware needed to facilitate the ones that were not is small. Even branches, which work very differently on ARM and RISC-V, could be facilitated with hardware that was almost entirely used by both architectures. There were two main differences. The first was that ARM branches use flags, but RISC-V has no flag register. The second was that ARM has some instructions which cannot execute in one cycle, but RISC-V does not. For example, most register instructions on ARM can be supplemented by an additional shift in the same opcode. If these instructions use a constant shift amount, the ALU of Combi can process it in one cycle. However, if the shift amount is specified in a third register, the ALU has to take two cycles. This is because the ALU has only two input ports, which was a design decision of Combi.

RISC-V does have more registers than ARM, which has consequences for calling ARM functions from RISC-V and vice versa. We have described a general methodology for switching between instruction sets, keeping in mind the differences in ABI. We have applied this methodology to RISC-V and ARM, and concluded that RISC-V and ARM binaries can interoperate without any modification. This can be achieved using the linker to insert code when a program is loaded, or in some cases binary interoperability can be resolved purely in hardware. Different methods have been presented, depending on the level of hardware support, properties of both the calling and called ISA, and the signature of the routine called. If the function signatures are sufficiently small, or both ISAs have the same amount of argument and return registers, the call can be resolved using only hardware. Otherwise, some extra program code would have to be inserted during the linking.

Additionally, three methods of switching between ISAs have been presented. Switching ISAs is necessary for library calls which cross an ISA boundary, or for running binaries of both ISAs in a shared operating system. These three methods are using an unused instruction to switch, using the paging system to detect the ISA of the program, or using statistical properties to detect the ISA. Using an unused instruction can work even without a memory mapping unit, but requires code to be inserted wherever the ISA changes. Using the paging system requires no modification of the binary, but requires a form of memory mapping which may be absent in some systems, for example Combi. Finally, using statistics requires neither inserting any code, nor using a memory mapping unit. It may be possible to do so reliably in the case of ARM and RISC-V, however it could still fail. Failing to decode even one instruction can be fatal to the correct execution of a program and must be avoided.

Finally, we have created Combi, a microarchitecture that can run both RISC-V and ARM binaries. To achieve this, Combi must use 13% more area and have a cycle time that is 9% slower than when it

is running only RISC-V or only ARM code. Combi is comparable in area to a typical RISC-V processor, but slower. But Combi is faster than a multi-ISA microarchitecture that tries to preserve its area, and is faster than binary translation in software. Overall, this thesis shows that a microarchitecture can support multiple ISAs at a competitive speed without nearly becoming as big as two separate processors. This can be done by sharing most of the hardware that is used to implement the ISAs, thus facilitating both with little extra required hardware.

6.2. Future Work

There is still work left to be done on multi-ISA microarchitectures. In this section, we will discuss some of the questions left unanswered and possible future work.

Software Support for Multiple ISAs

- Support for linking binaries of different ISAs: in this work we developed a methodology for switching instruction sets, but not a linker capable of performing any proposed method. For example, being able to statically link a new program targeting RISC-V with a library written in ARM that has no source code available may be necessary if a platform is transitioning from one ISA to another. This might even be done dynamically instead. In this case, the performance of the linker itself is also a concern, since the program will be linked before it is run. One can investigate the extra load time due to linking two binaries of different sizes, or the extra run time due to the code inserted by the linker at every ISA switch.

- Kernel support for programs of different ISAs: what this work has not touched upon at all is the possibility of an operating system that can have programs running of multiple different ISAs. This will require the kernel to keep track of which ISA every program is running in, and ensuring the ISA is switched back to the one the kernel uses when a program is interrupted. Multitasking operating systems will regularly interrupt a program to allow other programs to execute, but programs may also be interrupted in the case of an exception. Examples of errors that interrupt the program include a memory access out of the bounds of allocated memory or executing a binary instruction that is not part of the ISA. In these cases, the operating system may also need to switch back to its own ISA before handling the exception.

- Emulation of system calls: in the presence of an operating system, programs may also perform system calls. These system calls can be thought of as requests to the operating system from the program, for example allocating memory, writing to a file, or sending data over the Internet. System calls also differ between ISAs. If an operating system runs programs of a different ISA, some system must exist to ensure the operating system can understand the system call from that foreign ISA.

Exploring Different ISAs

- Combining CISC and RISC ISAs: in the introduction of this work, we stated that implementing a CISC microarchitecture would not be feasible. This leaves the question open as to what the limitations of combining a RISC and a CISC ISAs would be. This problem has been addressed by software simulation in earlier work such as Rosetta 2 and the Crusoe processor, but has seen little attention at a hardware level. If a CISC microarchitecture were to split its complex instructions into simple micro instructions, it may be able to support a RISC ISA too.

- Further classification of ISAs: this work has looked at three classes of instructions: register instructions, memory instructions, and branch instructions. However, more kinds of instructions are also common in ISAs. For example, some ISAs have instructions that operate on memory directly, such as adding a register to an address in memory. Floating point instructions are also common, which may share a lot of similarities between ISAs. Finally, ISAs for high performance computing may support Single Instruction Multiple Data extensions. These extensions may be work very differently, and it will be interesting to know if a common microarchitecture can facilitate multiple.

- Combining variable and fixed width ISAs: ARM and RISC-V both have 32-bit instructions, which made the fetch stage of Combi almost identical for both instruction sets. If instruction sets were

combined that did not have the same length encoding, the fetching system becomes more interesting. It is currently unknown what the consequences of such a system would be for the area or performance of a processor.

Exploring More of the Microarchitecture

- Other architectures: Combi is a single issue, in-order pipeline microarchitecture. As stated in the introduction, other microarchitectures have been made that implement RISC-V. It may be interesting to see how well the Combi approach works on, for example, a superscalar microarchitecture. Superscalar architectures use more area than single issue pipelines, so perhaps the overhead would be less. On the other hand, instructions of different ISAs may interact in ways that were not considered in this thesis when dispatched simultaneously.
- Memory models: many processors have a memory system that ensures programs only interact with memory in appropriate ways. For example, the memory of concurrent programs is isolated such that program A cannot interfere with the memory of program B. This memory model is often also different between processors, and was not considered in this thesis.
- Power management: in Chapter 5, it was shown that a control line determines if the instruction executing is a RISC-V or an ARM instruction. It may be beneficial to disable the hardware that is only needed for ARM instructions when RISC-V instructions are executing, or vice versa. This could lead to a design that uses less power.

Bibliography

[1] ARM Limited. *ARM Architecture Reference Manual*. Tech. rep. ARM DDI 0100E. 2000.

[2] ARM Limited. *ARM7TDMI-S Data Sheet*. Data Sheet ARM DDI 0084D. 2000.

[3] ARM Limited. *Procedure Call Standard for the Arm® Architecture*. Tech. rep. AAPCS32. 2024Q3. (Visited on 02/09/2025).

[4] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. Aug. 2014. (Visited on 03/26/2025).

[5] Jeremy Bennett. *Embench*. https://github.com/embench/embench-iot. Mar. 2025. (Visited on 03/17/2025).

[6] Dileep Bhandarkar and Douglas W. Clark. "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. New York, NY, USA: Association for Computing Machinery, Apr. 1991, pp. 310–319. ISBN: 978-0-89791-380-5. DOI: 10.1145/106972.107003. (Visited on 03/17/2025).

[7] Greg Chadwick et al. *Ibex*. https://github.com/lowRISC/ibex. Mar. 2025. (Visited on 04/01/2025).

[8] Kito Cheng and Jessica Clarke. *RISC-V ABIs Specification*. Tech. rep. RISC-V International, Aug. 2024. (Visited on 01/30/2025).

[9] David Chisnall. "How to Design an ISA". In: *ACM Queue* 21.6 (Jan. 2024). (Visited on 09/10/2024).

[10] Pasquale Davide Schiavone et al. "Slow and Steady Wins the Race? A Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications". In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Thessaloniki: IEEE, Sept. 2017, pp. 1–8. ISBN: 978-1-5090-6462-5. DOI: 10.1109/PATMOS.2017.8106976. (Visited on 04/01/2025).

[11] J.C. Dehnert et al. "The Transmeta Code Morphing/Spl Trade/ Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges". In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* Mar. 2003, pp. 15–24. DOI: 10.1109/CGO.2003.1191529. (Visited on 07/03/2024).

[12] M.E. Domingo et al. "High-Level Implementation of an ARM7 Microprocessor with Multicore Capabilities". In: *TENCON 2007 - 2007 IEEE Region 10 Conference*. Oct. 2007, pp. 1–4. DOI: 10.1109/TENCON.2007.4429057. (Visited on 03/31/2025).

[13] J. Dougall. *Why Is Rosetta 2 Fast?* Nov. 2022. (Visited on 03/17/2025).

[14] *FreePDK45 | NC State EDA*. https://eda.ncsu.edu/freepdk/freepdk45/. (Visited on 04/01/2025).

[15] Fujitsu. *ARM7TDMI Processor Core*. Data Sheet ASIC-FS-20831-4/00. USA. (Visited on 03/31/2025).

[16] Steve Furber. *ARM System-on-Chip Architecture*. Subsequent edition. Harlow Munich: Addison Wesley, Aug. 2000. ISBN: 978-0-201-67519-1.

[17] Tom R Halfhill. "Transmeta Breaks X86 Low-Power Barrier: 2/14/00". In: *Microprocessor Report* (Feb. 2000), pp. 9–18.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture Book Companion*. https://www.elsevier.com/books-and-journals/book-companion/9780128119051. 2011. (Visited on 03/17/2025).

[19] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th edition. Morgan Kaufmann Publishers In, Oct. 2011. ISBN: 978-81-7867-266-3.

[20] Matthew Honan. "WWDC: Apple Drops IBM PowerPC Line for Intel Chips". In: *Macworld* (June 2005). (Visited on 03/17/2025).

[21] Libo Huang et al. "RVAM16: A Low-Cost Multiple-ISA Processor Based on RISC-V and ARM Thumb". In: *Frontiers of Computer Science* 19.1 (Jan. 2025), p. 191103. ISSN: 2095-2228. DOI: 10.1007/s11704-023-3239-x. (Visited on 08/08/2024).

[22] *Ibex/Rtl/Ibex_register_file_latch.Sv*. https://github.com/lowRISC/ibex/blob/master/rtl/ibex_register_file_latch.sv (Visited on 04/01/2025).

[23] Martin Kindall. *Risc-v-Single-Cycle*. https://github.com/martinKindall/risc-v-single-cycle/tree/main. Dec. 2022. (Visited on 04/01/2025).

[24] T. Kistler and M. Franz. "Continuous Program Optimization: Design and Evaluation". In: *IEEE Transactions on Computers* 50.6 (June 2001), pp. 549–566. ISSN: 00189340. DOI: 10.1109/12.931893. (Visited on 07/03/2024).

[25] Alexander Klaiber. *The Technology Behind Crusoe™ Processors*. Tech. rep. Transmeta Corporation, Jan. 2000.

[26] Lattice Semiconductor. *ECP5 and ECP5-5G Family Data Sheet*. Tech. rep. FPGA-DS-02012-3.3. Jan. 2024.

[27] Xinyu Li et al. "BTBench: A Benchmark for Comprehensive Binary Translation Performance Evaluation". In: *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. May 2024, pp. 36–47. DOI: 10.1109/ISPASS61541.2024.00014. (Visited on 03/12/2025).

[28] Saqib Madni, Muhammad Shafiq, and Haroon Ur Rashid. "ARMINTEL: A Heterogeneous Microprocessor Architecture Enabling Intel Applications on ARM". In: *2020 17th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. Jan. 2020, pp. 370–377. DOI: 10.1109/IBCAST47879.2020.9044572. (Visited on 07/10/2024).

[29] *Mflowgen*. https://github.com/mflowgen/mflowgen. Apr. 2025. (Visited on 04/01/2025).

[30] Patrick Moorhead. "Apple MacBook Pro 13" M1 Review- Why You Might Want To Pass". In: *Forbes* (Nov 21, 2020, 03:16pm EST). (Visited on 04/08/2025).

[31] Tim Newsome, Andrew Waterman, and Yunsup Lee. *Riscv Tests*. https://github.com/riscv-software-src/riscv-tests/tree/master. Mar. 2025. (Visited on 04/01/2025).

[32] *OpenROAD*. https://github.com/The-OpenROAD-Project/OpenROAD. (Visited on 04/01/2025).

[33] Charles Papon. *VexRiscv*. https://github.com/SpinalHDL/VexRiscv. 2021. (Visited on 08/19/2024).

[34] Charles Papon and Yindong Xiao. *SpinalHDL*. https://github.com/SpinalHDL/SpinalHDL. Mar. 2025. (Visited on 03/12/2025).

[35] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. USA: Oxford University Press, Inc., July 1999. ISBN: 978-0-19-512583-2.

[36] David Patterson and 2019. *Embench™: Recruiting for the Long Overdue and Deserved Demise of Dhrystone as a Benchmark for Embedded Computing*. June 2019. (Visited on 03/19/2025).

[37] *Picorv32/Tests*. https://github.com/YosysHQ/picorv32/tree/main/tests. (Visited on 04/05/2025).

[38] Sarah L. Harris and David M. Harris. *Digital Design and Computer Architecture*. ARM edition. Morgan Kaufmann, May 2015. ISBN: 978-0-12-800056-4.

[39] Sarah L. Harris and David M. Harris. *Digital Design and Computer Architecture*. RISC-V edition. Morgan Kaufmann, Oct. 2021. ISBN: 978-0-12-820064-3.

[40] David Shah et al. "Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2019, pp. 1–4. DOI: 10.1109/FCCM.2019.00010. (Visited on 03/19/2025).

[41] Stephen Shankland. "Apple Is Giving Macs an Arm Chip Brain Transplant after 14 Years Relying on Intel". In: *CNET* (June 2020). (Visited on 03/17/2025).

[42] Tom Shanley. *X86 Instruction Set Architecture*. Mindshare Press, 2010. ISBN: 978-0-9770878-5-3.

[43] Anton Shilov. "Transmeta Quits Microprocessor Business". In: *X bit labs* (Feb. 2007). (Visited on 03/17/2025).

[44] Richard Stallman and the GCC Developer Community. *GCC 14.2 Manual*. GNU Press, Aug. 2024. (Visited on 02/03/2025).

[45] Dr. N. Umapathi. “ARM Processor Architecture”. Lecture Slides. Karimnagar, India: Jyothishmathi Institute of Technology & Science, Feb. 2020. (Visited on 03/31/2025).

[46] Ashish Venkat, Harsha Basavaraj, and Dean M. Tullsen. “Composite-ISA Cores: Enabling Multi-ISA Heterogeneity Using a Single ISA”. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2019, pp. 42–55. DOI: 10.1109/HPCA.2019.00026. (Visited on 07/05/2024).

[47] Ashish Venkat and Dean M. Tullsen. “Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor”. In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 121–132. ISSN: 0163-5964. DOI: 10.1145/2678373.2665692. (Visited on 03/17/2025).

[48] *Verilator*. https://www.veripool.org/verilator/. (Visited on 04/03/2025).

[49] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Standard 20240411-draft. RISC-V Foundation, Dec. 2019. (Visited on 01/30/2025).

[50] Claire Wolf. *PicoRV32*. https://github.com/YosysHQ/picorv32. Apr. 2025. (Visited on 04/01/2025).

[51] Li Xinbing. *SSRV(Super-Scalar RISC-V)*. https://github.com/risclite/SuperScalar-RISCV-CPU. Aug. 2020. (Visited on 04/01/2025).

[52] *Yosys Headquarters*. https://github.com/YosysHQ. (Visited on 04/01/2025).