Experimental characterization of the effect of various ForwardCom architecture parameters on microprocessor performance

Written by
Kai Rese
964856

Bachelor Thesis for obtaining the degree of
`Bachelor of Science`
at
Hochschule Niederrhein

First supervisor: Prof. Dr. Edwin Naroska
Second supervisor: Prof. Dr. Agner Fog

April 11, 2023

Abstract

This thesis provides a first examination of some experimental design parameters of the ForwardCom instruction set architecture. It tests the performance impact of these parameters via an experiment. The experiment was performed using two benchmark programs, simulated on two CPU models via the General Purpose Core Architecture Simulator. The parameter impact is derived by allowing or disallowing the corresponding feature, then testing each feature combination and extracting the performance difference of allowing each feature for each combination of the others. The used benchmarks programs are N-body and Mandelbrot. The tested parameters are multi-word formats, ALU-and-branch instructions, instructions with a memory operand, variable vector length support, and vector loops.

Due to the small sample size of benchmark programs and CPU models, the experiment results contain some large variations. Because neither a compiler nor a complete hardware implementation of ForwardCom exist yet, the thesis cannot provide a predictive result, but it can provide a first rough indication as a pilot study. Taking this limitation into account, the results show a significant positive performance impact of each tested feature, as well as for feature combinations. The results of all features combined range from a 26% improvement up to a 80% improvement, ignoring the increased flexibility for size in vector unit implementations. Taking that into account, improvements of up to 623% are observed.

Table of contents

List of Figures

Chapter 1

Introduction

Casual computer users usually do not care about the Instruction Set Architecture (ISA) of the computers they use, or even know what an ISA is. They care about the operating system instead, which makes little difference for buying decisions, as the popular operating systems have one dominant ISA they run on: Windows runs on x86, Android and iOS run on ARM. macOS recently transitioned from x86 to ARM, but as Apple owns the ecosystem, they had the power to perform an easy transition. Microsoft also tries to put Windows on ARM devices, but the success has been limited so far [1].

The computing environment looks different for servers and software developers. x86 is the highest share, but ARM-based servers are on the rise [2]. The dominating operating system for servers is Linux, which can run on a variety of ISAs, including x86, ARM, RISC-V, and IBM Power. Efforts are made to make software independent from the ISA, but its choice still defines most of the available software ecosystem.

The dominating ISAs are owned by big companies that oversee their development and charge hardware vendors for building compatible processors. In the specific case of x86, Intel has not issued a new license to anyone in at least 20 years, so there are only three companies including Intel that are even allowed to build compatible processors. ARM is more liberal with their licensing, but has recently announced plans for a price increase across the board [3].

The dependence on these companies, especially considering the financial burden, is not ideal in many situations. As a consequence, a few new royalty-free ISAs started development, such as RISC-V or ForwardCom. While RISC-V aims to be usable for all kinds of processors and puts the focus on modularity and proven Reduced Instruction Set Computer (RISC) principles, ForwardCom aims to be

an ideal ISA for high performance general-purpose processors and incorporates many experimental ideas [4, ch. 1.1]. These ideas result in design parameters that provide theoretical benefits, but none of them have been tested in practice. This thesis aims to provide an indication of the practical performance benefits for some of the ForwardCom parameters.

The thesis is structured into four main parts. Chapter 2 explains non-basic concepts that are used throughout the thesis. Secondly, chapter 3 performs a closer look at the ForwardCom ISA and its experimental features. The setup of the experiment, including the parameters tested, the benchmarks and the tested Central Processing Unit (CPU) models, is described in chapter 4. Finally, chapter 5 presents the results, including some additional investigation for cases that did not conform to the expectations. The thesis is rounded off with a conclusion and a few future prospects in chapter 6.

Chapter 2

Concepts

The thesis touches on several non-basic computer concepts, which are important to be understood in order to fully make sense of the content. This chapter explains these concepts in detail.

2.1 Instruction Set Architecture

An ISA is a specification for the interface between a program and the hardware it executes on.

An ISA originally described the interface to a specific computer, but with the advent of microcode (see chapter 2.2) and computer families with compatible members such as the IBM System/360 [5, p. 5], the description shifted to that of an abstract machine [6, p. 7]. This machine can be implemented in different ways, but the common specification makes the implementations compatible with each other.

There are different kinds of computational hardware, but the rest of this chapter specifically describes ISAs of recent general-purpose processors, since they are the relevant category for this thesis.

Privilege levels

Most architectures define a set of privilege levels in which a program can run. The privilege level defines which instructions are allowed to be used and what state it can access. The highest privilege level is typically used by operating systems or firmware and grants full access to the hardware [7, p. 9]. User programs only get

access to a part of the available memory, which may also be restricted to read-only mode. They also are forbidden from using privileged instructions. Finally, they also cannot directly access hardware resources, but use a set of functions supplied by the operating system or device drivers.

Memory model

The memory model describes the structure and properties of the memory address space.

Current systems with 64 bit addresses have several orders of magnitude lower main memory capacity than addressable. The free address space can serve multiple functions: it can be mapped to storage memory that is not loaded, triggering the load on access; it can be mapped to external hardware ports or directly to externally attached memory [8, p. 19-2], or it can just be unused.

It can also define a permission model for different portions of memory, as well as some mechanism of address translation to enable the use of virtual addresses.

Instructions

Instructions, together with plain data, are the basic primitive that comprises programs. Depending on the design of the architecture (see chapter 2.2), they can be simple or complex. In the former case, they signify tasks such as transforming the type of a value, combining two (or sometimes three [9, FMADD]) values by applying an arithmetic or logical operation, reading from or writing to memory, calling into and returning from functions, jumping or branching conditionally. Depending on the specific architecture, privileged instructions can access connected hardware, manage the state of memory and other programs, change the hardware state, and/or read and write sensitive data.

Instructions are encoded into binary data that can have a fixed size or have one of multiple sizes, depending on the architecture. An encoded instruction contains an identifier of the instruction, as well as its operands. It can also contain information about the instruction length and operand types, but this information can also be included in the identifier, or be fixed by the architecture [10, p. 8].

An operand can be encoded as either an immediate value, which is literal data, or an index to a register inside a set. If the instruction has a memory operand, its address can be calculated from a multitude of encoded registers and immediate values, as well as special registers such as the instruction pointer. The addressing mode is also defined by the instruction.

Registers

Registers are small bits of memory that are directly addressable. They can be special parts of main memory, but their small size makes it inexpensive to implement them as fully separate memory units. Their main purpose is to save encoding space: Some registers are directly used by special instructions, while others are grouped in a set and accessed by their index in that set. As the amount of registers in such a set is several orders of magnitude smaller than the available memory addresses, their address can be encoded in much fewer bits.

An ISA typically contains a set of general-purpose registers, some individual special ones and sometimes also one or multiple sets of additional number registers [8, p. 3-2]. General-purpose registers are used by most instructions and hold 64 bits of data on a modern standalone client device. Depending on the instruction, the data is interpreted as an integer number or an address. The additional number registers are usually dedicated to either floating-point numbers, or vectors of multiple numbers. One reason for separate registers is that there are usually instructions implicitly using them, so more registers can be provided within the same encoding space. They can also provide a bigger size of memory compared to general-purpose instructions. Finally, they are likely to be used by different parts of the hardware, so it makes sense to also divide them conceptually.

The special registers each hold an individual meaning, with a few of them existing in most common ISAs. The most important one is called the instruction pointer or program counter, which contains the address of the instruction to be executed [10, p. 13]. It has a fair number of instructions interacting with it, since jumps, calls, and returns all directly write to the instruction pointer. Then there normally is a stack pointer, which contains the address of the last pushed element of the stack and which can also be a general-purpose register instead of a special one. A stack can save local function data by pushing them onto it, which enables functions to save their own data overwriting no data of the function they got called from. There are more special registers usually, but they are very individual to the respective ISA.

Addressing modes

Instructions using an address into memory can calculate that address in different ways, which are called modes. The support of addressing modes varies wildly; some ISAs, such as ARMv8-A, even integrate arithmetic functionality in some modes [6, p. 33]. In terms of the actual calculation, modern ISAs tend to only use a few basic modes in order to keep implementation complexity down.

The most common addressing mode for memory instructions is register indirect

mode, where the address is taken straight from a register. Sometimes, an offset can also be added from an additional operand. This mode is shared with indirect branches, though most ISAs only provide those branches in unconditional form as simple jumps. If the offset for branches is an immediate operand, it is often scaled by the alignment requirement of instructions in order to boost the effective offset range. An alignment requirement means that the address must be divisible by the given number.

For most branches, the instruction pointer-relative mode is used, which applies a scaled immediate offset value to the instruction pointer [9, B]. As the instruction pointer always points to the branch at the time of executing that branch, it is essentially self-relative.

Assembler

An ISA usually includes a low-level programming language with statements that directly map to machine instructions, called assembly language. This is not strictly necessary as compilers can be used together with higher-level languages to write programs, but embedded assembly code inside a higher-level language is still common. The reason for this can be performance, but also a lack of support for some ISA features inside the higher level language. Assembly code gets translated to machine code by an assembler.

An assembly language typically allows labels for different locations in the code and in data memory, having the assembler calculate the effective address [6, p. 39]. This enables programmers to use the labels for loads, stores, and jumps instead of manually calculating the target address themselves.

ABI

An Application Binary Interface (ABI) describes the layout of data fields and structures in memory, the function calling convention, the handling of the stack, the format of binary files, and so on [11]. There is no strict definition of an ABI, so this list only summarizes common elements.

The ABI is not always defined by the ISA, so it can partly be defined by the programming language, compiler, or operating system. However, defining it as part of the ISA improves the compatibility of binaries from different languages, compilers, and operating systems.

2.2 CISC/RISC

The terms of Complex Instruction Set Computer (CISC) and RISC were originally coined by David A. Patterson in the paper ”The case for the Reduced Instruction Set Computer” in 1980 [12]. The paper does not provide a definition for each, but contrasts them against each other as paradigms for computer architectures. CISC is described as a paradigm that favors a large set of complex instructions, whereas RISC goes the opposite way and advocates for a small set of simple instructions.

CISC ISAs are often implemented using microcode. Microcode is code used directly by the CPU, translating ISA instructions into control signals that the rest of the hardware can understand. Those micro programs add an additional layer of complexity, but they enable powerful instructions that perform a lot of work, for example, the ”PCMPISTRI” instruction for string searching in the x86 ISA [13, PCMPISTRI].

On the other hand, RISC ISAs tend to do only one simple thing per instruction, making those easy to implement and fast to execute. Since accessing memory takes longer than only accessing registers, load and store operations typically receive dedicated instructions, while regular ones operate on registers only. The instructions are also simple enough that microcode is unnecessary, avoiding its complexity and the latency of microcode interpretation [14]. This also means that on average, the same amount of work requires more instructions, but that does not seem to affect the performance of modern machines negatively [15].

Although they are opposing paradigms, they have been combined in various instances. Processors implementing a CISC ISA eventually began translating instructions into simpler, RISC-like instructions called micro-operations and feeding them into what was essentially a RISC processor. A very popular example of this is the Intel P6 architecture [16]. On the other end, some RISC implementations began to combine multiple instructions into a single one that the processor could process more efficiently. This process is called Macro-op fusion [17]. Finally, some ISAs that were traditionally RISC adopted some more complex features over time, such as ARM adding memory addressing modes that also modify the used index register [6, p. 33].

2.3 SIMD

In Flynn's taxonomy [18], Single Instruction, Multiple Data (SIMD) is a class of computers which apply the same operation on multiple data elements at a time. Compared to Single Instruction, Single Data (SISD) processors, which only perform the operation on one element, they can provide higher performance when operating on vectors, which are one-dimensional arrays of data elements. Both SIMD and Multiple Instructions, Multiple Data (MIMD) can provide higher performance than SISD computers, but a SIMD processor requires less complexity than the MIMD approach of putting multiple processors in the same system. Both can be combined into a system with multiple processor cores that are also SIMD-capable.

This class is the chosen method of vector processing for all relevant CPUs and ISAs mentioned in this thesis. As with chapter 2.1, the rest of this chapter will only take recent general purpose processors into account.

Requirements

SIMD implementations typically require explicit support and programming. They have separate ISA instructions for loading, storing and operating on vectors, separate registers and separate Arithmetic and Logic Units (ALUs), though those can also be shared with scalar instructions. The width of registers and ALUs usually match, but there are exceptions if an ISA requires a certain width, but the implementation does not invest the needed space and power for a full data path of that size [19].

To take advantage of existing SIMD capabilities, a program must be compiled to use the corresponding instructions. The easiest way to ensure this is by using an explicitly parallel language like OpenCL [20] or a program library for data-parallel computation. Modern compilers can vectorise regular program code to an extent, but it is easy to write code that cannot be vectorised by it.

Most ISAs provide additional addressing modes to fetch elements from multiple locations into a single vector. This enables SIMD processing with non-ideally structured data, but it sacrifices a big part of the performance gain to a long load and/or store procedure, accessing multiple cache lines in most cases. For optimal performance, all elements of a potential vector should be grouped into a continuous block of memory.

Application

Because vector registers are most likely smaller than the vectors a program wants to process, processing them still happens inside a loop, but instead of iterating over every single element, each iteration will process a batch of elements equal to the size of the vector register.

In order to fully use the register size and data path width, vector registers are usually packed, so the smaller the elements, the more fit inside a single register. For example, a vector register with a size of 128 bit can hold up to 2 elements of 64 bit each, or up to 16 elements of 8 bit each [21, p. 18]. Because all elements in a register are usually operated on simultaneously, smaller elements result in more elements per register and therefore higher performance.

Fixed vector size

Some SIMD implementations in an ISA define a specific, fixed size for the vector registers, for example, the Intel extensions from MMX to AVX-512 [8, ch. 10-15] or ARM NEON [22].

Fixed-size SIMD functionality is simpler to implement, but has numerous disadvantages. Most importantly, new processors cannot provide support for larger vectors without adding new instructions to the ISA. Old programs cannot take advantage of such new instructions until recompiled. Conversely, programs using those new instruction cannot run on old processors anymore, or they have to provide multiple code versions and a method to select the appropriate one.

The second big disadvantage comes into play if the size of a vector in memory is not a multiple of the supported vector size. In this case, iterating over it will yield an incompatible amount of elements in the last iteration, which require special handling code. This code can often get larger than the code inside the iterating loop. If element predication is supported, it can mitigate this problem by masking off the excess element space in the vector register.

Variable vector size

Some ISAs provide a way to restrict the size of a vector register (and any vector operation by extension) temporarily to any number of elements lower or equal the maximum supported one. Examples are ARM SVE(2) [23] and the RISC-V V extension [21].

Implementations with a variable size usually have additional architectural state, for example, to save the current vector length. In return, they do not fragment the

software support like implementations using a fixed vector size do. Any processor implementing the ISA can process programs using it, while processors with larger vectors automatically gain more performance if the vectors inside the program are large enough. The implementations also do not need any special setup for program vectors of arbitrary size.

Chapter 3

ForwardCom

As ForwardCom is the ISA that is examined in this thesis, this chapter delivers a detailed description of its properties, as well as some comparisons to RISC-V as the currently dominating free ISA. A higher level overview of the differences between ForwardCom, RISC-V, OpenRISC, x86 and ARM is available at https://www.forwardcom.info/comparison.php.

ForwardCom is an open and free experimental ISA for high performance processors. The initial proposal was made by Agner Fog in 2015, but the goal is to develop the ISA through a public, collaborative process [24]. This effort is ongoing, with multiple iterations of the specification being released since 2016, the latest being version 1.12, which released in 2023. The project hosts its website at https://forwardcom.info. At the time of writing, there are several tools like an assembler, disassembler, linker, emulator, debugger [25], partial support for a CPU simulator, as well as a partial hardware implementation that can run on a Field Programmable Gate Array.

Besides the core ISA, ForwardCom also aims to standardize parts of the surrounding software ecosystem for improved compatibility between compilers, binary tools, programming languages and operating systems, as well as better efficiency.

3.1 Vector support

The ISA has first-class support of vector instructions and registers. This support is deeply integrated into the fundamental design, which is discussed further in chapter 3.3.2.

Each ForwardCom vector register has a variable length by means of saving the

length of the value it is holding [4, ch. 2.4]. This enables ForwardCom programs to use algorithms that are implemented independently of the vector length of a specific hardware implementation. This enables the same code to run on processors of differing vector length support. Forgoing the need to change the program to take advantage of a newly released supported vector length is the main reason for the name ”ForwardCom”.

Like ForwardCom, RISC-V also supports variable-length vector support, though the extension containing them is still under review [26]. The extension contains addressing modes with stride and index vectors, as well as arithmetic reduction instructions and trapping on partially executed vector instructions. ForwardCom deliberately does not support these features. RISC-V also uses a global configuration of vector length, while ForwardCom saves the length per vector register.

3.2 Register set

The register set of ForwardCom contains 32 general-purpose-registers of 32 or 64 bit each, 32 vector registers of implementation-defined length and a few special registers. Floating-point values only use the vector registers, they have no dedicated register set [4, ch. 2.3].

The special registers contain a set of performance counters, a set of capability registers and a few individual ones:

- The instruction pointer, containing the address of the current instruction.
- The data section pointer, containing the address of a reference point for position-independent data access.
- The thread data pointer, containing the address of a reference point for position-independent access to thread-local data.
- The numeric control register, used when instructions specify no mask register. It contains bits to control floating-point number behavior, such as rounding.

There is no register for ALU flags and no registers for handling traps, except for a few performance counters tracking them.

For RISC-V, the available registers depend on the chosen ISA extensions. There are extensions for floating-point numbers that use a separate floating-point register set [10, ch. 11.1]. The ISA uses registers called Control and Status Registers (CSRs) for communicating capabilities, behavior and statistics.

3.3 Encoding

ForwardCom has various different instruction formats, but all formats are encoded in one of five templates that specify what type of field is in which part of the instruction. Some of these templates can also be extended with additional immediate value fields. The locations of all fields, except immediate values, are the same in all templates. The only change is if they exist.

3.3.1 Instruction length

ForwardCom currently supports instruction lengths of one, two and three instruction words of 32 bits each. The length is encoded in the first two bits of any instruction. There was support for a smaller instruction size of 16 bits in previous versions, but its implementation as pairs of two instructions in one 32 bit word proved too inflexible, so it was dropped in version 1.10 [27]. There also exists unused encoding space inside three word instructions identified by an additional bit. That space is reserved for instructions of 4 or more words.

In contrast, RISC-V potentially allows any multiple of 16 bits for instruction length, which is encoded by a number of bits increasing with length [10, p. 8]. At the time of writing, only the 16 bit and 32 bit sizes are in use by the base specification or any standard extension.

3.3.2 Instruction formats

ForwardCom has relatively few instructions compared to ISAs like x86, but the most common of them can be used in a completely orthogonal fashion. This means that the same instruction can use different data types as well as different formats providing different operand types [4, ch. 2.1].

The format defines whether the instruction uses general-purpose registers or vector registers for the register operands. For general-purpose formats, the data type can be an integer number of 8, 16, 32, or 64 bits. For vector formats, the data type additionally can be an integer number of 128 bits or a floating-point number of single, double or quadruple precision. Half precision floating-point numbers are supported in numerous cases, but typically use separate instructions.

A format can have up to three input operands. All three can be register operands, while the destination register can still be separate as up to four registers can be encoded. Up to one input can be an immediate value of up to 64 bits. Floating-point immediate values are supported, integer values can also be shifted. Up to one other operand can be a memory operand, with one of many addressing modes.

It is important to note that ForwardCom instructions cannot have more than one data output to simplify dependency tracking.

The format can provide a mask field, see chapter 3.3.4. Formats using multiple instruction words can also provide additional option bits which can alter the behavior of the instruction operation.

RISC-V puts more focus on simplicity, only containing a few formats that are selected by the instruction, though the various extensions can provide additional ones. The base formats can provide up to two input operands, up to one of them being an immediate value [10, ch. 2.3]. Memory operands are handled by dedicated load and store instructions. Some RISC-V instructions have two outputs.

3.3.3 Addressing modes

The memory operands calculate their address with up to three operands. They always use a base register and can have an index and/or an offset.

The index can be scaled by the size of the instruction data type. For vector loops, there is a mode using a negative index.

Vector loops can apply a negative index to the end of an array instead of a positive one to the start. Instead of counting the processed memory, it will then count the amount of memory that still needs to be processed. Because vector instructions use the maximum vector length when a length greater than supported is supplied, this index is also used as the length of such an operation. In loops, this mode enables the automatic use of the fitting vector size in the last iteration, leaving no leftover that would need special handling as described in chapter 2.3.

The offset is always supplied by an immediate value. For immediate fields of 8 bits, the offset is scaled by the size of instruction data type. It is not scaled for larger fields of 16 or more bits. Instead of an offset, the format can also define an immediate value as a limit for the index value, acting as a safety feature [4, ch. 2.8].

RISC-V again chooses a simpler approach and only supports a base register and a 12-bit offset. On the other hand, the vector extension supports some additional, more complex modes [21, ch. 7.2] that are already described in chapter 3.1.

3.3.4 Predication

Except for jumps and a few special other instructions, all instructions can be masked. This is part of the orthogonal encoding, so instructions working on general-purpose registers can be masked, as well. Instead of additional architectural state

and instructions for dedicated mask registers, the first seven registers of either register set can be used.

Additionally, a fallback register can be specified by the register operands. Masked off elements for vectors or the entire instruction for general-purpose registers will instead source their output value from that fallback register. If there are not enough operands to specify a separate fallback register, the first input operand can be used instead [4, ch. 3.2].

RISC-V supports masking in their vector extension, but not in the base instruction set [10, p. 23]. Masks are always supplied by the vector register v0. The ISA does not support a separate fallback register, but a CSR that defines if masked off elements in a vector should keep the previous value of the destination register, or are allowed but not guaranteed to get all bits set to logical 1.

3.4 Instructions

This chapter contains several omissions for the sake of brevity. It does not mention Instructions which can reasonably be assumed to exist, such as basic arithmetical or logical operations.

Many advanced instructions are fairly general and use instruction bits to define the exact behavior. Because instruction bits are not included in instruction formats using a single instruction word, these instructions define the most common case for all option bits being zero, which is the implied value when they do not exist in the format [4, ch. 3.2].

Most operations using additional information, such as overflow bits, are implemented as element pairs in vector registers, one element for the value and another for the additional information. This is done in order to comply the restriction to a single output for instructions.

3.4.1 Arithmetic instructions

Arithmetic operations have saturating variants, as well as variants that generate and use overflow and carry values. Instructions depending on the operand order have additional versions switching the order. This enables using the same register operand for both the first input and the destination in more cases. There is also a square root and a power-of-two function.

ForwardCom has two fused multiply-add instructions. One of them changes the operand order, and both can change the sign of the product and/or the addend

using option bits. There also is an addition instruction with three operands, which can invert the sign for each.

Generating boolean values by comparing two numeric values is done using a single compare instruction. The type of comparison is defined by option bits.

3.4.2 Logic instructions

ForwardCom has various instructions for all kinds of bit manipulations. Single bits can be set, cleared, or toggled. A single bit can be tested, yielding a boolean value, with option bits using the fallback value as another operand for a boolean combination [4, ch. 5.5]. Groups of bits can be tested, moved or replaced. Set bits can be counted, the index of the highest or lowest set bit can be determined. The bits or bytes of values can be reversed. The category of floating-point values can be checked, for example, for finding invalid number values.

3.4.3 Vector instructions

The ISA provides a multitude of instructions for dealing with vectors. Elements can be extracted, inserted, shifted, replaced and repeated. Vectors can be expanded, truncated, and concatenated. Boolean vectors can be compressed to bit vectors and the other way around.

ForwardCom offers relatively few instructions for combining vector elements horizontally, meaning different elements of the same vector. In order to limit the wiring complexity of implementations, ForwardCom provides a permute instruction and boolean reduction, but none for arithmetic reduction [4, ch. 14.3].

3.4.4 Branch instructions

Branch instructions have their own set of instruction formats, using the same templates as mentioned in chapter 3.3. Branches that are relative to the instruction pointer can have an offset of up to 32 bits, which gets multiplied by the instruction word size to provide a range of $\pm$ 8 gigabytes [4, ch. 3.4].

Unconditional jumps and calls

Unconditional jumps and function calls share the same formats. Calls push the return address on the call stack, which is separate from the data stack referenced by the regular stack pointer. The call stack cannot be referenced directly, which makes it easy to implement in hardware since it only performs fundamental stack operations for dedicated instructions.

Besides relative jumps and calls, there is an absolute version with the address in a general-purpose register, as well as two indirect ones. One uses a base pointer to load an address from memory. The other version has a memory operand of a base pointer and either a scaled index register or an offset without scale, which is added to another register value to form the effective target address. This version can be used for function tables or jump tables.

RISC-V takes a simpler, slightly different approach. It has one jump instruction relative to the instruction pointer, and one instruction with a register address and an offset. There are no calls, but both jump instructions can save the address of the instruction after themselves into a general-purpose register, functioning as a return address [10, ch. 2.5].

Conditional branches

Branch instructions, like in RISC-V [10, p. 16], compare two values and jump depending on the result. ForwardCom provides instructions that additionally output a register value. Addition, subtraction and logical operations can jump depending on the result, while the incrementing and decrementing instructions can do a comparison with a second value. This enables loops with only a single instruction in the loop body being concerned with the control flow. Branch instruction doing addition or subtraction can alternatively jump on overflow, carry or borrow.

Some branch instructions can also use the vector registers, though only their first element is used. Logic branch instructions treat the element as an integer, while the compare branch instructions use floating-point numbers.

3.4.5 Memory instructions

As any instruction can have a memory operand, there is no dedicated load instruction. In addition to the regular store instruction, a variant stores a single element from any position of a vector register. As an exception to the other instructions that do not need additional control circuitry, ForwardCom also features push and pop instructions that push and pop a range of registers in a single instruction [4, ch. 5.2]. These operations will need to be sequenced in most implementations.

3.5 Security features

Security is an integral part of the ForwardCom ISA design. Many included features either improve the handling of abnormal operation, or reduce the attack vector by restricting access to different types of data in a variety of contexts:

- The separate call stack described in chapter 3.4.4 effectively prevents manipulation of any return address.
- The ISA provides addressing modes and instructions that make many validity checks nearly free, described in chapter 3.3.3 and 3.4.4, respectively.
- Exceptional operation never produces undefined behavior, there always is a set of permissible responses.
- Function pointers and jump tables are placed in a read-only section of memory.
- A program does not have read or write access to its code memory, and no execute access to its data memory.
- Threads can contain local data that is not accessible to other threads of the same process.
- The memory management approach and its security implications is described in chapter 3.6.
- System calls also define the memory that is shared with the system function.
- Drivers can be restrained in their access to memory and I/O ports.
- Preference for static linking prevents library updates from changing program behavior silently.

3.6 Memory model

ForwardCom programs are generally position independent. Instructions and constants are accessed relative to the instruction pointer, while mutable data is accessed relative to the data section pointer or the thread data pointer. This renders virtual memory unnecessary for compatibility reasons.

Memory management is not done using pages with fixed sizes like in x86 [28, ch. 4], ARM and RISC-V [29, ch. 4.3-4.6], but with few segments of variable length. Every segment defines permissions for reading, writing and executing the contained data. Translation to another physical address is implemented via an offset, but this is optional per segment. The big advantage of this implementation is that the description of the entire address space used by a thread potentially fits inside the hardware. This enables translation and permission checks for any memory access with low latency, improving performance compared to multi-level table look-ups of prevalent virtual memory implementations, and avoiding spec-

ulatively accessing memory without permission, which has lead to side-channel attacks in the past [30].

The limited number of memory segments per thread puts pressure on the ecosystem to limit the fragmentation of data. ForwardCom implements several features to address this.

A significant one is the handling of libraries, which are statically linked, re-linked as part of the installation process, or loaded at runtime. In the former two cases, the library code is effectively integrated into the main program, producing no additional memory segments [4, ch. 12.8]. This also has a side effect of more efficient communication between the main program code and the library code.

Another such feature is the calculation of the required stack size. Object files store the needed stack size of each function, so the exact needed stack size can be calculated as long as the program does not use recursive functions. If it does, a reasonable estimate might still get reserved at load time. This can prevent stack overflow.

Chapter 4

Experiment setup

This chapter describes the content of the experiment, as well as the methodology. The experiment aims to take several parameters of the ForwardCom ISA and measure their impact on microprocessor performance.

The basis for the ForwardCom ISA specification used in this experiment is the ForwardCom manual version 1.12.

The experiment is performed using the General Purpose Core Architecture Simulator (GPCAS) version 0.2.1 [31]. There currently is no hardware capable of running full ForwardCom programs. As of the time of writing, GPCAS is the only ForwardCom-compatible CPU simulator.

4.1 General Purpose Core Architecture Simulator

GPCAS is a deterministic, ForwardCom-compatible [31, gpcas_forwardcom] simulator for CPU models. It simulates the execution of an executable file on a CPU model with its configuration supplied via a JavaScript Object Notation (JSON) file. The structure of the configuration can be found at https://docs.rs/gpcas_cpu_model/latest/gpcas_cpu_model/struct.CpuModel.html.

The simulation happens on a pipeline level. This means that the logic of each component inside a CPU is modeled, but not the actual logic gates or transistors. The simulation results are written to another JSON file. That report file contains a hash of both the used program file and the used CPU model file, the amount of instructions needed to emulate the program to completion, and the amount of clock cycles it took the model to run the program to completion [31, gpcas_simulator/src/simulator/report.rs].

The rest of this chapter will shortly describe the function of each simulated component.

4.1.1 Model of the CPU core

Starting at the instruction fetch unit, it will request the next instructions from memory, as well as perform branch prediction [31, gpcas_simulator/src/simulator /model/components/front_end/fetch.rs]. If a taken branch is predicted, the fetch unit will cancel all requests made since the occurrence of the predicted branch. A fetch buffer will receive the requested instruction memory and buffer it until the decoder uses it.

The decoder translates instruction memory into individual instructions and registers them in the reorder buffer, the register file and the load/store unit, if applicable. It then sends the instructions to the dispatcher. The amount of instructions decoded per clock cycle, as well as the latency, is configurable. The decoder can also be configured to perform unconditional, direct branches it finds. Direct branches are branches that are relative to the instruction pointer and do not use register or memory operands. Doing so will empty the fetch buffer and cancel all open requests from the fetch unit.

The register file contains the register state, which can be equal to the ISA defined registers, but can also contain more registers for each set to enable register renaming [31, gpcas_simulator/src/simulator/model/componnts/reg_file.rs]. Register renaming maps the registers defined by the ISA to a larger set of physical registers, enabling instructions to write to a register before all preceding instructions are done reading from it.

The dispatcher sends incoming instructions to connected execution pipelines or schedulers that are not full and capable of handling that instruction type. It will always send instructions to schedulers, but stall if it tries to send instructions to execution ports while they are missing some input operands. Its latency is configurable. If no scheduler or execution pipeline can handle ALU instructions with a memory operand, they will be split into the ALU and the memory operation. The dispatcher will also perform register renaming for schedulers, as well as for execution pipelines if those are configured to support it.

Schedulers are optional components that enable out-or-order execution. For each connected execution port, it will send the oldest instruction it holds that has all operands ready [31, gpcas_simulator/src/simulator/model/componnts/scheduler.rs].

Execution pipelines are a general term for a combination of an ALU and/or an Address Generation Unit (AGU) with a connection to the load/store unit. Each

execution pipeline can process up to one instruction per clock cycle. They have a configurable delay before processing arriving instructions, simulating pipeline stages for register reading. ALUs have a configurable latency depending on the operation type. AGUs always have a single clock cycle of latency. Pipelines can be configured in three modes, AGU only, memory only, or combined. The ALU only mode contains a single ALU. In memory only mode, it contains an AGU and a memory access stage of a single clock cycle. If the pipeline supports combined mode, it contains an AGU, followed by a memory stage, followed by an ALU [31, gpcas_simulator/src/simulator/model/componnts/execution_pipe.rs]. Execution pipelines forward their results to dependent instructions, so a scheduler or dispatcher can dispatch them in the same clock cycle.

The load/store unit manages load and store operations and tracks their dependencies. Each clock cycle, it performs a configurable number of load or store requests to memory. It always sends performs the oldest operation that is ready. The results of performed load operations for forwarded to instructions depending on it, so they can proceed in the same clock cycle.

The reorder buffer tracks instructions that are in progress and completes them in program order if they performed their operation [31, gpcas_simulator/src/simulator /model/componnts/reorder_buffer.rs]. Register results are written back to the register file. Store instructions are marked as completed, so the data can be committed to memory. The reorder buffer performs a redirection of the program flow in case of a branch that was predicted incorrectly or not at all. If it performs a redirection, the core is flushed, meaning all instructions in progress are canceled.

4.1.2 Model of the memory hierarchy

The modeled CPU core has two connections to memory: one from the instruction fetch unit for instructions, and one from the load/store unit for memory operations from instructions. Both can connect to either a cache or the memory controller. The load/store unit can be configured have multiple ports, enabling it to perform multiple operations in parallel.

The memory controller serves requests for memory data. It has a configurable, uniform latency and is fully pipelined, meaning it can process a new request in every clock cycle. It can also serve all connections in parallel. Its bandwidth per connection can be configured, which effectively limits throughput and generates some additional latency [31, gpcas_simulator/src/simulator/model/connections/memory.rs].

Caches can cache memory data, enabling lower latency and higher bandwidth if data inside the cache is requested. This component supports configuration for

several parameters, including the total cache size, the size of cache lines, the associativity, the latency and the bandwidth. The latency is uniform and applied as a request delay, so a response is sent in a single clock cycle. The cache is also fully pipelined and can be configured to either handle all of its ports in parallel, or if only one request can be processed in a clock cycle. In the latter case, the port with the lowest index is always given priority, except if it sends a prefetch request. If new data is loaded into the cache while the corresponding sets of lines already contain data, a line for eviction is selected using a pseudo-least-recently-used algorithm. Finally, caches can be configured to include a prefetcher. If the cache port towards the next memory layer is idle, the prefetcher will select an ongoing request and generate requests for a subsequent cache line until the line fill buffer is full [31, gpcas_simulator/src/simulator/model/componnts/cache.rs].

4.2 Methodology

To measure the effect of the chosen ISA parameters on performance, a general performance measurement methodology has to be defined first.

Processor performance can generally be defined as the reciprocal value of the execution time for a task [32, p. 39], which is straightforward to measure. The simulator provides the execution time of the given program in clock cycles. As the implementation complexity and potential clock speed impact of the tested parameters is explicitly not examined in this experiment, and the comparison is strictly relative, clock speeds can safely be assumed to be constant and therefore ignored. Therefore, the number of clock cycles needed to execute a task can be used as a metric for execution time.

To vary a ForwardCom parameter, the benchmark programs are transformed into a new version that explicitly uses the chosen variation of that parameter. For most of them, this is a simple matter of presence or absence of an optimization using their properties. To isolate the impact of a single parameter on performance, each variant of the parameter is tested in combination with each variant of the other parameters. This yields the performance impact of each parameter in relation to the others and also allows to observe a potential synergy effect between parameter combinations.

By its nature, this experiment cannot generate results that are representative of real-world programs or real-world processors, and real-world performance by extension. As ForwardCom is missing a compiler, there are no real programs written for it yet. The usage of GPCAS and the simulation of virtual CPU models instead of real microprocessors means embracing its limitations, especially in terms of

branch prediction. Therefore, any results can only be seen as rough indicators of real-world performance instead of precise values. To limit the effect of properties specific to the used benchmark program or CPU model on the result, two of each are tested.

Each combination of parameters is tested for each benchmark and each CPU model, if applicable. The benchmark described in chapter 4.3.1 doesn't use ALU instructions with a memory operand, so this parameter is not tested in this case. The "Serial" CPU model described in chapter 4.5.1 doesn't support long static vector lengths, so those cases are also not tested.

4.3 Benchmark programs

As the benchmarks need to be in the ForwardCom assembly language and there are none yet, reimplementing other benchmarks in ForwardCom assembly is part of the experiment. For this reason, they need to be small and not use large external library functions. Most benchmarks on the "The Computer Language Benchmarks Game" [33] website perfectly match these requirements. Although the site uses them to compare different programming languages with each other, the focus of this test is similar: Changing the ISA parameters also yields a different interface to the processor. For these reasons, the benchmarks are adapted from this website.

To fit multiple benchmarks in the time frame of the thesis and because implementation complexity is not strictly an indicator of program behavior, the most simple arithmetic problems are chosen.

The programs were implemented naively, then deoptimized to remove all features defined by the parameters. Those versions were then optimized without using the features to provide a valid baseline version. Starting from these versions, usage of features is selectively added to create a version for each parameter combination.

4.3.1 Mandelbrot

The "Mandelbrot" benchmark generates a visualization of the Mandelbrot set as a portable bitmap [34] file.

Each pixel of the bitmap corresponds to a point in the complex plane. For each point, the program performs 50 iterations of the quadratic recurrence equation. If the point value after 50 iterations is not greater than 4, the pixel is colored black, as seen in figure 4.1.

The written assembly implementation of this benchmark is very simple. It does

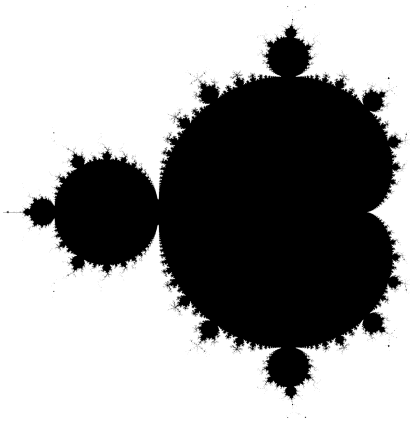

Figure 4.1: Output of the ”Mandelbrot” benchmark

limited memory operations, and the implementation makes no use of the parameter in chapter 4.4.3, which therefore is not tested. The calculation of each pixel is independent from each other, which allows for every pixel to be calculated in parallel, although the chosen implementation is limited to a single horizontal line at a time. Consequently, performance on an otherwise identical processor should scale almost linearly with its supported vector size. In addition, an implementation with known vector size might omit several checks, so using variable length vectors might cause slightly lower performance. This is the case for a vector size of 512 bits especially, since they operate on eight 64 bit floating-point numbers simultaneously, which exactly result in a single byte of the bitmap. Where smaller vector sizes produce bit fields that need to be merged into whole bytes, that entire code can be omitted for sizes of 512 bits and up.

For the experiment, a bitmap size of 1920 times 1920 pixels is chosen. The exact size is arbitrary, but the order of magnitude is a compromise between the suggested resolution of 16000 times 16000 pixels for performance testing from the website, and a reasonable simulation time. A too small size would introduce more noise in the result, while the suggested size would take numerous hours to simulate all parameter combinations on the hardware available for this thesis. The runtime in this configuration is roughly comparable to the ”N-body” benchmark.

4.3.2 N-body

The ”N-body” benchmark simulates the movement of planets while being influenced by each other’s gravity. The used benchmark is not a full reimplementation, it is adapted from the ForwardCom assembly test program included in GPCAS instead. It uses five bodies, as does the original version on the website.

Each simulation step, each body causes the other bodies to gravitate towards it. The strength of that force is determined by the body's mass, as well as the distance between both. Before and after simulation, the program calculates the total energy of the system for validation. The energy of the whole simulated system should theoretically stay constant while simulating. In reality, the limited precision of floating-point number implementations in processors will accumulate an error that needs to be considered when checking for correctness.

The algorithm implementation makes frequent use of all tested parameters. However, it only exhibits limited data parallelism, mainly constrained by the small amount of bodies in this benchmark. When calculating the gravitational force between bodies, the maximum number of forces that can efficiently be calculated at a time is the number of bodies minus one, so four, in this case. A vector size of 256 bits suffices to extract maximum parallelism, using 64 bit floating-point values. This case also appears only once per simulation step, so the scaling above 128 bit vectors is also quite limited.

The program simulates one million simulation cycles for this experiment. This exact value is also arbitrary, but the order of magnitude is still a compromise between the suggested amount of cycles and a reasonable simulation duration, just as with the "Mandelbrot" benchmark. The website suggests the use of 50 million cycles for performance testing.

4.4 Tested parameters

Of all ForwardCom parameters, the ones chosen for testing are those whose impact can be examined using the selected benchmark programs in the used execution environment. The selected parameters are features that mainly enable different kinds of instructions and instruction formats. When used, these features enable an implementation that does the same amount of work in fewer instructions inside an isolated block of code, making them easy to measure in isolation. Variable length vectors (see chapter 4.4.4) are a bit of an exception since they only improve performance indirectly by preventing a lowest-common-denominator implementation. Such an implementation would suffer from low hardware utilization on any CPU model that features larger vectors.

There also are several parameters that are not tested. For example, the ABI of ForwardCom can theoretically provide efficiency gains at the module level. However, modules only come into play naturally when a program reaches a certain size and complexity. Such programs would be easy to write in a high level language, but translating them in executable code requires a compiler, which does not exist for

ForwardCom at the time of this thesis. Consequently, this parameter is not chosen for testing.

ForwardCom also has extensive masking support, but implementing the benchmark programs revealed that none of them could efficiently make use of masking for the chosen algorithm.

Other parameters impact efficiency at the system level, but these also need tooling that does not exist yet in order to be tested. For example, the virtual memory implementation of ForwardCom cannot be tested without an operating system managing the memory.

Some parameters affect other properties, such as security, but as this experiment only tests for performance, they are beyond its scope.

4.4.1 Multi-word formats

This parameter can be either allowed or disallowed.

Having instructions of different lengths requires the processor to determine the length through decoding in order to find the beginning of the next instruction. This process adds complexity to the hardware implementation, and its serial nature can limit the amount of instructions possible to be decoded in parallel. In exchange for this limitation, it allows to encode a multitude of types of additional information into the instruction:

- Additional opcode bits.
- A fourth register operand, so instructions with three inputs can still choose a separate output register.
- Large immediate values, which would otherwise need to be loaded from memory separately.
- Option bits, which can make some instructions much more flexible. For example, there is only one compare instruction because the comparison mode is encoded in option bits.
- Additional formats providing broadcast immediate values, shifted immediate values, memory operands with both scaled index and offset, and so on.

Because the tested programs do not use all available registers of the register set, large constants are loaded from memory into a register outside of the main loop if the parameter is disallowed. This represents a common mitigation if this parameter is not available.

Since ForwardCom is designed with multi-word formats in mind, some instructions strictly require an instruction length of over 32 bits and cannot be avoided.

address

This instruction returns the absolute address of an item in static memory. The item can be data, but also an instruction.

The instruction takes a 32 bit immediate value as an offset to identify items relative to the data pointer or instruction pointer base address. For this reason, the instruction is encoded using 64 bits.

Popular instruction sets such as ARM [9, p. 36] or RISC-V [10, p. 28] use an instruction for the upper bits and another one for the lower bits of the target address. Because the chain of two dependent instructions is arguably more important than the instruction size, in this case, that constellation will be emulated by adding a dependent ”add” instruction of zero after each ”address” instruction in the respective tests.

compare

The ”compare” instruction uses option bits for its operation mode, which are only present in some instruction formats that use two or more words. The only exception is the default mode of equality comparison. Since this could easily be encoded in a smaller one when designed for it, no compensation for this instruction is performed.

4.4.2 ALU-and-branch instructions

This parameter can be either allowed or disallowed. Disallowing this parameter means only using compare-and-branch instructions, while doing any ALU operation in a separate one.

4.4.3 Instructions with memory operand

This parameter can be either allowed or disallowed.

As written in chapter 3.3.2, ForwardCom allows up to one operand of most instructions to be sourced from memory. The orthogonal encoding of ForwardCom allows all addressing modes to be used with most instructions, as well. This should provide ample opportunity to combine loading a value from memory with the first operation using it, potentially improving code density and simplifying dependency tracking.

The parameter should have a larger impact when multi-word instructions are also allowed. Because the operands for the memory address calculation have to be encoded in the instruction, that code space cannot be used to provide other inputs. As single-word instructions can only provide up to three operands, that can quickly leave the instruction with just one or even zero available additional input operands. Having more code space in multi-word instructions relaxes this constraint.

4.4.4 Variable vector length

This parameter is tested in three variations: Variable vector length, a fixed length of the lowest common denominator of 128 bits and a fixed length equal to the highest one supported among the CPU models, or the highest one that still yields some performance benefit, whichever is lower.

Not using a variable vector length means falling back to the lowest common denominator of all targeted processors, which the ForwardCom manual suggests to be 128 bits. Alternatively, multiple versions of the code could be delivered, with a dispatching mechanism selecting the variant matching the hardware implementation. Because the potential number of algorithm versions for dispatching is large, only the first variant is tested.

On the negative side, folding a vector of variable length needs a loop and therefore has a higher overhead than a static length. An algorithm also cannot make most assumptions about the length of a vector inside a loop. For these reasons, an algorithm implementation with variable vector length might exhibit slightly lower performance than an implementation with a matching static length. Because ForwardCom is designed with variable vector length support in mind, there is no way to load a vector from memory directly without specifying its length, so a static length could provide some additional encoding efficiency gain that is not captured by this experiment.

4.4.5 Vector loops

This parameter can be either allowed or disallowed. As this feature does not work without variable length vector support, that case will not be tested.

The feature represents the negative addressing mode described in chapter 3.3.3, as well as a branch instruction that subtracts the maximum vector length from an index register. If this parameter is disallowed, programs are forced to use a separate index register in a loop and to update both that register and the remaining length in every iteration.

4.5 CPU models

As ForwardCom aims to be a general-purpose ISA, a variety of optimization points must be considered when designing a potentially realistic CPU model. The limitations of the used simulator must also be considered. The most problematic shortcoming of it is the absence of any form of global history branch prediction. Because of this, real high performance models, such as the SonicBOOM [35] processor, cannot be adapted. To somewhat mitigate the impact of this circumstance, both designed CPU models feature a limited maximum throughput of two instructions per clock cycle. This limits the amount of instructions wasted through mispredictions. They also feature very short pipelines with an unrealistically low branch misprediction penalty of only 6 and 7 clock cycles, as can be seen in table 4.2. This is especially apparent when comparing to real designs such as the mentioned SonicBOOM or AMD's Jaguar CPU core [36], which features a misprediction penalty of 12 and 14 clock cycles, respectively.

Data path	**Latency in clock cycles**			
	Serial	Parallel	SonicBOOM	AMD Jaguar
Predicted branch	1	1	1-3	1, 2, 4
Relative jump	4	4	4	?
Mispredicted branch	7	6	12	14
ALU to use	1	1	?	?
Load to use	4	4	4	4

Figure 4.2: Latency of critical data paths for both CPU models. Lower latency values are better. A latency of one clock cycle enables back-to-back operation using the previous result.

Both models feature a 2048-entry Branch Target Buffer (BTB) as the maximum possible sensible configuration for branch prediction of the simulator. Their fetch bandwidth of 16 bytes per clock cycle is sufficient for maximum throughput except in cases of multiple very long instructions in a row. They feature L1 Caches capable of sustaining either two loads or one load and one store per clock cycle. Both L1 caches also feature a next-line prefetcher.

ALU instructions with a memory operand get split into the ALU and the memory operation at dispatch. They only occupy a single entry in the control unit or Reorder Buffer (ROB), but take up two execution ports.

4.5.1 ”Serial”, optimized for fast execution of scalar code

This model represents a low power client device. The most common workloads in a client environment that are compute-heavy and parallel, picture and video processing, are usually processed by a dedicated graphics processor, so the CPU does not need to handle them effectively. For that reason, this model features a small vector data path of only 128 bits and puts more emphasis on general instruction throughput through its capability to execute instructions out-of-order. To this end, it features register renaming, as well as schedulers, which are shared between the execution ports of each type.

As seen in figure 4.3, the model provides ample execution resources for sustaining two instructions per clock cycle. Exceptions include branches, stores and divisions. The level 2 cache can match the bandwidth of the L1 data cache and therefore supports full throughput, even if the L1 cache misses.

4.5.2 ”Parallel”, optimized for code with high data parallelism

This model represents a low power, throughput-oriented core for processing larger data vectors. It puts less emphasis on scalar execution speed and therefore does not feature instruction reordering capabilities, but can still sustain up to two instructions per clock cycle in an ideal case. The vector data path is much wider at 512 bits, as noted in figure 4.4.

The level two cache cannot sustain even a single full vector read or write per clock cycle, so the model is designed for small working sets running out of the L1 data cache. In general, more emphasis is put on area and power efficiency, which results in caches with lower capacity and associativity. The back end can sustain two instructions per cycle, except for branches, stores, scalar multiplication and divisions for both scalars and vectors.

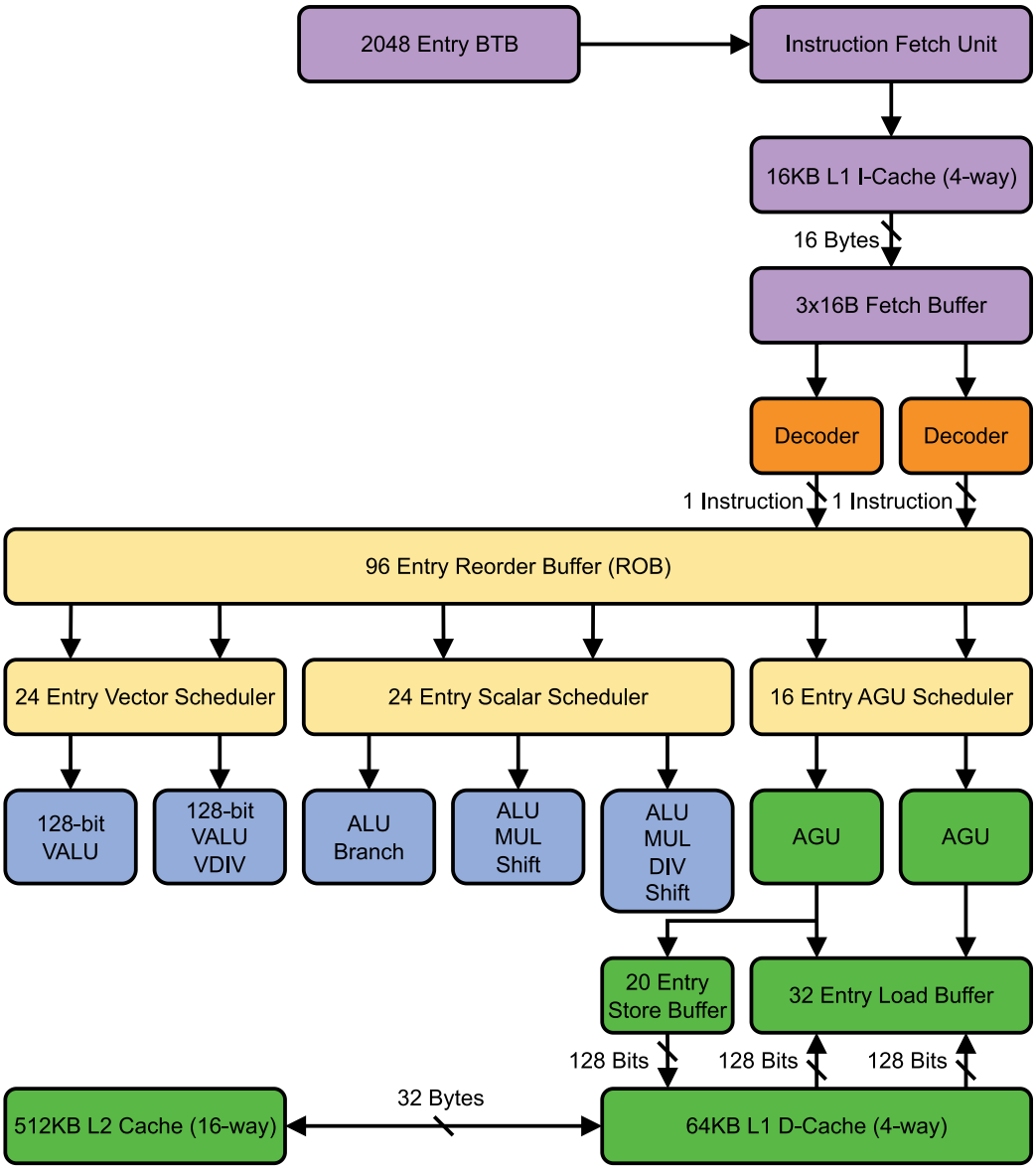

Figure 4.3: Data flow diagram for the ”Serial” CPU model

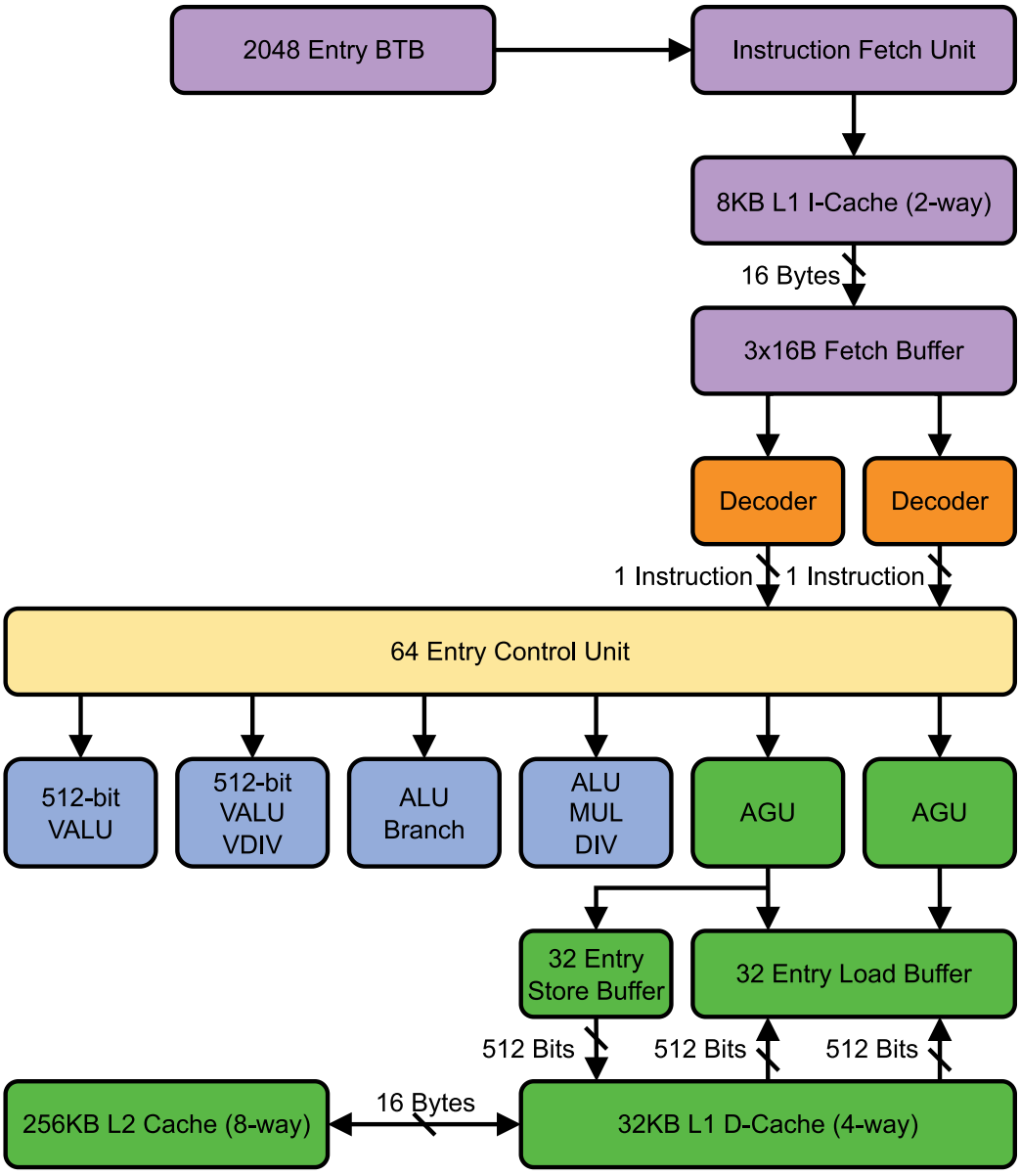

Figure 4.4: Data flow diagram for the ”Parallel” CPU model

Chapter 5

Test results

The amount of emulated instructions, as well as the amount of simulated clock cycles to run the program, have been recorded for each combination of parameters, benchmark programs, and CPU models.

There are several ways to interpret the data, so this chapter will start with a general overview and then proceed to more specific comparisons.

5.1 Combined impact

Choosing the tested parameters to allow all included features yields a substantial improvement over the opposite case as seen in figure 5.1, considering both configurations ran the same algorithms on the same CPU models. As the ”Parallel” model provides longer vector support than the ”Serial” model, the former is compared to two baseline programs: One that is compatible with the ”Serial” model, and one that fully uses the supported vector length.

Independent of this, the ”Parallel” model benefits to a significantly greater extent from allowing the features than the ”Serial” model does. The lower performance baseline of the former, as pictured in figure 5.2, can explain this. The simulator sadly does not provide more insight into the model behavior, but considering the small enough working sets to fit in the L1 data cache, and the main difference of the cores being of instruction reordering, a reasonable guess can be made: allowing the features might remove some dependency chains by combining instructions, which can otherwise easily stall a CPU with no reordering capabilities, lowering performance.

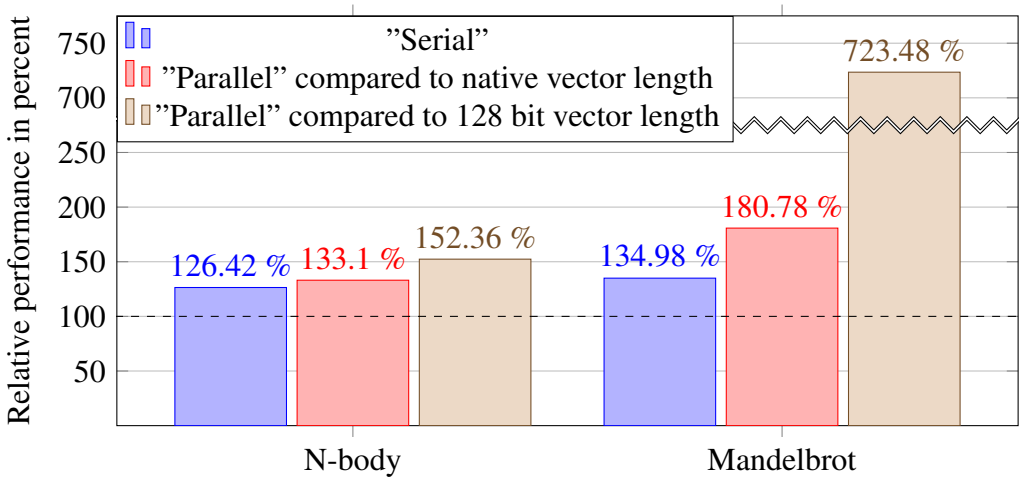

Figure 5.1: Performance of the CPU models when allowing all features, relative to allowing no features. Only the baseline using a 128 bit vector length is compatible with both CPU models.

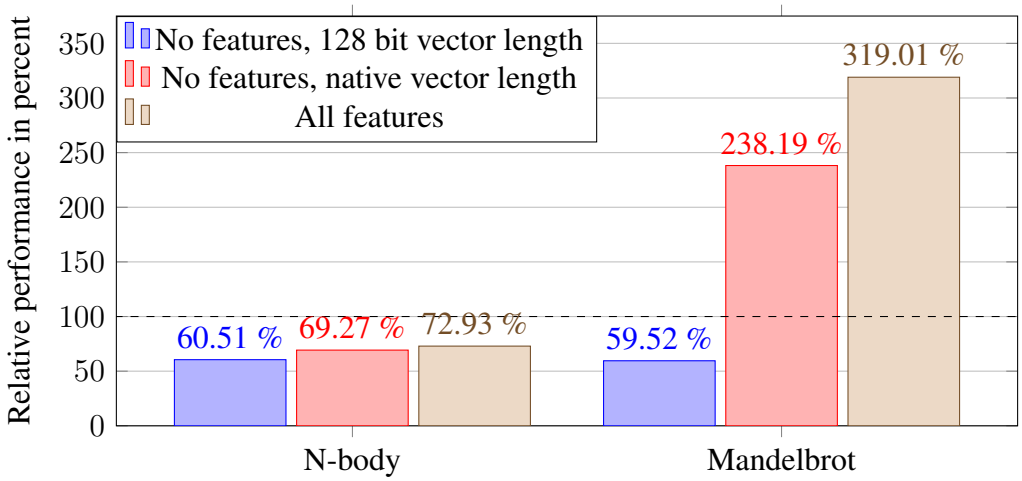

Figure 5.2: Performance of the ”Parallel” model relative to the ”Serial” model.

5.2 Per-feature comparison

ForwardCom provides all features tested per definition, and therefore can deliver the combined impact shown above. But from a perspective of ISA design, it is also interesting to see the impact of individual features, as well as their behavior when combined to find synergies or inefficiencies.

5.2.1 Multi-word formats

Both models on both programs show a reaction to this parameter, which is not always the case for others. That being said, the positive performance impact using the Mandelbrot benchmark is about ten times that on N-body, as seen in figure 5.3. The benefit of the ”Parallel” model also is about twice that of the ”Serial” model.

The ”Serial” model reacted especially positively to the combination of multi-word formats, variable length vector support, and no vector loops. That may not be of relevance because the effect was minuscule, at two to three percent for the N-body benchmark and three to five percent for the Mandelbrot benchmark. Meanwhile, the ”Parallel” model running the Mandelbrot benchmark shows a clear division into two sets. Using ALU-and-branch instructions in combination with multi-word formats delivered a performance improvement of 72 to 75 percent, while not using ALU-and-branch instructions puts the improvement at a lower area of 61 to 63 percent.

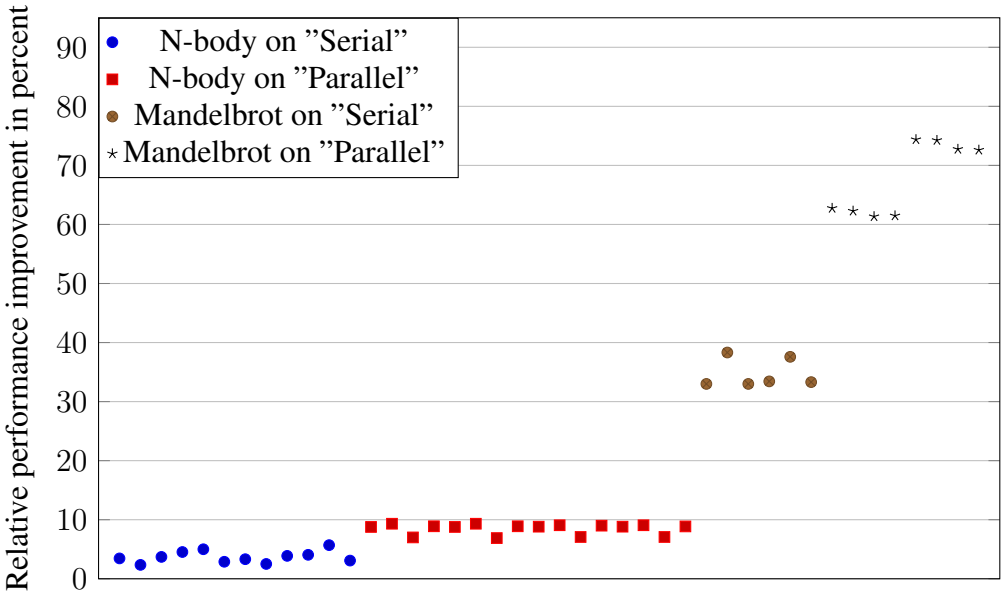

Figure 5.3: Relative performance improvement through the use of multi-word formats.

5.2.2 ALU-and-branch instructions

Excluding a few very slight regressions, both CPU models and programs saw a performance improvement when using ALU-and-branch instructions. However, except for Mandelbrot running on the ”Parallel” model, the improvement always stays below two percent as seen in figure 5.4.

The results for that combination show the same synergy with multi-word formats as seen in chapter 5.2.1. With them, performance improves by 13 to 14 percent. Without them, the improvement stays in a range of 5 to 7 percent.

Knowing the code of the Mandelbrot benchmark, an educated guess for the cause can be made: The inner loop performs lots of iterations with a relatively small loop body. For this reason, both multi-word formats and ALU-and-branch instructions provide a relatively high reduction in the amount of instructions inside the loop body, which compounds when combined.

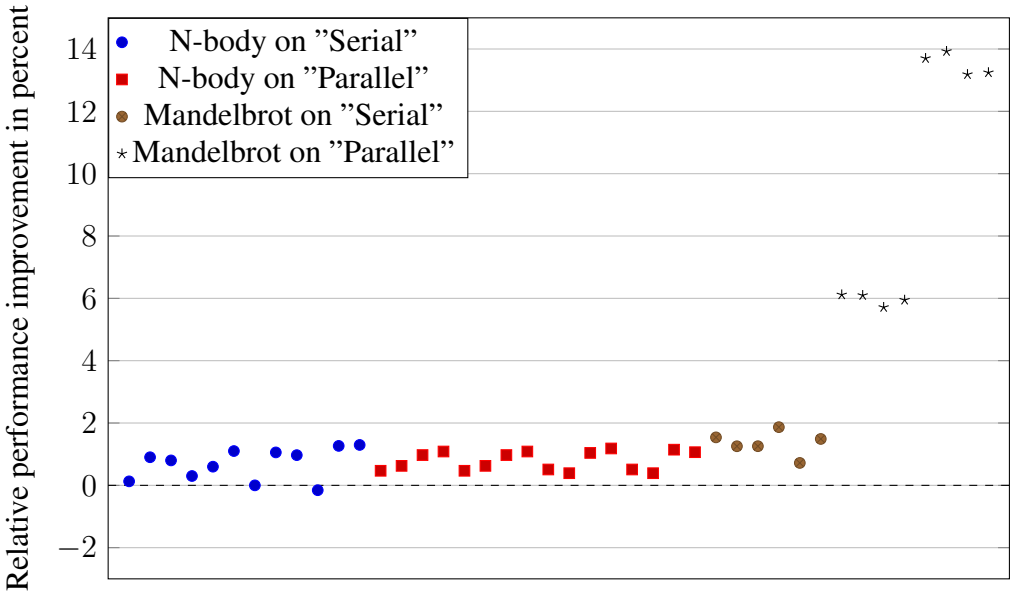

Figure 5.4: Relative performance improvement through the use of ALU-and-branch instructions.

Figure 5.5 shows that the amount of reduction in the required instruction count for the benchmark is almost the same between the ”Serial” and ”Parallel” model, but they react differently in simulation. The low benefit for the ”Serial” model shows no limitation by the raw instruction throughput, nor the dependency chains that get removed by combining the ALU and the branch operation, while the ”Parallel”

model struggles in this case. The simulation did not provide enough metrics to determine the exact cause.

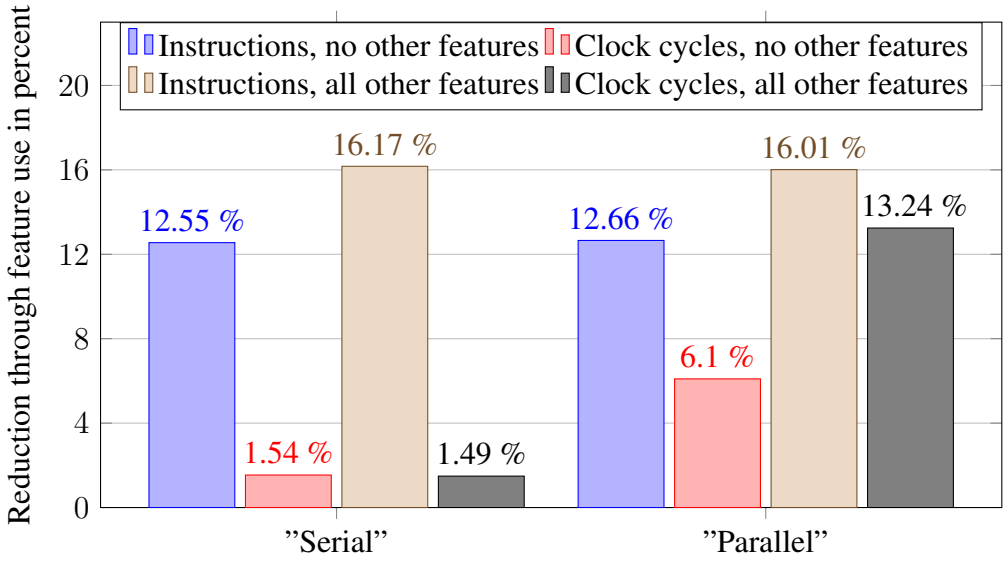

Figure 5.5: Reduction of the amount of required instructions and clock cycles for the Mandelbrot benchmark using ALU-and-branch instructions.

5.2.3 Instructions with memory operand

As explained in chapter 4.4.3, the Mandelbrot benchmark does not use this instruction type, so only the results for the N-body benchmark can be examined. This parameter exhibits the opposite property of the previous ones: The ”Serial” model shows significant performance gains of 16 to 20 percent, while the ”Parallel” model shows so little change that it would drown in the noise of any non-deterministic benchmark, as shown in figure 5.6.

As seen in chapter 5.2.1, the ”Serial” model reacted slightly more to parameter changes when multi-word formats, variable length vector support and no vector loops are combined. That also is the case for the parameter of instructions with a memory operand.

Interestingly, the reduction in required clock cycles for the ”Serial” model running the N-body benchmark scaled well beyond the instruction reduction, as seen in figure 5.7, which points to the ”Serial” model having severe trouble executing the ALU and memory operations in their split form. However, the existence of a bug in the CPU simulator also cannot be ruled out because it did not deliver enough metrics to investigate further.

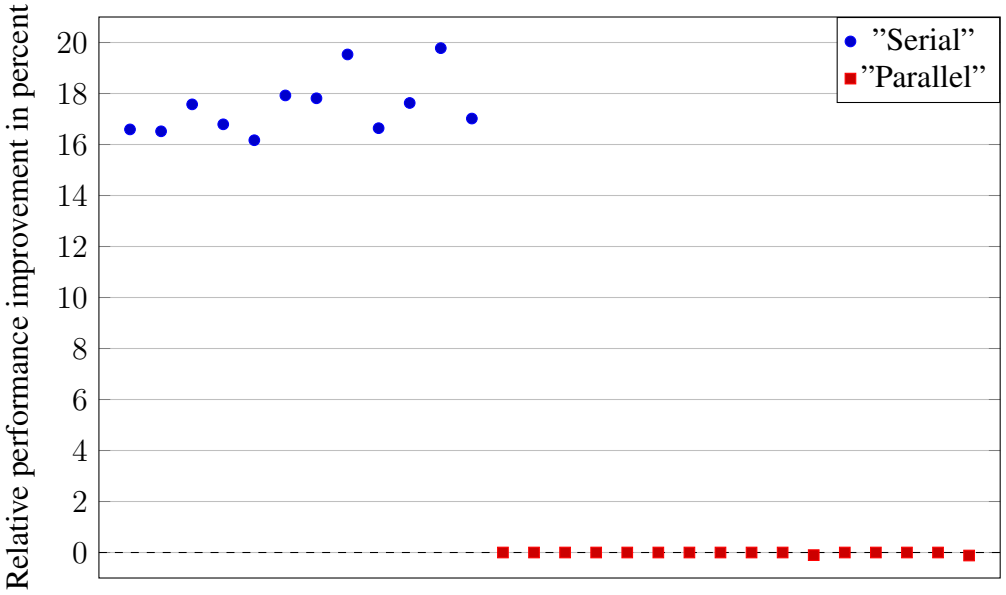

Figure 5.6: Relative performance improvement through the use of instructions with a memory operand on N-body.

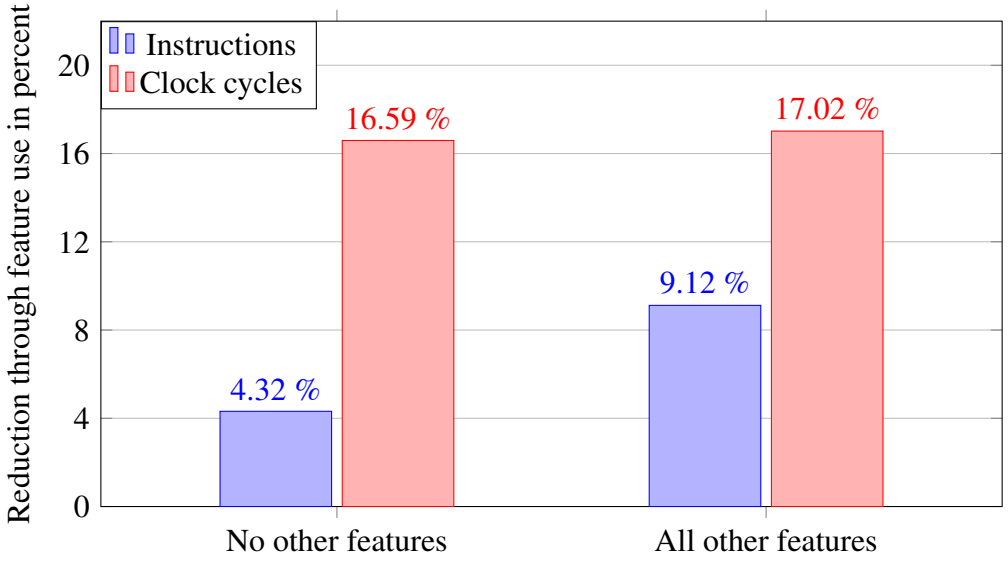

Figure 5.7: Reduction of the amount of required instructions and clock cycles for the N-body benchmark on the ”Serial” CPU model for instructions with a memory operand.

5.2.4 Variable vector length

As this parameter mainly changes compatibility for programs to CPU models with a different maximum vector length supported, it is not expected to always provide a positive performance impact. While the N-body benchmark mostly shows a performance improvement, sometimes significantly so, the Mandelbrot benchmark results are largely unaffected for the ”Serial” model, while the ”Parallel” model shows a slight performance regression in figure 5.8.

The results for the N-body benchmark are split into two sets again. In combination with the vector loops feature, the ”Serial” model sees an improvement of three to five percent, else it shows a regression of up to two percent, unless all previous parameters have their feature in use. The performance improvement for the ”Parallel” model sinks from 20 to 22 percent down to five to nine percent if vector loops are not used.

On the Mandelbrot benchmark, the ”Serial” performed better when multi-word instructions were allowed, while vector loops were not. This is the only combination that resulted in an actual performance benefit of two to four percent. Meanwhile, the ”Parallel” model always exhibited a slight regression of one to three percent.

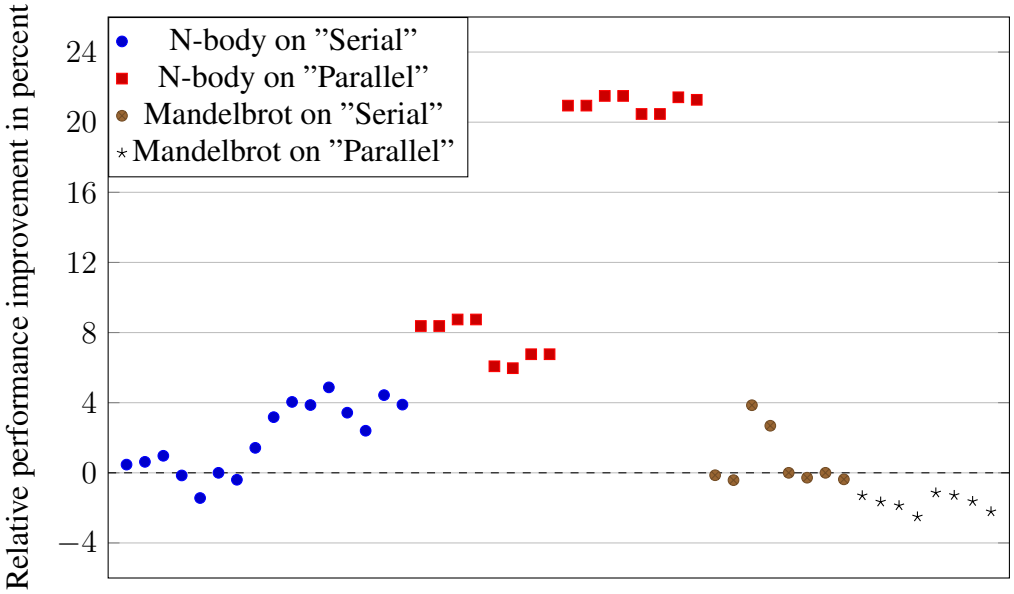

Figure 5.8: Relative performance improvement of variable vector length support compared to static native length. Includes combination with vector loops.

A more detailed inspection of the results for the N-body benchmark in figure 5.9

reveals that while variable vector support does increase the amount of instruction needed for the benchmark on average, the instructions performed per clock cycle increase by a larger percentage, resulting in improved total performance in most cases.

This implies that both models execute the folding of a vector more efficiently than the handling of leftover vector elements. However, one should be especially careful to not generalize this observation, because it can easily be caused by the branch predictor not handling the branch pattern well, for example.

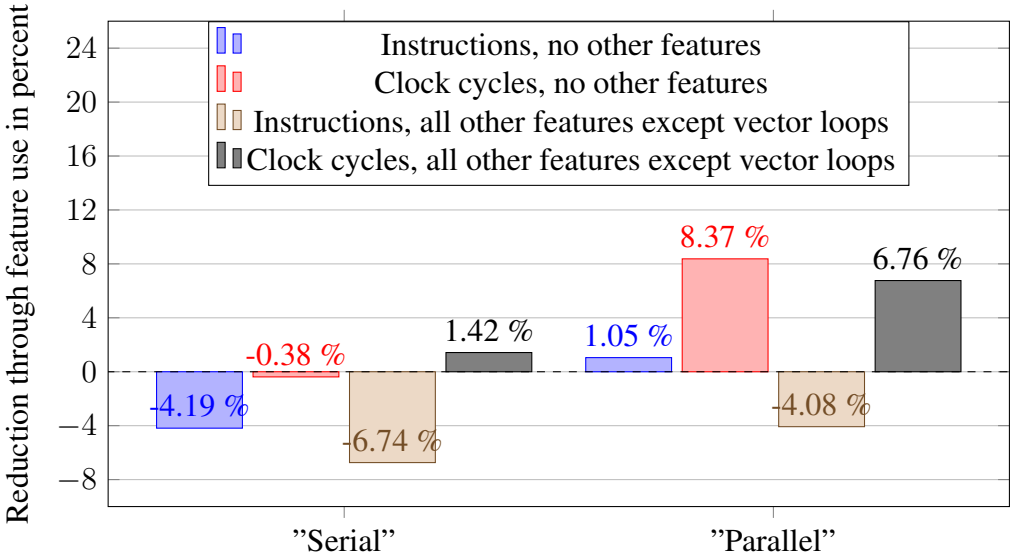

Figure 5.9: Reduction of the amount of required instructions and clock cycles for the N-body benchmark using variable vector support.

If the ”Parallel” model would otherwise get a program with a used static vector length of 128 bits for the sake of compatibility, removing this restriction with variable vector supports results in quite large performance gains, as shown in figure 5.10. The N-body benchmark shows another jump from the previous five to nine percent without vector loops to a range of 21 to 25 percent. With vector loops, the improvement rises from 20 to 22 percent to a range of 38 to 40 percent.

As mentioned in chapter 4.3.1, the Mandelbrot benchmark scales very well with the usable vector length. Having quadruple the resources in vector length, the performance of the ”Parallel” model explodes with an improvement of 290 to 296 percent.

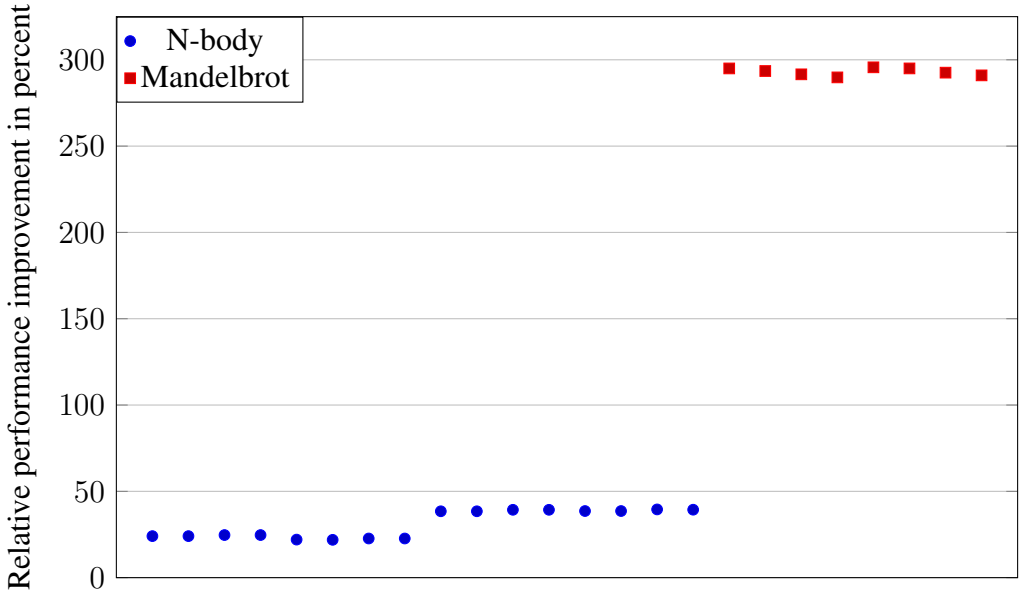

Figure 5.10: Relative performance improvement of variable vector length support on the ”Parallel” model compared to fixed 128 bit vectors.

5.2.5 Vector loops

As mentioned in chapter 5.2.4, combining variable vector support and vector loops results in a performance improvement for the N-body benchmark. This is shown in figure 5.11, but it also shows no significant impact on the Mandelbrot benchmark for the ”Parallel” model, with a regression for the ”Serial” CPU model.

Investigating this regression in figure 5.12 reveals that the amount of executed instructions behaves as expected, while the measured performance reacts negatively to the feature. Similar to chapter 5.2.3, this behavior can point to a special circumstance in the ”Serial” model, or to an error in the simulation.

5.3 Characterization

In this experiment, all parameters proved they can impact the execution speed in a variety of situations. Changing them to introduce a feature resulted in a performance benefit in most cases. However, the differences in results between the benchmark programs, as well as the CPU models, show that the results can in no way be taken as a general behavior of any program or CPU. Programs or CPU models outside of the tested ones could show very different results yet again. That being said, as the intent of this thesis is to characterize the impact of those param-

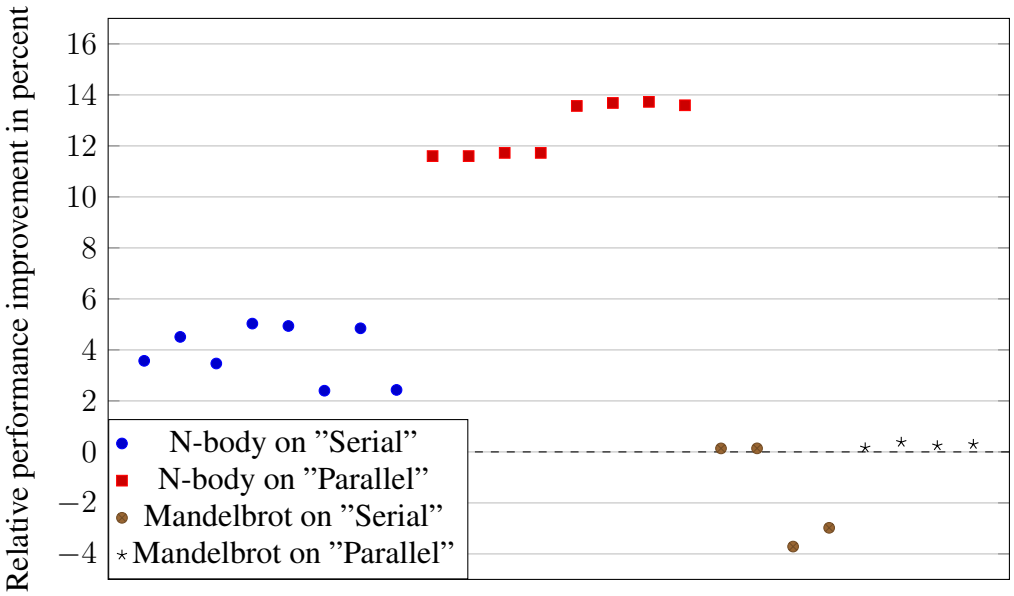

Figure 5.11: Relative performance improvement through the use of vector loops.

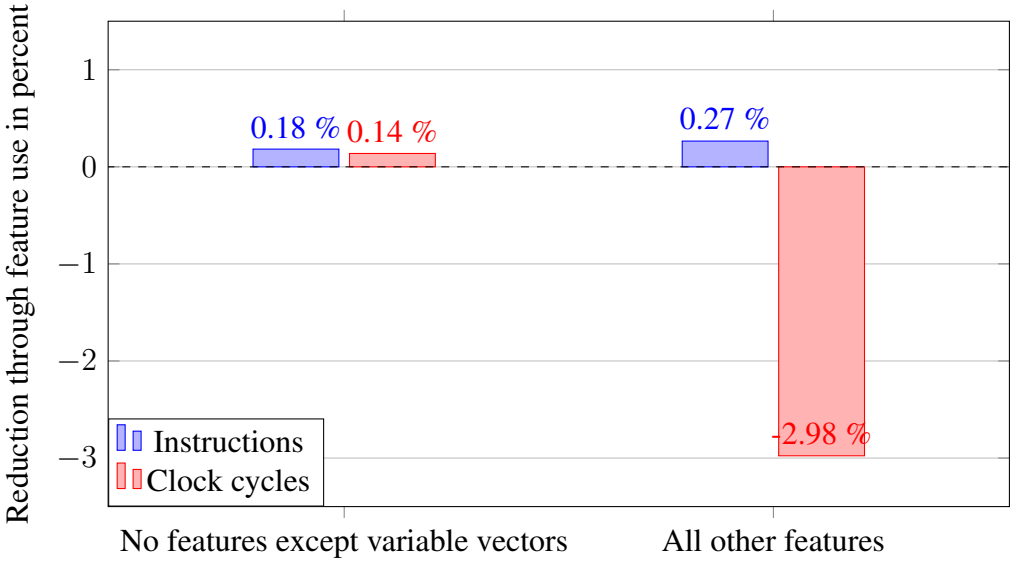

Figure 5.12: Reduction of the amount of required instructions and clock cycles for the Mandelbrot benchmark on the ”Serial” model using vector loops.

eters, the results can give a first glimpse.

5.3.1 Parameters

Starting with multi-word formats, they provide a performance uplift in nearly all cases, sometimes significantly so. They provide synergy with most other parameters, amplifying their impact. This experiment explicitly makes no assumption on the implementation cost, but we can assume from the existing implementation[37, ch. 11.6] that the cost is small. Considering this, as well as the test results, it makes multi-word formats a worthwhile feature to include in the ISA.

Moving on to ALU-and-branch instructions, this parameter provided little absolute impact in most cases, but did result in a double-digit performance improvement using the feature in a specific combination of program and CPU model. It can be argued that the experiment did not include cases that especially benefit from this feature, such as short loops or a separate data path for simple and branch instructions inside a CPU, as proposed by the ForwardCom manual [4, ch. 8.3]. Still, considering the conceptual simplicity of this parameter, the resulted impact can be described as significant.

The results for instructions with a memory operand are more difficult to evaluate. The sample size is even smaller than for the other parameters because the Mandelbrot benchmark does not make use of the feature. On the N-body benchmark, the difference of impact between both CPU models raises a few questions. The ”Parallel” model shows almost no reaction, while the ”Serial” model experiences a double-digit performance improvement. This parameter definitely needs further evaluation. Going off the obtained results, it did provide an impact, but considering its potential influence on the pipeline design [37, ch. 11.4], its significance relative to its complexity is inconclusive.

Arriving at variable vector length support, the potential impact of this parameter is major. Solving the compatibility issue between different vector sizes can cause a performance improvement of several hundred percent in extreme cases. Most importantly, the results show no significant performance regression between a native static length and a variable vector length, even when the program can be optimized for a specific one.

Finally, vector loops also have a significant impact, considering it only affects one addressing mode and one ALU-and-branch instruction, but it also has restrictions. The observed regression case is very specific and therefore requires further investigation.

5.3.2 Total impact

In combination, the tested parameters can provide a significant impact on performance on a magnitude of one to two hardware generations [38]. This is a major performance improvement, but again has to be put into perspective, especially since high performance processors can mitigate some ISA inefficiencies though techniques, such as fusion of instructions [17]. All in all, the results show that the tested parameters of the ForwardCom ISA enable comparatively efficient and fast program execution in processor implementations without resorting to advanced techniques in the micro-architecture.

Chapter 6

Conclusion

This thesis examined a first few parameters of the ForwardCom instruction set architecture by an experiment. As the ISA is still missing complete hardware implementations and critical parts of the software ecosystem, the examination is performed via an experimental characterization using CPU simulation on GPCAS. The tested parameters are multi-word formats, ALU-and-branch instructions, instructions with a memory operand, variable vector length support, and vector loops. The tested CPU models are simple, two-wide super-scalar designs, one with out-of-order execution capability, the other with large vector units.

The results reveal that, in the test environment, all tested parameters can provide a substantial performance impact, but the individual results can vary significantly. As most parameters show the intended positive impact when using their corresponding feature, this can be seen as a justification for their implementation in the ForwardCom ISA.

The variance of the experiment results emphasizes that they can only be used as a rough indicator for the parameter effects, not as predictive results. To get a more nuanced insight, further tests using additional, more sophisticated benchmarks and/or CPU models should be performed. Advancements in the ecosystem, such as a better simulator, hardware implementations, and most importantly a compiler, should enable and/or greatly simplify those test cases.

Bibliography

[1] M. Coppock, "Why Windows on ARM still couldn't catch up this year." https://www.digitaltrends.com/computing/windows-on-arm-still-didnt-catch-up-this-year/, Dez. 2022. Accessed Apr. 10, 2023.

[2] M. Korolov, "Arm Chips Gaining in Data Centers, But Still in Single Digits." https://www.datacenterknowledge.com/arm/arm-chips-gaining-data-centers-still-single-digits, Aug. 2022. Accessed Apr. 10, 2023.

[3] R. Amadeo, "RISC-Y Business: Arm wants to charge dramatically more for chip licenses." https://arstechnica.com/gadgets/2023/03/risc-y-business-arm-wants-to-charge-dramatically-more-for-chip-licenses/, Mar. 2023. Accessed Apr. 10, 2023.

[4] A. Fog, *ForwardCom: An open-standard instruction set for high-performance microprocessors*. https://raw.githubusercontent.com/ForwardCom/manual/master/forwardcom.pdf, 1.12 ed., Mar. 2023. Accessed Apr. 6, 2023.

[5] IBM, https://dl.acm.org/doi/pdf/10.5555/1102026, *IBM System/360 Principles of Operation*, first edition ed., 1964. Accessed Apr. 6, 2023.

[6] ARM, https://developer.arm.com/documentation/102374/0101, *Learn the architecture - A64 Instruction Set Architecture*, 1.1 ed., Nov. 2022. Accessed Apr. 6, 2023.

[7] ARM, https://developer.arm.com/documentation/102412/0103, *Learn the architecture - AArch64 Exception Model*, 1.3 ed., Dec. 2022. Accessed Apr. 6, 2023.

[8] Intel, https://cdrdv2.intel.com/v1/dl/getContent/671200, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, Dec. 2022. Accessed Apr. 6, 2023.

[9] ARM, https://developer.arm.com/documentation/ddi0602/latest/, *Arm A64 Instruction Set for A-profile architecture*, Dec. 2022. Accessed Apr. 6, 2023.

[10] SiFive Inc. and University of California at Berkeley, https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, 20191213 ed., Dec. 2019. Accessed Apr. 6, 2023.

[11] *System V Application Binary Interface AMD64 Architecture Processor Supplement*. https://www.uclibc.org/docs/psABI-x86_64.pdf, 0.99.7 ed., Nov. 2014. Accessed Apr. 6, 2023.

[12] D. A. Patterson and D. R. Ditzel, "The Case for the Reduced Instruction Set Computer," tech. rep., University of California at Berkeley and Computing Science Research Center at Murray Hill, New Jersey, https://inst.eecs.berkeley.edu/ n252/paper/RISC-patterson.pdf, 1980. Accessed Apr. 6, 2023.

[13] Intel, https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A-Z*, Dec. 2022. Accessed Apr. 6, 2023.

[14] G. Radin, "The 801 Minicomputer - An Overview," tech. rep., IBM, http://www.bitsavers.org/pdf/ibm/system801/The_801_Minicomputer_an_Overview_Sep76.pdf, Oct. 1976. Accessed Apr. 6, 2023.

[15] A. Akram, "A Study on the Impact of Instruction Set Architectures on Processor's Performance," Master's thesis, Western Michigan University, https://scholarworks.wmich.edu/cgi/viewcontent.cgi?article=2517&context=masters_theses, Aug. 2017. Accessed Apr. 5, 2023.

[16] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design - Next Generation of x86 Integrates L2 Cache in Package with CPU," *Microprocessor Report*, vol. 9, Feb. 1995.

[17] C. Celio, D. Dabbelt, D. A. Patterson, and K. Asanović, "The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V," Tech. Rep. UCB/EECS-2016-130, University of California, Berkeley, http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-130.pdf, Jul. 2016. Accessed Apr. 6, 2023.

[18] M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Sep. 1972.

[19] AMD, “4th gen AMD EPYC processor architecture.” https://www.amd.com/en/campaigns/epyc-9004-architecture, Nov. 2022. Accessed Apr. 6, 2023.

[20] Khronos®OpenCL Working Group, https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, *The OpenCL™Specification*, 3.0.13 ed., Feb. 2023. Accessed Apr. 6, 2023.

[21] RISC-V, https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf, *RISC-V "V„ Vector Extension*, 1.0 ed., Sep. 2021. Accessed Apr. 6, 2023.

[22] ARM, https://developer.arm.com/documentation/102159/0400, *Neon Programmer Guide for ARMv8-A*, 4.0 ed., Jul. 2022. Accessed Apr. 6, 2023.

[23] ARM, https://developer.arm.com/documentation/102340/0100, *Learn the architecture - Introducing SVE2*, 1.0 ed., May 2022. Accessed Apr. 6, 2023.

[24] A. Fog, “Proposal for an ideal extensible instruction set.” https://www.agner.org/optimize/blog/read.php?i=421, Dez. 2015. Accessed Apr. 6, 2023.

[25] A. Fog, “bintools.” https://github.com/ForwardCom/bintools, Nov. 2017. Accessed Apr. 6, 2023.

[26] “riscv-v-spec.” https://github.com/riscv/riscv-v-spec, Sep. 2021. Accessed Apr. 6, 2023.

[27] A. Fog, “Re: Proposal to drop tiny instructions.” https://www.forwardcom.info/forum/viewtopic.php?f=1&t=133#p365, May 2020. Accessed Apr. 6, 2023.

[28] Intel, https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, Dec. 2022. Accessed Apr. 6, 2023.

[29] SiFive Inc. and University of California at Berkeley, https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, 20211203 ed., Dec. 2011. Accessed Apr. 6, 2023.

[30] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, (https://spectreattack.com/spectre.pdf), 2019. Accessed Apr. 6, 2023.

[31] K. Rese, "General Purpose Core Architecture Simulator." https://git.h3n.eu/gpcas/gpcas, Apr. 2023. Accessed Apr. 6, 2023.

[32] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, sixth edition ed., 2017.

[33] I. Gouy, "The computer language benchmarks game." https://salsa.debian.org/benchmarksgame-team/benchmarksgame, Mar. 2023. Accessed Apr. 6, 2023.

[34] B. Henderson, "pbm - netpbm bi-level image format." https://netpbm.sourceforge.net/doc/pbm.html, Nov. 2013. Accessed Apr. 6, 2023.

[35] J. Zhao, A. Gonzalez, B. Korpan, and K. Asanović, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, (https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf), University of California, Berkeley, May 2020.

[36] AMD, ""JAGUAR", AMD's Next Generation Low Power x86 Core." https://old.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.120-Jaguar-Rupley-AMD.pdf, Aug. 2012. Accessed Apr. 6, 2023.

[37] A. Fog, *ForwardCom softcore model A*. https://raw.githubusercontent.com/ForwardCom/softcoreA/main/manual/softcore_A.pdf, a1.01 ed., Dez. 2022. Accessed Apr. 4, 2023.

[38] AMD, "AMD Launches Ryzen 7000 Series Desktop Processors with "Zen" Architecture: the Fastest Core in Gaming." https://d1io3yog0oux5.cloudfront.net/_48cf8cf4cff97455a2762f32deb4a8ee/amd/news/2022-08-29_AMD_Launches_Ryzen_7000_Series_Desktop_Processors__1089.pdf, Aug. 2022. Accessed Apr. 4, 2023.

Glossary

Address Generation Unit A unit inside a processor that calculates the effective address of memory operands. 24, 55

Arithmetic and Logic Unit A unit inside a processor that performs arithmetical and logical operations. 55

Branch target buffer A buffer holding the target addresses of recently encountered branches. It can also include a simple bimodal predictor, indicating if a branch will be taken the next time it is encountered . 33, 55

Field Programmable Gate Array A microchip that features programmable logic. Can be used to run different hardware models, but at a lower performance than a hardware chip dedicated to that model . 14

Vector Arithmetic and Logic Unit A unit inside a processor that performs arithmetical and logical operations on multiple element simultaneously . 56

Acronyms

ABI Application Binary Interface. 9, 29

AGU Address Generation Unit. 24, 25, 35, 36, *Glossary:* Address Generation Unit

ALU Arithmetic and Logic Unit. 11, 15, 24, 25, 33, 35, 36, *Glossary:* Arithmetic and Logic Unit

BTB Branch Target Buffer. 33, 35, 36, *Glossary:* Branch target buffer

CISC Complex Instruction Set Computer. 10

CPU Central Processing Unit. 5, 10, 11, 14, 23, 25, 27, 33, 37, 40–43, 45, 47, 49

CSR Control and Status Register. 15, 18

GPCAS General Purpose Core Architecture Simulator. 23, 26, 28, 49

ISA Instruction Set Architecture. 4–6, 8–16, 18–21, 23, 24, 26, 27, 33, 39, 47–49

JSON JavaScript Object Notation. 23

MIMD Multiple Instructions, Multiple Data. 11

RISC Reduced Instruction Set Computer. 4, 10

ROB Reorder Buffer. 33, 35

SIMD Single Instruction, Multiple Data. 11, 12

SISD Single Instruction, Single Data. 11

VALU Vector Arithmetic and Logic Unit. 35, 36, *Glossary:* Vector Arithmetic and Logic Unit

Appendix

All appendixes are provided via the included CD-ROM.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

"Experimental characterization of the effect of various ForwardCom architecture parameters on microprocessor performance"

ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel anfefertigt habe. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mönchengladbach, 11.04.2023 — Kai Rese

Ort, Datum — Unterschrift